Escola Politècnica Superior
de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

MASTER THESIS

TITLE: Development of a wireless sensor network with 6LoWPAN support

MASTER DEGREE: Master in Science in Telecommunication Engineering & Management

AUTHOR: Ana de Pablo Escolà

DIRECTOR: Josep Polo Cantero

DATE: July 10th 2009

Title: Development of a wireless sensor network with 6LoWPAN support

Author: Ana de Pablo Escolà

Director: Josep Polo Cantero

Date: July, 10th 2009

Overview

The aim of this project is to develop a wireless sensor network with 6LoWPAN support. Along this document wireless sensor network (WSN) and its technologies are presented from a global point of view, but emphasizing 6LoWPAN.

To design and develop the network it has been necessary the use of a specific operative system, TinyOS, specially used in wireless sensor networks. It has been also required the use of different programming languages such as NesC and Python, and the use of IPv6, one of the main topics in the 6LoWPAN specification.

The 6LoWPAN support of the network is achieved basing the development on an application designed by Berkeley WEBS group. During this project, a newest version of the one being in use was released; and it was necessary to adapt it to the project's application.

Finally, an environmental wireless sensor network that can be located in the campus has been developed; including MySQL database and a server to allow the users to make queries through a website.

INDEX OF FIGURES AND TABLES

INTRODUCTION

In our daily lives we are constantly surrounded by thousand of wireless networks, and in the last years, the number of those networks related with sensors has been increasing.

Some recent technologies like ZigBee and Bluetooth have impulse such increment and have opened the wireless sensor networks to a large number of applications. But as the wireless technologies evolve so should the wireless sensor networks, they are able to connect to the Internet, but not only that, they are already adapted to work with IPv6, the future network addresses of any device connected to the Internet.

The first chapter of this project introduces the wireless sensor networks and their technologies and standards in which are based. The 6LoWPAN and the IPv6 standard are explained and focused to their application in the wireless sensor networks.

The second section introduces how 6LoWPAN works to adapt the packets that are sent through the network to work under IPv6 addresses, that is, the adaptation layer that has been developed by the working group in the internet area of IETF.

The third chapter presents the software requirements of the project and the steps to their installation: TinyOS and Ubuntu. The hardware needed is also specified comparing different hardware platforms and their characteristics. At the end of this section the platform used in this project, TelosB, is presented including the sensors and their main features.

The design for the application is explained in chapter four. The main application has been developed by the Berkeley WEBS university group. Based on their code releases, our application has been designed to control the EPSC environment. In order to achieve this objective, in this chapter is presented how to implement the sensors and how to configure the network to join all the devices and start receiving all the data by using IPv6 sockets.

Once the data is collected, it is uploaded into a MySQL database to able future queries through a server. This is explained in the last section. The required software, XAMPP, its installation and configuration is explained in this chapter, and also the design on the website where the users can request information about the sensors included in the different devices.

Configuration files, tutorials of how to install TinyOS or work under IPv6 and other files are included in the annexes at the end of this document.

CHAPTER 1. OVERVIEW OF WSN

The aim of this chapter is to briefly introduce wireless sensors networks or WSN, their applications and technologies that have been developed recently to improve the efficiency of those networks.

Such networks are often deployed in resource-constrained environments, for instance with battery powered nodes running untethered.

The actual and future applications of those networks will also be introduced.

Included in the latest improvements of the WSN, it is 6LoWPAN, also known as the Internet of the things, the main topic that will be presented along this first chapter.

1.1 Wireless Sensor Network

A wireless sensor network or WSN is a wireless network consisting of spatially distributed autonomous devices. These devices operate untethered and have limited power resources, which limits and shapes all aspects of their architecture, including the node's processing, sensing, and communication subsystems.

In most applications, those devices use sensors to cooperatively monitor physical or environmental conditions. A sensor network is designed to give an administrator the ability to instrument, observe and react to events in the specified environment in which it is working.

The environment can be either the physical world or an information technology framework. That is why WSN can be used in many industrial and civilian applications, including industrial process monitoring and control, healthcare applications, habitat monitoring, home automation and many others.

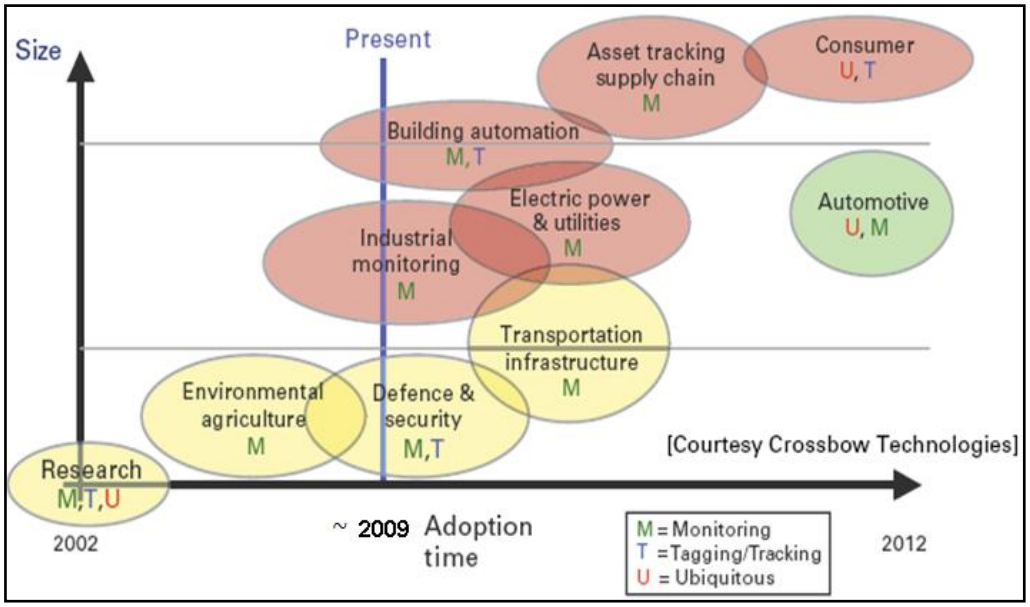

Fig. 1.1 WSN application areas

Networked sensing offers unique advantages over traditional centralized approaches. Dense networks of distributed communicating sensors can increase energy efficiency by the multi-hop topology of the network. The greatest advantages of networked sensing are robustness and scalability. A decentralized sensing system is inherently more robust against individual sensor node or link failures, because of redundancy in the network.

The main features and challenges of WSN can be summarized as:

- Low cost devices
- Energy-efficient devices
- End-to-end quality of service
- Seamless operation under context changes
- Secure operation
- Large scale of deployment
- Mobility of nodes

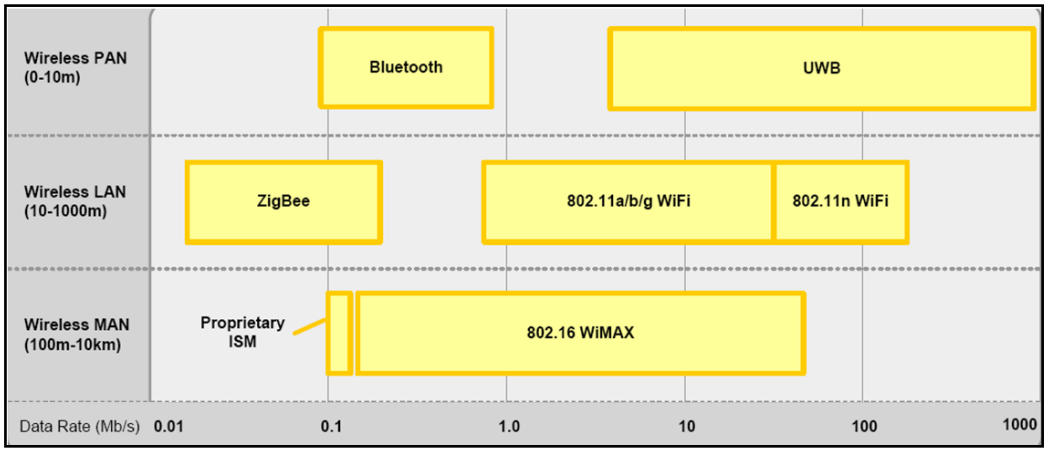

Fig. 1.2 Wireless technology comparison

A sensor network has to be capable of managing a large numbers of devices, even if those change their location, join or left the network. Usually, nearly all the devices are located far from the administrator and are battery powered. That is why they have to be energy-efficient devices, to avoid running down the batteries in a short period of time and, probably, to prevent losing information.

To accomplish all those challenges, the elements and interfaces that form the networks are numerous. It is necessary to have a user interface, a security system, power supply, network management, among others.

One of the main elements in those kinds of networks is the devices. Each device or node is typically equipped with the following:

- One or more sensing units
- A processing unit: a microcontroller
- A communication unit: a transceiver
- A power unit: normally batteries, but also solar panels or others power harvesting process

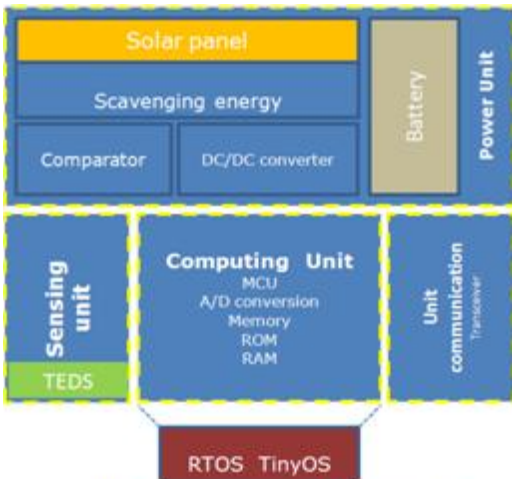

Fig. 1.3 Block diagram of a sensor node

The battery unit can be a combination between batteries and a solar panel; the A/D converter is used to recharge the batteries through the power harvested in the solar panel.

The nodes have limited capabilities and resources. They can be configured to read data from the environment and send it to a central point where all the information is collected through another device. Then the administrator is able to manage it, normally by developing appropriate software adapted to the specific application and with the use of higher capabilities equipment, such as a computer.

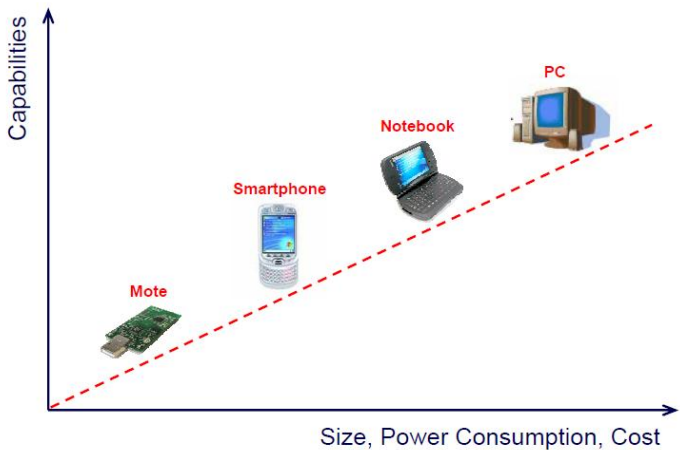

Fig. 1.4 Devices vs. capabilities

1.1.1 WSN Technologies

Several standards are currently either ratified or under development for wireless sensor networks.

- WirelessHART is an extension of the HART Protocol and is specifically designed for industrial applications like process monitoring and control.
- ZigBee is a mesh-networking standard intended for uses such as medical data collection, consumer devices like television remote controls, and home automation.
- 6LoWPAN is the IETF standards track specification for the IP-to-MAC-Layer mapping for IPv6 on IEEE 802.15.4.

WirelessHART devices communicate using Time Division Multiple Access (TDMA). Each WirelessHART device maintains a precise sense of time and remains synchronized with all neighbouring devices. All device-to-device communication is done in a pre-scheduled time window which enables very reliable (collision-free), power-efficient, and scalable communication.

ZigBee, WirelessHART, and 6lowpan all are based on the same underlying radio standard: IEEE 802.15.4.

Table 1.1 WSN Technologies

Standard	ZigBee	6LoWPAN	WirelessHART
Main application	Control and monitoring	Control and monitoring	Industrial control and monitoring
Memory	4 – 32 kB	4 – 32 kB	
Battery Lifetime (days)	100 – 1000+	100 – 365+	760+
Network nodes	255	65536	200
Throughput	Up to 250 Kbps	Up to 250 Kbps	Up to 250 Kbps
Range	1-75	1-100	1-100
Main feature	Reliability, low consume, low cost	IPv6 over IEEE 802.15.4	Reliability

1.1.2 IEEE802.15.4

IEEE 802.15.4 is a standard which specifies the physical layer and media access control (MAC) for low-rate wireless personal area networks (LR-WPANs). It is maintained by the IEEE 802.15 working group and the first version was completed in May 2003.

The IEEE 802.15.4 standard specifies a wireless interface meant for wireless embedded applications, such as building automation, industrial automation and

other sensing and tracking purposes. The standard is very flexible, allowing from ad-hoc mesh networks to infrastructure based tree topologies.

As said before, it is the basis for the ZigBee networking stack and WirelessHART, each of which further attempts to offer a complete networking solution by developing the upper layers (which are not covered by the standard). Alternatively, it can be used with 6LoWPAN and standard Internet protocols.

Table 1.2 IEEE 802.15.4 features

Frequency bands and data rates	868-868.8 MHz and 20 Kb/s 902-928 MHz and 40 Kb/s 2400-2483.5 MHz and 250 Kb/s
Range	10-20 m
Addressing	IEEE 64-bit addresses
Network nodes	Up to 2^{64} devices
Security	128 AES
Channel access	CSMA-CA

1.1.2.1 Physical layer

The physical layer (PHY) provides the data transmission service and performs channel selection and energy and signal management functions. It operates on one of three possible unlicensed frequency bands:

- 868-868.8 MHz: Europe, allows one communication channel
- 902-928 MHz: North America, up to ten channels, extended to thirty in 2006 revision
- 2400-2483.5 MHz: worldwide use, up to sixteen channels

The 2.4 GHz physical layer provides a 250 kbps data rate, but lower rates can be considered with the resulting effect on power consumption.

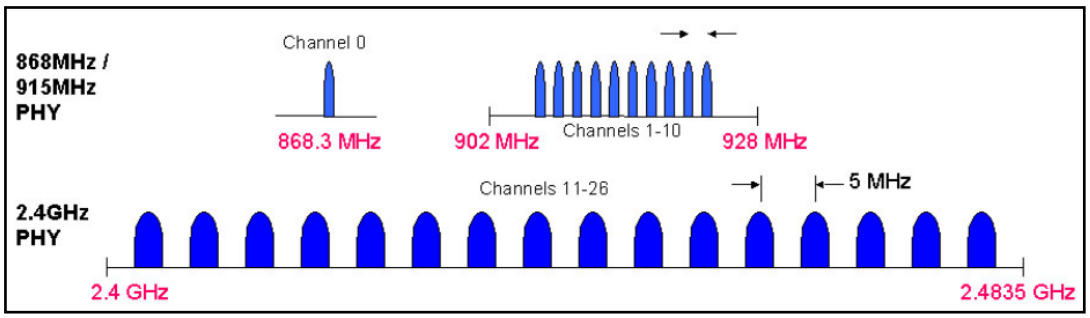

Fig. 1.5 IEEE 802.15.4 channel distribution

Consider a placement where various wireless networks can be present working at the same frequency bands. It is necessary to implement a dynamic selection of channels:

- MAC layer includes searching algorithms to find the best channel through the list of the possible ones.
- PHY layer implements some functions to detect the received energy, consider the quality of the channel and channel commutation.

In the next figure it is shown how wireless sensor networks can coexist with a WiFi (802.11) network without interfering.

The channels that can be used in IEEE 802.15.4 at 2.4 GHz are: 15, 20, 25, 26 and so.

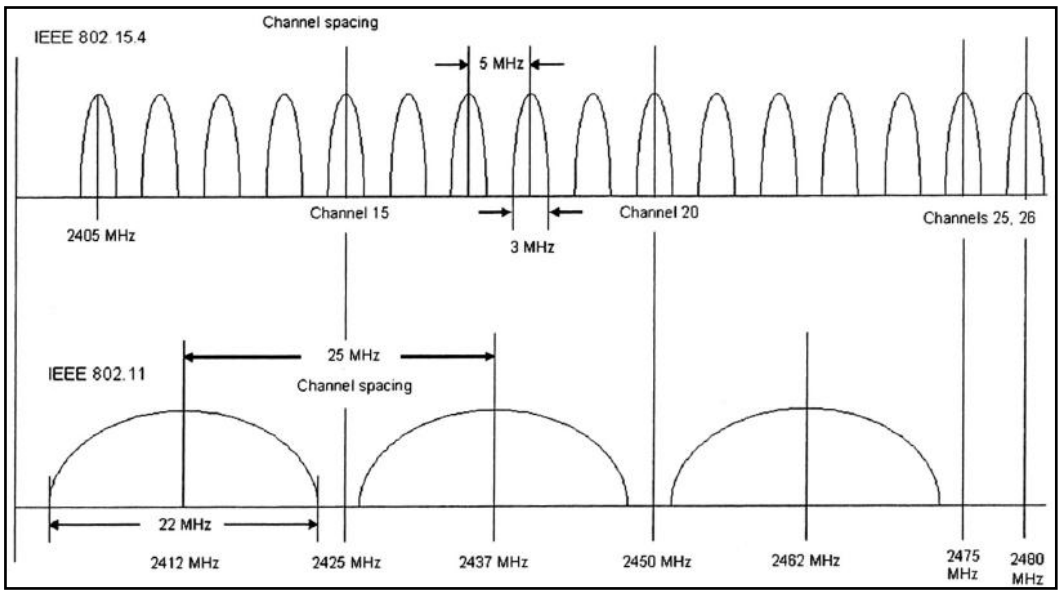

Fig. 1.6 IEEE 802.15.4 and IEEE 802.11 channels

The 2.4 GHz employs a 16-ary quasi-orthogonal modulation technique based on DSSS. Binary data is grouped into 4-bit symbols, each symbol specifying one of 16 nearly orthogonal 32-bit chip pseudo noise (PN) sequences for transmission. PN sequences for successive data symbols are concatenated and the aggregate chip is modulated onto the carrier using minimum shift keying (MSK). The use of nearly orthogonal symbol sets simplifies the implementation, but incurs minor performance degradation. In terms of energy conservation, orthogonal signalling performs better than differential BPSK. However, in terms of receiver sensitivity, the 868/915 MHz layer has a 6-8 dB advantage.

Modulation parameters are summarized in the following table.

Table 1.3 Modulation characteristics of the PHY layer

Frequency	Bandwidth	Symbol rate		Data characteristics		
		Chip rate Kchip/s	Modulation	Bit rate Kb/s	Symbol rate ksymbol/s	Symbols
865/915	868–868.6	300	BPSK	20	20	Binary
	902–928	600	BPSK	40	40	Binary
2450	2400–2483.5	2000	O-QPSK	250	62.5	16 orthogonal

1.1.2.2 MAC Layer

The medium access control sublayer (MAC) allows the transmission of MAC frames through the use of the physical channel and its fundamental goal is to reduce or avoid packet collisions in the medium.

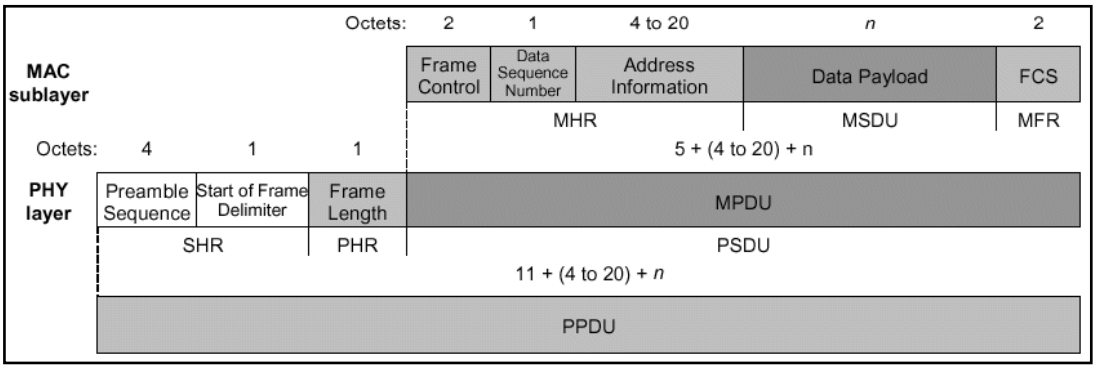

Fig. 1.7 MAC frame format

MAC and PHY layers share a single packets structure as shown in the figure above.

The MAC protocol data unit (MPDU) consists of the MAC header (MHR), MAC service data unit (MSDU) and MAC footer (MFR).

The MHR consists of a 2 byte frame control field that specifies the frame type, the address format and controls the acknowledgement, 1 byte sequence number which matches the ACK frame with the previous transmission, and a variable sized address field (4-20 bytes). This allows either only the source address or both source and destination address. The payload field is variable in length but the maximum possible size of an MPDU is 127 bytes. The MFR is a frame check sequence field which is basically a 16-bit CRC code.

The PHY protocol data unit (PPDU) consists of a 32-bit preamble or synchronization header, a PHY header for the packet length, and the payload

itself which is also referred to as the PHY service data unit (PSDU). The synchronization header is used for acquisition of symbol and chip timing and possible coarse frequency adjustment and an 8-bit start of packet delimiter. Out of the 8 bits in the PHY header, seven are used to specify the length of the PSDU which can range from 0-127 bytes. Typical packet sizes for monitoring and control applications are expected to be in the order of 30-60 bytes.

1.1.2.3 Nodes

Basically, there are two kinds of nodes: Full Function Devices (FFD) and Reduced Function Devices (RFD).

- FFD: It can serve as the coordinator of a PAN or as a common node. It implements a general model of communication which allows it to talk to any other device: it may also relay messages, in which case it is acting as a coordinator (PAN coordinator when it is in charge of the whole network).
- RFD: These are meant to be extremely simple devices with very modest resource and communication requirements. They can only communicate with FFD's and can never act as coordinators.

1.1.3 ZigBee

ZigBee is a standard ratified on 2004 by ZigBee Alliance. This group, which is responsible for its maintenance, announced public availability (for non-commercial purposes) of version 1.0 on June 2005, known as ZigBee 2004 Specification.

ZigBee is a specification for a suite of high level communication protocols using small, low-cost, low-power digital radios based on the IEEE 802.15.4-2003 standard. The technology defined by the ZigBee specification is intended to be simpler and less expensive than other WPANs, such as Bluetooth. ZigBee is targeted at radio-frequency (RF) applications that require a low data rate, long battery life, and secure networking.

1.1.3.1 Features

The main features of ZigBee standard are:

- Operate in different radio bands: 868 MHz in Europe, 915 MHz in the USA, and 2.4 GHz worldwide
- In the 2.4 GHz band there are 16 ZigBee channels, with each channel requiring 5 MHz of bandwidth and 250 kbit/s data rate
- The radios use direct-sequence spread spectrum (DSSS) coding

- Very low power consumption (ZigBee modules can sleep most of the time)
- Long battery life (individual devices must have a battery life of at least two years)
- Low cost
- Intended for use in embedded applications requiring low data rates
- Beacon and non-beacon enabled networks, in the second case power consumption is asymmetrical

1.1.3.2 Applications

Those mentioned above and other features allow ZigBee to have some typical application areas such as:

- Home Entertainment and Control: Smart lighting, advanced temperature control, safety and security
- Home Awareness: Water sensors, power sensors, smoke and fire detectors
- Mobile Service: mobile monitoring and control, security and access control, m-healthcare and teleassistance
- Commercial Building: Energy monitoring, lighting, access control
- Industrial Plant — Process control, environmental management, energy management, industrial device control

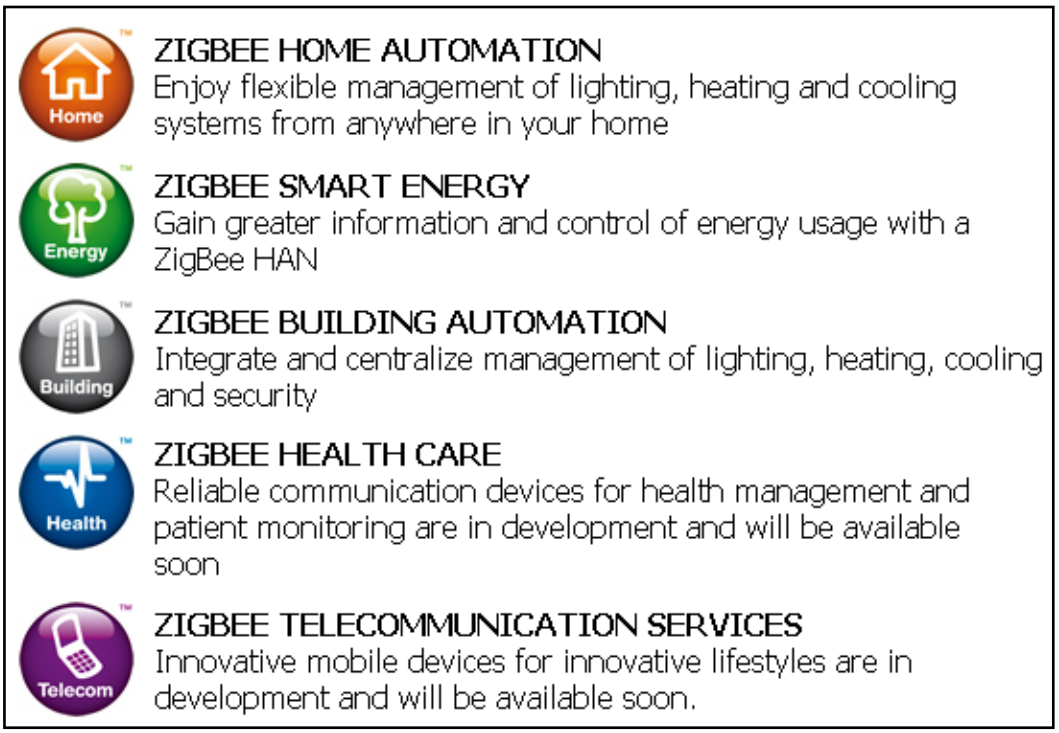

Fig. 1.8 ZigBee application areas

1.2 Introduction to 6LoWPAN networks

6LoWPAN is an acronym of IPv6 over Low power Wireless Personal Area Networks. 6lowpan is the name of the working group in the internet area of IETF and a specification to allow the use of IPv6 over IEEE 802.15.4 networks, defined in RFC4919.

The 6LoWPAN group aimed at defining header compression mechanisms that allow IPv6 packets to be sent to and received from over 802.15-based networks. Applications for 6LoWPAN networks are made in a very similar way to the Internet.

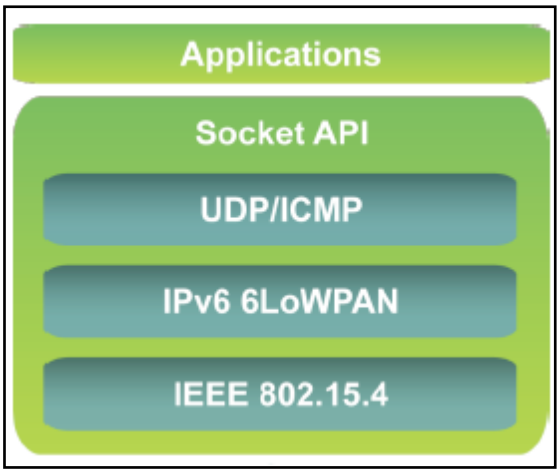

Fig. 1.9 6LoWPAN and OSI model

1.2.1 Characteristics

Those are the main features of the 6LoWPAN specification:

- Small packet size: considering a maximum PHY packet size of 127 bytes and the worst case of overhead from the above layers, it will leave 81 octets for data packets.
- Support for both 16-bit short (unique within a PAN) or IEEE 64-bit extended MAC addresses
- Low bandwidth. Data rates of 250 kbps, 40 kbps, and 20 kbps for each previously defined physical layers
- Topologies include star and mesh operation
- Low power and low cost devices
- Large number of devices supported

- Location of the devices is typically not predefined, and additionally, these devices may move to new locations.
- 6LoWPAN devices may sleep for long periods of time in order to save energy

Devices in 6LoWPAN can be divided in two groups as mentioned in IEEE 802.15.4 section: full function and reduced function devices. FFDs will typically have more resources and may be main powered. Accordingly, FFDs will aid RFDs by providing functions such as network coordination, packet forwarding, interfacing with other types of networks, etc.

1.2.2 6LoWPAN and ZigBee

The main differences between both technologies can be summarized as follows:

- ZigBee establishes small-scale isolated ad-hoc networks and 6LoWPAN performs massively scalable networks as a part of the Internet (IPv6)
- ZigBee is limited to a single radio standard and only defines communication between 15.4 nodes. 6LoWPAN can be applied to any low-power, low-rate wireless radio.
- Zigbee defines new upper layers, rather utilizing existing standards.
- ZigBee is not a standard, it is a special interest group. The IETF produces open, long-lived, standards.

1.2.3 IP technology in LoWPANs

The application of IP technology is assumed to provide the following benefits:

- The pervasive nature of IP networks allows use of existing infrastructure
- IP-based technologies already exist, are well-known, open and free, and proven to be working
- Tools for diagnostics, management, and commissioning of IP networks already exist
- IP-based devices can be connected readily to other IP-based networks, without the need for intermediate entities like translation gateways or proxies

The last topic consider the case that if, in addition to IEEE 802.15.4, the devices user other kinds of networks interfaces such as Ethernet or IEEE 802.11, the goal is to easily integrate the existing networks over those technologies.

But there are also some requirements for IP within LoWPANs that have to be considered:

- The many devices in a LoWPAN make network auto configuration and statelessness highly desirable. And for this, IPv6 has ready solutions
- The large number of devices poses the need for a large address space, well met by IPv6
- Given the limited packet size (81 bytes in the worst case), headers for IPv6 and layers above must be compressed whenever possible

1.2.4 Routing considerations

LoWPANs support various topologies (see Figure 1.5) including mesh, ring, fully connected and star. Mesh topologies imply multi-hop routing. In this case, intermediate devices act as packets forwarders to the neighbours at the link layer (as routers do) and, typically, these are FFDs.

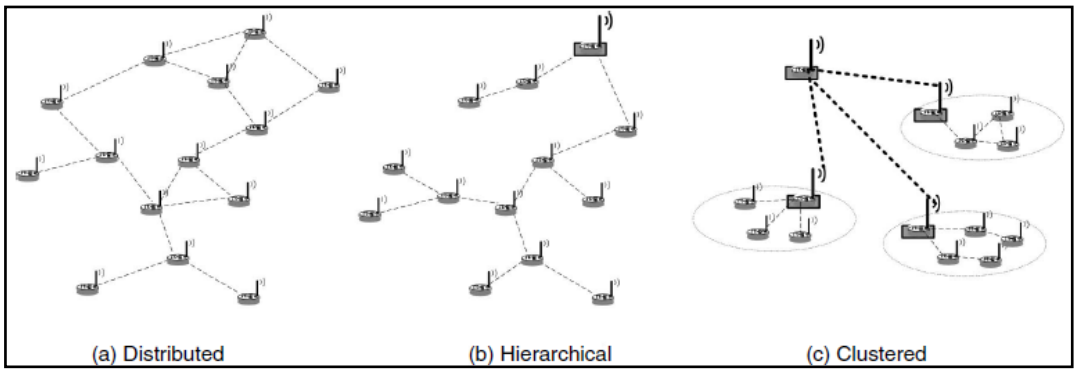

Fig. 1.10 Sensor network architectures

Some considerations on the routing protocol are:

- The routing protocol must impose low (or no) overhead on data packets
- They should have low overhead balanced with topology changes and power conservation
- To satisfy low cost and low power, the computation and memory required should be minimal. Thus, manage large routing tables is harmful.
- Appropriate considerations in case of presence of sleeping nodes

1.2.5 Functions

The main goal of the different functions implemented in the specification is to reduce packet overhead, bandwidth consumption, power consumption and processing requirements.

- Adapting the packet sizes: the data unit's size is 81 octets, far below the minimum IPv6 packet size of 1280 octets. A fragmentation and reassembly adaptation layer must be provided at the layer below IP
- Header compression: working with 81 octets and IPv6 header of 40 bytes, leaves 41 octets for upper-layer protocols like UDP. UDP uses 8 octets, and fragmentation/reassembly layer will use even more octets leaving very few bytes for data. This points the need for header compression.
- Address resolution: IPv6 nodes are assigned 128 bit IP addresses in a hierarchical manner, through an arbitrary length network prefix. 6LoWPAN specifies methods for creating IPv6 stateless address, reducing the configuration overhead on the hosts.

1.2.6 Security

6LoWPAN applications often require confidentiality and integrity protection. This can be provided at the application, network or at the link layer. Some constraints may influence the choice of a particular protocol: small code size, small bandwidth, low power...

Some threats that should be considered when choosing the protection are man-in-the-middle and denial of service attacks.

One reason for using link layer security is that most IEEE 802.15.4 devices already support AES link-layer security. For network layer security, there are two models: end-to-end or security limited to the wireless portion of the network (e.g. firewall).

1.3 IPv6

Internet Protocol version 6 (IPv6) is an Internet Layer protocol for packet-switched internetworks and the Internet. IPv4 is currently the dominant Internet Protocol version, and was the first to receive widespread use.
In December 1998, the Internet Engineering Task Force (IETF) designated IPv6 as the successor to version 4.

IPv6 has a much larger address space than IPv4. This results from the use of a 128-bit address, whereas IPv4 uses only 32 bits. The new address space thus supports 2^{128} (about 3.4×10^{38}) addresses. This expansion provides flexibility in allocating addresses and routing traffic and eliminates the primary need for network address translation (NAT).

IP runs over virtually any underlying communication technology, ranging from high-speed wired Ethernet links to low-power 802.15.4 radios and 802.11 (WiFi)

equipment. For long-haul communication, IP data is easily transported through encrypted channels over the global Internet.

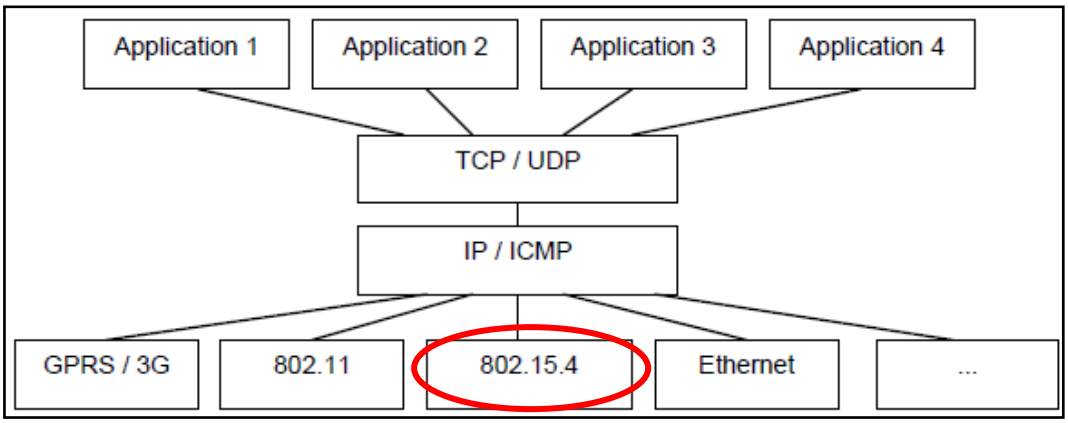

Fig. 1.11 IP architecture

The general requirements for implementing IPv6 on a network host are specified in RFC 4294.

1.3.1 Addressing

IPv6 addresses are typically composed of two logical parts: a 64-bit (sub) network prefix, and a 64-bit host part, which is either automatically generated from the interface's MAC address or assigned sequentially.

IPv6 addresses are normally written as eight groups of four hexadecimal digits, where each group is separated by a colon (:). To shorten the writing and presentation of addresses, several simplifications to the notation are permitted:

- Leading zeros in a group may be omitted
- Any number of consecutive groups of 0 value may be replaced with two colons (::), but only once in an address

`2001:0db8:0000:0000:0000:0000:1428:57ab` → `2001:db8::1428:57ab`

1.3.2 IP Tunneling

In order to reach the IPv6 Internet, an isolated host or network must use the existing IPv4 infrastructure to carry IPv6 packets. This is done using a technique known as tunneling which consists of encapsulating IPv6 packets within IPv4, in effect using IPv4 as a link layer for IPv6.

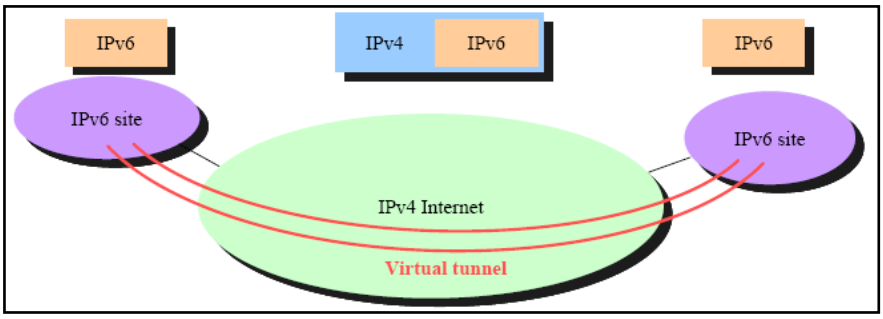

Fig. 1.12 IPv6 Tunneling

There are different tunnel broker providers, which offer a free service of network tunneling. The provider selected in this project is Hurricane Electric (http://www.tunnelbroker.net/), which offers worldwide coverage with servers distributed along different countries. Registration is required to use its services and the configuration is made through the website.

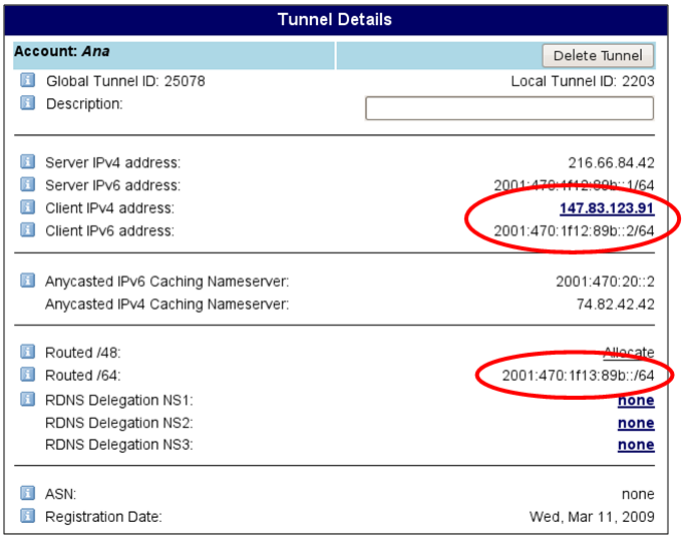

Fig. 1.13 IPv6 tunnel details

1.3.3 Features and differences from IPv4

IPv6 specifies a new packet format, designed to minimize packet header processing. Since the headers of IPv4 packets and IPv6 packets are significantly different, the two protocols are not interoperable.

IPv6 hosts can configure themselves automatically when connected to a routed IPv6 network using ICMPv6 router discovery messages. When first connected

to a network, a host sends a link-local multicast router solicitation request for its configuration parameters; if configured suitably, routers respond to such a request with a router advertisement packet that contains network-layer configuration parameters.

A number of simplifications have been made to the packet header, and the process of packet forwarding has been simplified, in order to make packet processing by routers simpler and more efficient:

- The packet header in IPv6 is simpler than that used in IPv4, although the addresses in IPv6 are four times larger, the (option-less) IPv6 header is only twice the size of the (option-less) IPv4 header.
- IPv6 hosts are required to perform end-to-end fragmentation, and not the IPv6 routers.
- The IPv6 header is not protected by a checksum; integrity protection is assumed to be assured by both a transport layer checksum and a higher layer (TCP, UDP, etc.) checksum.
- The Time-to-Live (TTL) field of IPv4 has been renamed to Hop Limit, reflecting the fact that routers are no longer expected to compute the time a packet has spent in a queue.

The next figure shows the structure of an IPv4 header:

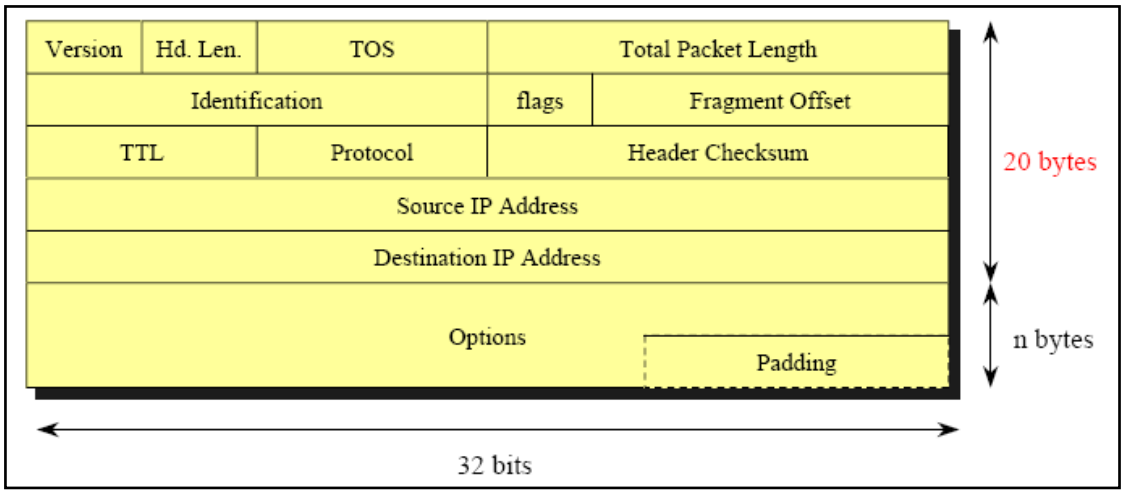

Fig. 1.14 IPv4 header format

The header packet is formed by 20 bytes with no options. IPv6 header is composed by 40 bytes as shown in the next figure.

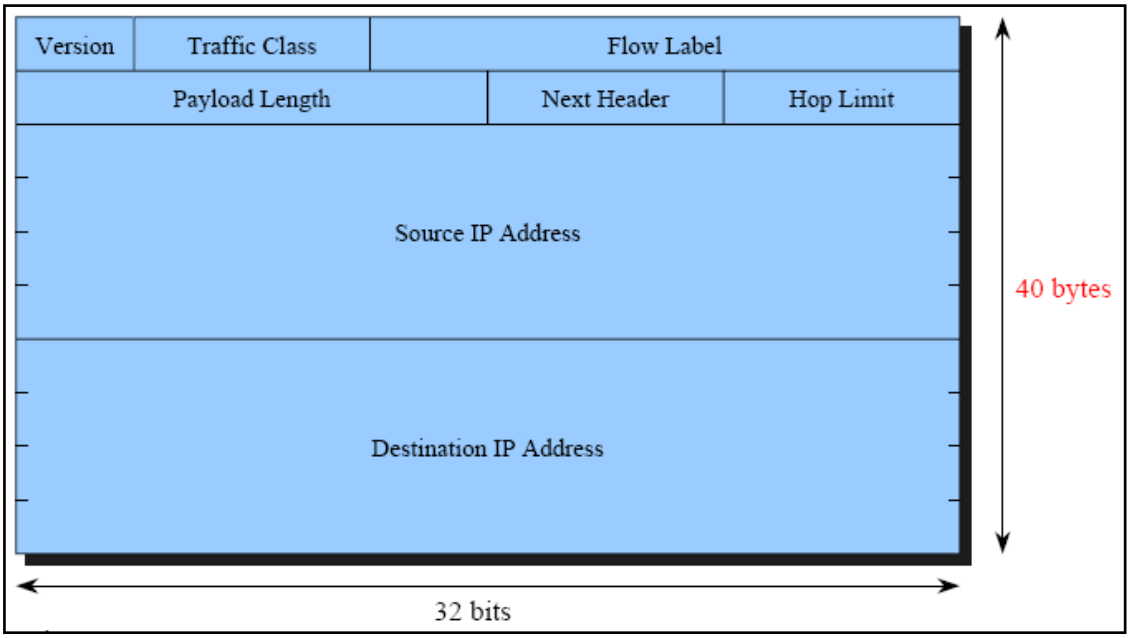

Fig. 1.15 IPv6 header format

CHAPTER 2. ADAPTATION LAYER

This chapter describes the frame format for transmission of IPv6 packets and the method of forming IPv6 link-local addresses on IEEE 802.15.4 networks. Since IPv6 requires support of packet sizes much larger than the IEEE 802.15.4 frame size, an adaptation layer is required.

2.1 LoWPANs addressing

IEEE 802.15.4 defines four types of frames: beacon frames, MAC command frames, ACK frames and data frames; and establishes several addressing modes: IEEE 64-bit extended addresses and 16-bit short addresses.

IPv6 packets must be carried on data frames and, for use within 6LoWPANs, both addresses are supported.

16-bit short addresses, basically an identification number within the PAN, are normally doled out by the PAN coordinator during an association event, and their validity and uniqueness is limited by the lifetime of that association.
64-bit addresses, known as the IEEE address or 802.15.4 interface identifier, may be formed from a short address by padding a 16-bit address out to 64 bits with zeros.

In IPv6 exists multicast (one-to-many) frames, not broadcast. But, multicast is not supported natively in IEEE 802.15.4, hence, IPv6 multicast packets must be carried as link-layer broadcast frames in IEEE 802.15.4 networks.

2.1.1 IPv6 Link local address

The IPv6 link-local address for an IEE 802.15.4 interface is formed by appending the interface identifier to the prefix FE80::/64.

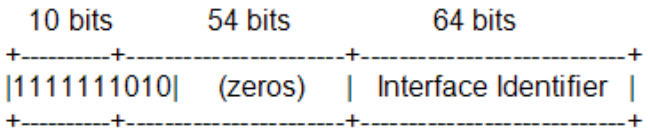

Fig. 2.1 IPv6 link local address

2.1.1.1 Example of IPv6 address formation

As shown in fig. 1.13, the routed address is:

2001:470:1f12:89b::/64

If an interface identifier (IEEE 802.15.4 short address) is given with value ‘20’ (which is 14 in hexadecimal format) for a determined device, the IPv6 address of that device will be formed as follows:

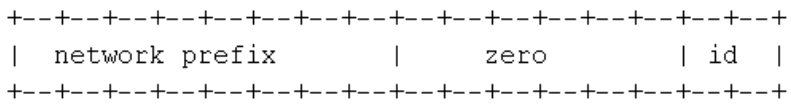

Fig. 2.2 IPv6 device address

- Network prefix (8 firsts octets) → 2001:0470:1f12:089b
- Zero (octets 9 – 14) → 2001:0470:1f12:089b:0000:0000:0000
- ID (octets 15 and 16) → 2001:0470:1f12:089b: 0000:0000:0000:0014

Finally, the address can be written as 2001:470:1f12:89b::14

2.2 Packets size

The maximum transmission unit (MTU) size for IPv6 packets over IEEE 802.15.4 is 1280 octets, hence, a full IPv6 packet does not normally fit in an IEEE 802.15.4 frame.
Considering the maximum physical layer packet size of 127 octets and a maximum overhead of 25 bytes:

127 bytes packet size – 25 bytes overhead = 102 bytes frame size,

Link-layer imposes more overhead, 21 octets in the worst case, which leaves only 81 octets available. Furthermore, IPv6 header is 40 bytes long:

81 octets packet size – 40 bytes IPv6 overhead = 41 bytes,

Those 41 octets are available for upper-layer protocols like UDP and TCP. UDP protocol uses 8 octets in the header, which leaves 33 octets for application data in the worst case. This points out the need for a fragmentation and reassembly layer. The use of this adaptation layer will increase the number of overhead octets.

The above observations lead to some considerations:

- The adaptation layer must comply with the IPv6 requirements of a minimum MTU. However, it is expected that small application payloads

with the proper header compression will produce packets that fit within a single IEEE 802.15.4 frame

- The above space calculation shows the worst case scenario, and points out the fact that header compression is almost unavoidable, but not all the applications of IPv6 over IEEE 802.15.4 will make use of it. It will be the case of a small application payload (about 33 octets)

2.3 LoWPAN encapsulation

The encapsulated frames are the payload in the IEEE 802.15.4 MAC protocol data unit.

All the LoWPAN encapsulated datagrams are prefixed by encapsulation header stack. Each header in the header stack contains a header type and header field. When more than one LoWPAN header is used in the same packet, there is an order established:

1) Mesh addressing header (M)
2) Broadcast/multicast header (B)
3) Header compression (HC1)

And after the headers, it is the payload:

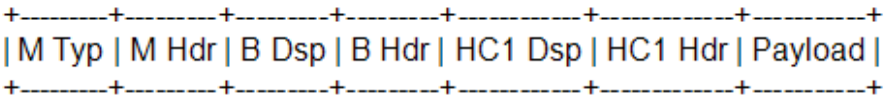

Fig. 2.3 Encapsulated IPv6 datagram

2.3.1 Dispatch value

In order to distinguish the different possible headers, the first byte in the header is the Dispatch. It can assume the values reported in the following table and it is divided in dispatch type (first two bits) and dispatch header (last six bits).

Table 2.1 Dispatch value bit pattern

Pattern		Header type
00	xxxxxx	NALP – Not a LoWPAN frame
01	000001	Uncompressed IPv6 address
01	000010	LoWPAN_HC1 compressed IPv6
01	010000	LoWPAN_B0 broadcast
01	111111	Additional dispatch byte follows
10	xxxxxx	Mesh header
11	000xxx	First fragmentation header
11	100xxx	Subsequent fragmentation header

The meaning of the dispatch value is the definition of the headers that follow and their ordering constraints relative to all other headers.

2.3.2 IPv6 Header compression

6LoWPAN allows a very heavy compression of the IPv6 header: instead of the original 40 bytes and IPv6 header may be in some cases represented with only 2 bytes. IPv6 header compression is possible by eliminating all those fields that can be retrieved somewhere else or that are fixed. The fields that cannot be compressed will be carried in-line after the HC1 header. There is just one field that must be fully carried, the Hop Limit (8 bits long).

For example, the version field can be omitted since it is fixed, and both IPv6 source and destination addresses could be link local, so the identifiers can be inferred from the layer two addresses; the same happens with frame length.

If the application does not need to know traffic class and flow label, they will be assumed to have fixed value zero. The following headers can be only UDP, ICMP or TCP.

If any field needs to be carried in-line, the corresponding bits will be set accordingly, and the fields will follow the HC1 header in the order:

- IPv6 Source address prefix and/or interface identifier
- IPv6 Destination address prefix and/or interface identifier
- Traffic Class and Flow Label (value zero)
- Next header

A common packet on 6LoWPAN network can follow this format by using the defined dispatch value of LOWPAN_HC1 followed by a LOWPAN_HC1 header ‘HC1 encoding’ field of 8 bits.

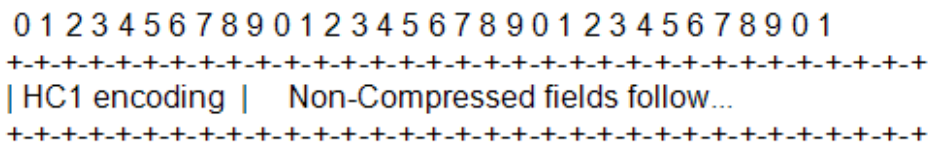

Fig. 2.4 Compressed header encoding

The firsts 4 bits indicate the IPv6 source and destination addresses. Bit 5 is for traffic class and flow label. Bits 5 and 6 show the next header (UDP, ICMP or TCP). The last bit indicates if more header compression for the next header is present.

2.3.3 UDP Header compression

The UDP header can also be compressed. The HC_UDP encoding allows compressing the source and destination port, and length, but the checksum must be carried in full. The length field can be deduced from information available elsewhere.

UDP header is 8 bytes length, but using compression can be reduced up to 4 octets.

2.4 Fragmentation

If an entire payload datagram fits within a single 802.15.4 frame, it is unfragmented and the LoWPAN encapsulation should not contain a fragmentation header. If the datagram does not fit within a single IEEE 802.15.4, it shall be broken into link fragments.
As the fragment offset can only express multiples of eight bytes, all link fragments except the last one must be multiples of eight bytes in length.

The first link fragment header should follow the structure shown in the next figure:

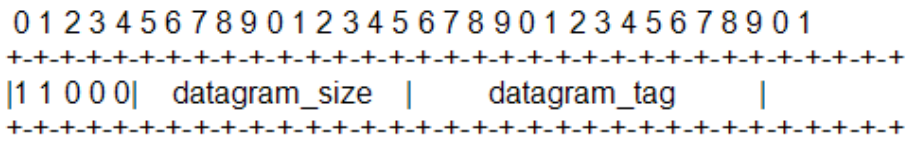

Fig. 2.5 First fragment

The second and subsequent link fragments should conform the next format:

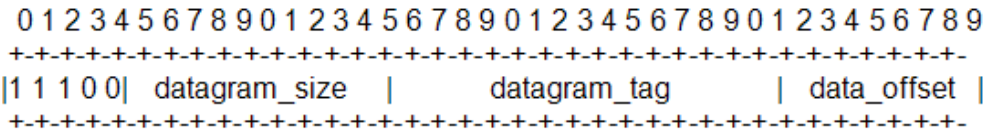

Fig. 2.6 Subsequent fragment

- Datagram_size (11 bits): size of the entire IP packet before link-layer fragmentation. This value must be the same for all fragments.
- Datagram_tag (16 bits): the sender should increment the datagram tag for successive fragmented datagrams. This value must be the same for all fragments of a payload.
- Datagram_offset (8 bits): only in the second and subsequent fragments and in increments on 8 octets.

The next figure is a graphical view of the different headers that can have a UDP packet sent over a 6LoWPAN network.

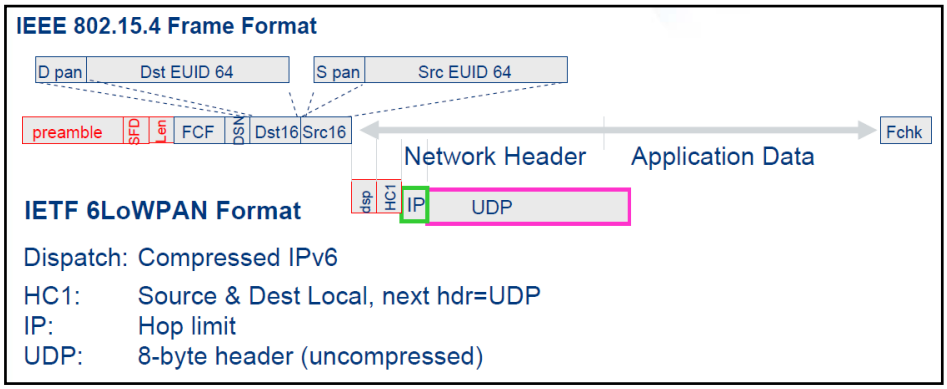

Fig. 2.7 6LoWPAN - UDP compressed

CHAPTER 3. ELEMENTS OF THE NETWORK

This chapter is an introduction to the application designed for this project. In the first place, it is necessary to define the software and hardware necessities, and most of the software packets need to be installed and configured properly.
TinyOS (http://tinyos.net) is a real time operative system used in WSN. TinyOS 6lowPAN group have implemented a 6lowpan/IPv6 stack for TinyOS 2.0.
The last section presents the hardware elements: the motes, the different products available at the market and their manufacturers.

3.1 Working environment

In order to develop applications with 6LoWPAN, one of the main elements is an appropriate working environment that will support software and hardware requirements.

3.1.1 Operative system

The applications of WSN use a specific operative system: TinyOS; and this OS can be installed in different environments like Windows XP and Vista, or Ubuntu, but considering the aim of the project, Ubuntu is the proper one.

3.1.1.1 TinyOS

TinyOS is a free and open source real time operative system component based for working with wireless sensor networks. The applications use NesC language as a set of cooperating tasks and processes.

TinyOS is a collection of software modules that can be glued together to build applications. NesC defines modules and interfaces, and create the configurations which are hierarchies of glued together modules. TinyOS is also a FIFO scheduler:

- Interrupts are handled immediately
- Background tasks are scheduled, that is put on a queue and executed when there is no other task being executed

This is going to be the operative system required to work with the devices of the network. Two versions have been developed: TinyOS 1.0 and 2.0. The last has been improved with later subversions, and the last one released is TinyOS 2.1.0 (also named as TinyOS 2.x). This entire project is referred to this latest version.

TinyOS is officially supported in Windows and Linux, but can work also under MacOS. At the official TinyOS webpage (http://www.tinyos.net/) the download and setup instructions are available for both operative systems.

3.1.1.2 Ubuntu

Even though TinyOS is able to work under Windows, the selected operative system is Linux, specifically Ubuntu 8.04 or 8.10.

Table 3.1 Ubuntu versions

	Release	Code names	Supported until
ubuntu	Ubuntu 8.04	Hardy Heron	April 2011
	Ubuntu 8.10	Intrepid Ibex	April 2010

Ubuntu is a free Linux distribution. Its goals include providing a stable operating system for the average user, with a strong focus on usability and ease of installation. Ubuntu is composed of multiple software packages distributed under either free software or an open source license. The main license used is the GNU General Public License (GNU GPL) which explicitly declares that users are free to run, copy, distribute, study, change, develop and improve the software. Different Ubuntu-like operative systems have developed: Kubuntu and Xubuntu.

Kubuntu uses the KDE graphical environment instead of GNOME, which can be considered more similar to graphical environment like MacOS or Windows.

Xubuntu is a more compact version of Ubuntu with the XFCE window manager. Because the XFCE desktop environment uses fewer system resources than the default GNOME, Xubuntu is often used on older computers, systems with limited resources.

XubunTOS is a variety of Xubuntu which goal is to simplify the installation of TinyOS by using a Linux Live CD. XubunTOS is built from Xubuntu and TinyOS 2.x.

Fig. 3.1 XubunTOS

Last release was launched on April with Xubuntu 8.04 and TinyOS 2.1. from CVS (a free software of revision control system).

For this project, as mentioned above, Ubuntu was the operative system required. TinyOS is completely installed by the user and the instructions are detailed in the next section.

3.1.2 Installing TinyOS on Ubuntu 8.10

Once Ubuntu has been correctly installed, it is necessary to install different packages and tools. It is recommended to install the entire packages through *Synaptic Package Manager*, a graphical user interface to easily install, remove, configure or upgrade software packages based on the apt-get command line tool (APT). But it is also possible to install them through the shell by using the *apt-get* commands.

By default, most of the packages are included in the repositories, but in the case of TinyOS it is necessary to add a new repository in order to install the package through Synaptic or commands.

Select 'add third party repositories' at the configuration menu of *Synaptic Package Manager* and include the Debian Repository provided by Stanford University. For more detailed information of how to install TinyOS step by step see Annex B.

It is strongly recommended to execute all the commands as super user during the installation:

> *sudo su* + password

The super user (administrator) password is established at the moment of installing Ubuntu, make sure knowing it before trying to install any package.

TinyOS package comes with Java 1.5 version and Graphviz, but does not include Python. Graphviz is a package of open source tools for drawing graphs based on scripts. And Python is a general-purpose high-level programming language; its purpose will be explained in following sections.
It is necessary to install Python manually with the corresponding commands.

Once all packages for TinyOS have installed, it is necessary to edit the script containing all the system paths. By default, TinyOS will be installed in the route:

> */opt/tinyos-2.1.0* or */opt/tinyos-2.x*

Where the default script is also placed: *tinyos.sh*. Edit the script and change the paths if it is necessary. This file has to be also placed at the */etc/profile.d* directory in order to be executed each time the computer starts.

```
#! /usr/bin/env bash
# Here we setup the environment
# variables needed by the tinyos make system

echo "Setting up for TinyOS-2.x"

TOSROOT=/opt/tinyos-2.x
TOSDIR=$TOSROOT/tos
CLASSPATH=$CLASSPATH:$TOSROOT/support/sdk/java/tinyos.jar:.
MAKERULES=$TOSROOT/support/make/Makerules
PYTHONPATH=:$TOSROOT/support/sdk/python

export TOSROOT
export TOSDIR
export CLASSPATH
export MAKERULES
export PYTHONPATH
```

Optionally, *bashr* in the */etc* directory can also be modified. At the end of the *bash.bahsrc* file add the next lines:

```
# tinyos.sh
if [ -f /opt/tinyos-2.x/tinyos.sh ] ; then
      . /opt/tinyos-2.x/tinyos.sh
fi
```

By doing this, each time that a shell is started, the system executes the *tinyos.sh* file. Thus is really profitable in case of modifying the file many times. If it is not done, the changes will only be effective after restarting the computer.

The last step to check the correct installation of TinyOS in Ubuntu is to execute the following command:

> *tos-check-env*

Different messages will appear showing the paths for the different tools that TinyOS requires. Probably a warning message will appear at the end of the shell informing that the Graphviz version has to be updated to 1.10. But it is not a problem because, in fact, the installed version of Graphviz is newest than the one warned by the *tos-check-env* command.

3.1.3 Other necessary packages

Not only TinyOS, Java and Python are necessary to work with 6LoWPAN networks. Other basic packages are required; all of them can be installed by using the *Synaptic Package Manager.* The list of those packages is:

- cvs: is a version control system, helps to manage releases.
- subversion: an open source version control system, the substitute of cvs
- build-essentials: required for building Debian packages
- stow: helps the system administrator to organise files

- automake: tool for automatically generate Makefile.in's from make definitions
- autoconf: useful to write programs
- libtool: generic library
- autobook (optional): digital book where the use of automake, autoconf and libtool is explained.

It is possible that one or more of them have been already installed, and it is also probable that at the moment of installing one of them, other packages related will also be installed.

3.1.4 First steps with TinyOS

All the packages have installed and the TinyOS environment is working correctly. With the installation packages some example codes are included. Those examples are a guide to get used to TinyOS.
It is strongly recommended to start using TinyOS following the tutorials that can be found at the website (http://docs.tinyos.net/index.php/TinyOS_Tutorials).

At those tutorials it is explained how the codes are executed, the relation between different files and how does NesC (the programming language used to build applications in TinyOS) works.
Basic programs can be built in the devices, which allow the user to check their correct start up.

These tutorials also explained the main commands that can be used through the shell.

3.1.4.1 Main commands

The main commands that are used to build the programs into the devices are:

- *motelist*: list which USB ports have devices attached
- *make 'platform'*: with this command the program is compiled but not downloaded over the devices, it just create a TOS image. It is used to check possible errors before installing on a mote

  > make telosb;

- *make 'platform' install,ID bsl,/dev/ttyUSBx*: compile and download the program over the device attached into USBx port. It will take a few seconds and after that, the device will start executing the code. It can also be done with *reinstall* instead of *install* if the same code has to be installed in more than one device because it just downloads it over the node.

> make telosb install, 2 bsl/dev/ttyUSB0;

With those basic three commands any application can be installed into a device. For other commands see [5].

3.1.5 Programming languages

Two different programming languages have already been installed: NesC and Python.

3.1.5.1 NesC

NesC is a component-based, event-driven programming language used to build applications for the TinyOS platform. It is built as an extension to the C programming language with components "wired" together to run applications on TinyOS.

The basic concepts behind NesC are:

- Programs are built out of components, which are assembled (“wired”) to form whole programs. Components define two scopes, one for their specification (containing the names of their interface instances) and one for their implementation.
- Interfaces may be provided or used by the component. The provided interfaces are intended to represent the functionality that the component provides to its user. The interfaces used represent the functionality the component needs to perform its job.
- Interfaces are bidirectional: they specify a set of functions to be implemented by the interface’s provider (commands) and a set to be implemented by the interface’s user (events). This allows a single interface to represent a complex interaction between components. This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signalled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the sendDone event.

TinyOS and NesC have a documentation feature called nesdoc, which generates documentation automatically from source code. In addition to comments, nesdoc displays the structure and composition of configurations.

To generate documentation for an application, type:

> make *platform* docs

Where *platform* is referred to the hardware in use. Then go to *opt/tinyos-2.x/doc/nesdoc*, there will be a directory for the platform: open its index.html. A list of the components and interfaces will appear.

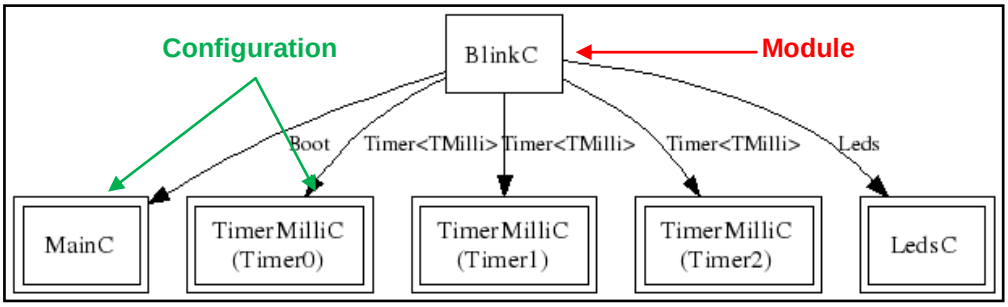

Fig. 3.2 Example of components graph

In nesdoc diagrams, a single box is a module and a double box is a configuration and lines denote wirings.

To build an application some files are required:

- The configuration and implementation file: where the interfaces are wired. This file has a *.nc* extension
- The module file: where the main code is written. This file also has a *.nc* extension, but used the file name used to contain a capital letter to distinguish between both files
- The *.h* file contains the definition of constants and structures
- The last file required is Makefile, which contains the compilation rules.

3.1.5.2 Python

Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, and comes with extensive standard libraries.

Python is known for its ease of quickly programming new applications, and will be used as a bridge to communicate between the base station (PC) and the nodes of the 6LoWPAN network. How this application is implemented will be explained in the following chapter (see: *4.2.1.3 Receiving the data*)

3.2 Hardware architecture

The main hardware elements of the 6LoWPAN sensor network are the devices or nodes, also known as motes. As it was mentioned in the first chapter, the

typical architecture of a mote is formed by a power unit, a processing and communicating units and a series of sensors.

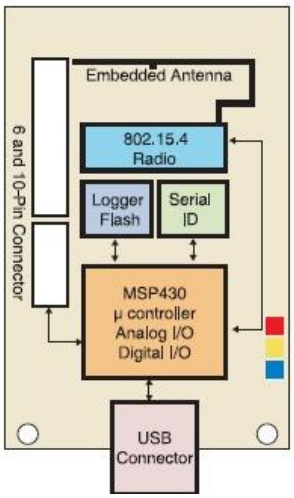

Fig. 3.3 Sensor architecture

3.2.1 TinyOS platforms

The processing and communicating units included into the motes define a series of platforms; and TinyOS supports different platforms. The next table show the main features of the most common platforms.

Table 3.2 TinyOS platforms

Platform		Imote2	MicaZ	Telosb	EPIC
Manufacturer		Crossbow	Crossbow	Crossbow	Arch rock
Processor		Intel PXA271 XScale	Atmel ATMega 128L	TI MSP430F1611	TI MSP430F1611
	Clock	13 to 416 MHz	8 MHz	8 MHz	8 MHz
	Bits	32	8	16	16
	FLASH	512 kB	128 kB	48 kB	48 kB
	RAM	256 kB	4 kB	10 kB	10 kB
	Max.Consumption	44 mA	19 mA	1.8 mA	1.8 mA
Radio unit		Chipcon CC2420	Chipcon CC2420	Chipcon CC2420	Chipcon CC2420
	Standard	CSMA/CA	CSMA/CA	CSMA/CA	CSMA/CA
	Band/Data rate	2.4 GHz / 250 kbps	2.4 GHz / 250 kbps	2.4 GHz / 250 kbps	2.4 GHz / 250 kbps
	Max.Consumption	23 mA	23 mA	23 mA	23 mA
External memory		32 MB	512 kB	1024 kB	16 MB
Onboard Sensors		-	Light, RH, temperature, pressure, acceleration	Light (TSR, PAR), temperature, humidity	Light, temperature, humidity

The onboard sensors can be expanded in some platforms, depending on the external I/O pins available.

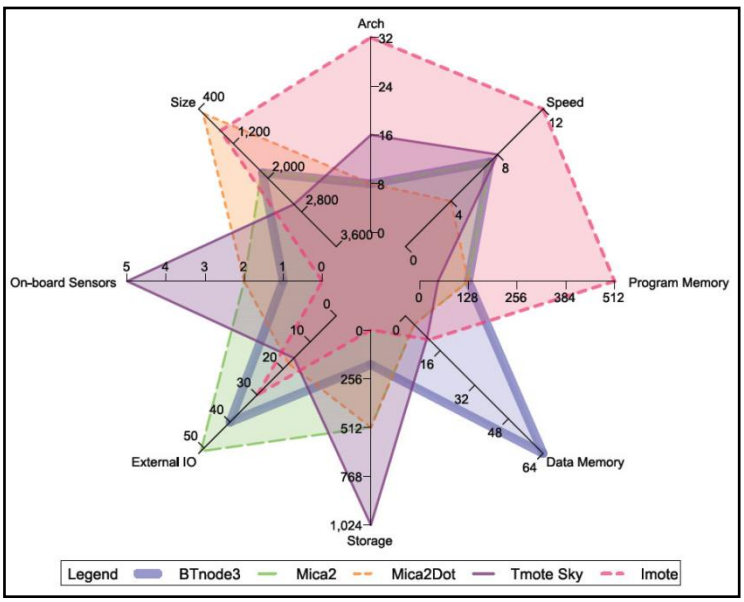

Fig. 3.4 Platform comparison

Table 3.2 and figure 3.4 show some of the existing TinyOS platforms. The figure 3.4 is an excellent tool to compare all the features of the different platforms and easily choose the most appealing for the final application.

3.2.1.1 TinyOS platforms compliant 6LoWPAN

In this section four platforms will be presented, all of them provide 6LoWPAN motes that support 6LoWPAN stack for TinyOS-2.x and other RTOS (Real-Time Operating System).

Development kits are offered including the motes and their own application software to work with 6LoWPAN networks.

In table 3.2, a series of TinyOS platforms have been described, but between them and others platforms from fig. 3.4, only Telosb is able to work using the TinyOS 6LoWPAN stack.

Telosb motes are configured and managed through TinyOS by using specific libraries and applications. But, it is not the only option to work with those networks; some manufacturers have developed their own application software, for example, Arch rock has a visual application to easily manage the network.

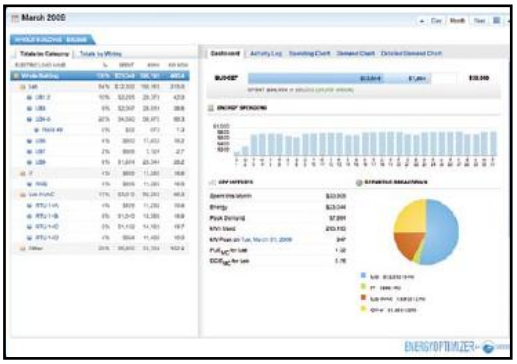

Fig. 3.5 Arch rock visual tool

The following table summarizes the development kits offered by those manufacturers and Telosb platform.

Table 3.3 6LoWPAN products

	Telosb	Jennic	Arch rock	Sensinode
Package	Telosb motes	JN5139-EK000	PhyNet OEM Development Kit – IE	K210 NanoDevkit
Interface	TinyOS	C API	C API and TinyOS	NodeView tool and RTOS Nano
Onboard sensors	Light (TSR,PAR), temperature, humidity	Light, temperature, humidity	Light, temperature, humidity	Light, temperature
Processor	TI MSP430F1611	JN5139 single chip	TI MSP430 F1611	RC2301AT module
Clock	16 MHz	8 MHz	8 MHz	32 MHz
Bits	16	32	16	16
Flash	48 kB	192 kB (ROM)	48 kB	128 kB
RAM	10 kB	96 kB	10 kB	8 kB
Transceiver	CC2420	JN5139 single chip	CC2420	CC2431
Power	Tx 0 dBm* Rx -90 dBm	Tx 3 dBm Rx -97 dBm	Tx 0 dBm Rx -90 dBm	Tx 0 dBm Rx -92 dBm

* Telosb transmission power can be modified by software to save energy, but reducing the power also reduces the range.

For more information about those products, see table in Annex C.

3.2.2 Telosb mote

In order to develop the 6LoWPAN network, the used device is Telosb or Tmote Sky, both platforms are equivalent. Even the hardware was manufactured by Tmote Sky (now Sentilla), the platform specified to work in TinyOS is Telosb.

Tmote platform was developed by Berkeley working group, and its license was given to different manufacturers, the most important are Crossbow, Moteiv and Arch rock.

Those devices allow the use of TinyOS without using an upper-layer application. This is, the user can configure the devices and the networks right from the beginning, and configure the network in a proper manner depending on the final application.

The main features of this device have been shown in table 3.2, but other interesting characteristics of this platform appear in table 3.4.

Table 3.4 Telosb features

Size	65 x 31 x 6 mm			
Operating free air temperature	-40 to 85 ºC			
Antenna	Integrated onboard			
Range	50 m indoor, 125 m outdoor			
I/O	16-pin expansion connector			
External oscillator	32 KHz			
Power	Two AA batteries / USB port			
Supply voltage	2.1 to 3.6 V / 3 V			
Current consumption	**MCU**	**Radio**	**Nom**	**Max**
	On	RX	21.8 mA	23 mA
	On	TX	19.5 mA	21 mA
	On	Off	1800 µA	2400 µA
	Standby	Off	5.1 µA	21 µA

Fig. 3.6 Telosb module

Telosb platform includes a series of sensors: humidity and temperature sensor, and light sensors.

3.2.2.1 Temperature and humidity sensor

This sensor, SHT10, is manufactured by Sensirion AG. A unique capacitive sensor element is used for measuring relative humidity, while temperature is measured by a band-gap sensor. Both sensors are coupled to a 14bit A/D converter and a serial interface circuit, and provide a fully calibrated digital output.

Three models available: SHT10, SHT11 and SHT15 (SHT1x series). The difference among them is that the SHT15 produces higher accuracy readings, and SHT11 has higher accuracy than SHT10. The motes used during this project have SHT10 model.

Table 3.5 SHT10

Parameter	Condition	Value
Humidity resolution		0.05 %RH
Humidity accuracy	typical	±4.5 %RH
Humidity response time	typical	8 s
Temperature resolution		0.01 ºC
Temperature accuracy	typical	±0.5 ºC
Temperature response time	max	30 s
Current consumption	measuring	1 mA
	average	28 µA
	Sleep	1.5 µA

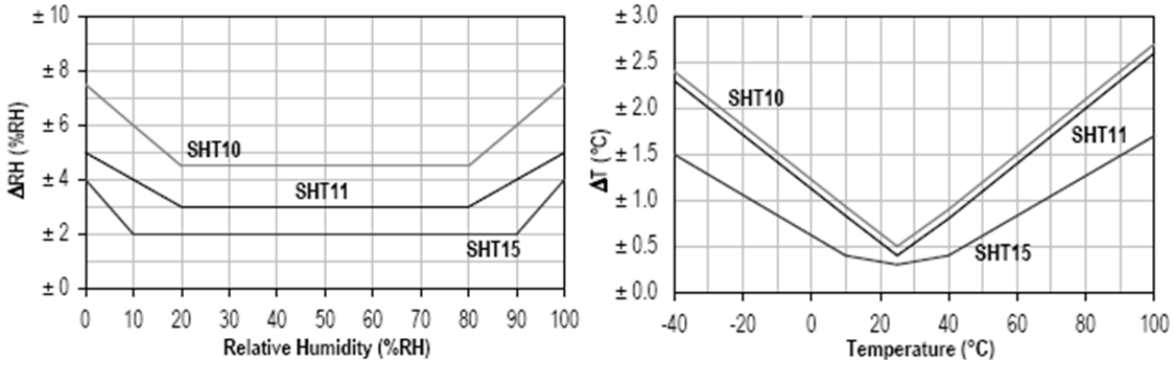

Fig. 3.7 SHT10 resolution

For compensating non-linearity of the humidity sensor it is recommended to convert the humidity readout (SO_{RH}) with the following formula:

$$RH_{linear} = c_1 + c_2 \times SO_{RH} + c_3 \times SO_{RH}^2 \ (\%RH) \qquad \textbf{(3.1)}$$

The values of coefficients are given in the following figure:

SO_{RH}	c_1	c_2	c_3
12 bit	-2.0468	0.0367	-1.5955E-6

Fig. 3.8 SHT1x humidity coefficients

For example, the SO_{RH} is 1481, then:

$$RH\ (\%) = -2.0468 + 0.0367 \times 1481 - 1.5955 \cdot 10^{-6} \times 1481^2 = 48.8\ \%RH$$

In case of temperature, the following formula is used to convert digital readout (SO_T) to temperature value:

$$T = d_1 + d_2 \times SO_T \quad \textbf{(3.2)}$$

And the coefficients are $d_1 = -39.6$ and $d_2 = 0.01$ (ºC).

For example, the SO_T is 6725, then:

$$T\ (ºC) = -39.6 + 0.01 \times 6725 = 27.65\ ºC$$

3.2.2.2 Light sensors

Telosb has connections for two light photodiodes, S1087 for sensing photosynthetically active radiation (PAR) and the S1087-01 for sensing the entire visible spectrum (TSR). Both photodiodes are manufactured by Hamamatsu.

The PAR sensor measures radiation between 400 and 700 nm, and its main function is to measure light that promotes photosynthesis in the plant cell.

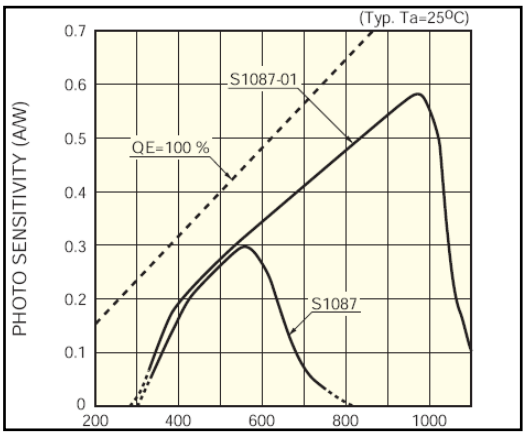

Fig. 3.9 Hamamatsu S1087 and S1087-01

Based on the figure above (available in the Hamamatsu S1087 datasheet [15]) the current of the sensor, I, may be converted to lux using the following formulas:

$$S1087 \rightarrow lx = 0.769 \times 10^5 \times I \times 1000 \quad \textbf{(3.3)}$$

$$S1087 - 01 \rightarrow lx = 0.625 \times 10^6 \times I \times 1000 \quad \textbf{(3.4)}$$

Lux (lx) is the IS unit to of luminance and luminous emittance. The first equation is referred to PAR, and the second to the TSR.

For example, the output of the A/D converter for TSR sensor is 203:

$$I = \frac{V_{sensor}}{100000\,\Omega};\ V_{sensor} = AD_{output} \times \frac{Vref}{2^{12}} \quad \textbf{(3.5)}$$

I = (203 x Vref /4096) / 100000 = 743.4 nA

The resistance at the output has a 100 kΩ value. Then in this case it is necessary to apply 3.4:

lx = 0.625x10^6 x I x 1000 = 464.63 lx

For more information about the measurements see Annex G.

3.3 Environmental impact

The environmental impact of the network is an important topic that should be considered.

The establishment of a network should have as less impact as possible: the placement of the devices should not modify the environment, and the components should not contaminate it.

The AA batteries that provide the power to the motes are rechargeable, that fact reduces the use of a large amount of batteries non rechargeable which have to be recycled after their use in order to avoid contamination. The components in which those rechargeable batteries are based are NiMH. NiMH batteries are commonly considered to have lower environmental impact than NiCd batteries (the previously technology used), due to absence of toxic cadmium.

The components of the motes are also environmentally friendly; all of them comply with RoHS regulations, including the sensors, which are also WEEE compliant. SHT1x is free of lead (Pb), cadmium (Cd), mercury (Hg) and chrome (Cr).

CHAPTER 4. 6LOWPAN APPLICATION

In this section will be presented the application developed to establish a 6LoWPAN network. The working environment and the devices have been already presented, and during this chapter the code and its compilation will be detailed, and so will be how the motes communicate with the base to send all the information collected.

4.1 Berkeley WEBS application

Berkeley WEBS (Wireless Embedded Systems) is a research group whose research vision is focused on real-world wireless devices that communicate wirelessly to perform tasks such as sensing and actuation.

They have developed an implementation of IPv6 for TinyOS. It uses 6lowpan/HC-01 header compression and network programming support. It has been tested on telosb platforms. Standard tools like ping6, tracert6, and nc6 can be used to interact with and troubleshoot a network of blip devices, and pc-side code is written using the standard Berkeley sockets API. A sensor network can also be easily mapped into the public subnet to provide global connectivity.

4.1.1 Releases

Different releases have been published until today, and the next one will be published at the end of August 2009.

At the beginning of this project, the last release was named *lowpan*, and was available since 13th November 2008. The first tests were made with this application, but, in 20th March 2009 a newest version was published under the name of *blip*.

The improvements of the last version were mainly the following:

- No longer need separate radvd (advertisement daemon is included in the routing driver)
- Driver runs a telnet server for route inspection and maintenance
- UDP now accessed through a generic component

All those topics will be introduced along this chapter.

4.1.2 Installation

In order to install, build and run the blip application there is a guide included in the blip documentation. But first it is necessary to download it.

CVS (Concurrent Versions System) is a tool used by many software developers to manage changes within their source code tree. CVS provides the means to store not only the current version of a piece of source code, but a record of all changes that have occurred to that source code. This tool was included in the list of necessary packages of the previous chapter, and is the one used to download the code.

Open a shell and type:

> cvs -d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroot/tinyos login

> cvs -z3 -d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroot/tinyos co -P *modulename*

When prompted for a password for *anonymous*, simply press the Enter key. The module to check out must be specified as the modulename, in this case, *tinyos-2.x-contrib/berkeley.* By doing this, the complete Berkeley module will be downloaded. The last step is to copy the folder into the correct directory, that is:

> /opt/tinyos-2.1.0

where the TinyOS is installed.

The modules have been downloaded; the following steps are specified in the readme file from the blip documentation (consult the link at Xbibliografia). One of the most important steps is to correctly modify the TinyOS script to add the blip environmental variables:

```
LOWPAN_ROOT=$TOSROOT/berkeley/blip
TOSMAKE_PATH=$LOWPAN_ROOT/support/make
```

4.1.3 Running the application

The Berkeley module includes a series of codes that can be installed on the devices. Among them, the most interesting ones are IPBaseStation and UDPEcho. Both applications can be found at the *apps* folder.

Plug the mote into the USB port and type *motelist* to check if it is recognized. Change location to the apps directory and, then to application folder:

> cd $LOWPAN_ROOT/apps/IPBaseStation

> make telosb blip

With the second command the code is build but not downloaded into the mote. To download it, it is necessary to type the command previously mentioned (see 3.1.4.1):

> make telosb blip install bsl,/dev/ttyUSBx

In case of the BaseStation it is not necessary to specify an ID for the device. But in case of UDPEcho it is mandatory to specify a different ID for each device. That is because the mote ID will form the IPv6 address of the device.

Note that, at the moment of writing the command the ID is written in decimal format, but when it forms the IPv6, it is expressed in hexadecimal format:

ID = 65 → 2001:470:1f12:89b::41

Download IPBaseStation in one mote, and UDPEcho in the desired number of motes that will belong to the network.

4.1.3.1 Router Advertisement Daemon

Router ADVertisement Daemon (radvd) is a daemon (a program that runs as a background process doing some specific task), which can automatically assign and configure IPv6 addresses to computers on a LAN, or in this case, a PAN.

To install it, just open a shell and type:

> apt-get install radvd

Or use the *Synaptic Package Manager.*

Its configuration file is /etc/radvd.conf. Edit this file with the corresponding IPv6:

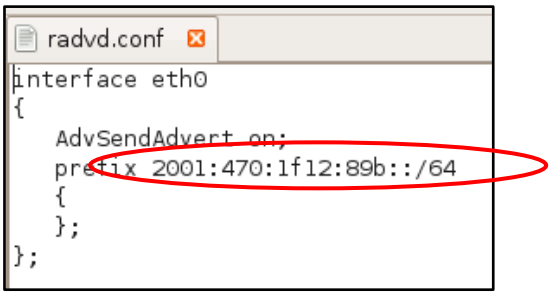

Fig. 4.1 Radvd file

In the previous release of blip application it was necessary to manually start and stop this daemon, but with blip, radvd is started through the driver.

4.1.3.2 IPv6 tunneling

As was mentioned in chapter 1, there are several tunnel broker providers that let the use of different tunnels to implement IPv6 over IPv4. In the Annex A it is explained how to create a tunnel in Hurricane Electric and how to get the information to implement by using a script.

4.1.3.3 Driver

The mote with the IPBaseStation code downloaded will act as the 802.15.4 interface between the LoWPAN network and the PC. In other words, its function is acting like a 'bridge' between serial and radio:

- Retransmitting the received packets from the motes to the PC (from radio to serial)
- Sending the information from the PC to the motes (from serial to radio)

On the radio link, it sends radioactive messages whose format depends on the network stack being, in this case, UDP.

On the serial link, it will filter the incoming radio messages that are not part of the network.

To start the driver, first of all it is necessary to build it. Move to the corresponding directory and built it as follows:

```
> cd $LOWPAN_ROOT/support/sdk/c/blip
> make
```

Edit the configuration file *serial_tun.conf* located at the same directory. Change the address that the router will advertise to the subnetwork and which network interface to proxy neighbor advertisements on.

In this case, the *serial_tun.conf* file will look as follows:

```
addr 2001:470:1f12:89b::2
proxy eth1
channel 21
```

Plug into a USB port the base station mote and start the driver with the command:

```
> sudo ./ip-driver /dev/ttyUSBx telosb
```

USBx has to be replaced with the corresponding port number, and *telosb* fixes the baudrate at 115200 bps.

Now a telnet service is running for a simple route inspection and maintenance.

```
root@isilab-desktop:/opt/tinyos-2.x/berkeley/blip/support/sdk/c/blip# ./ip-drive
r /dev/ttyUSB0 telosb
07/03/2009 11:59:19.856027: INFO: Read config from 'serial_tun.conf'
07/03/2009 11:59:19.856430: INFO: Proxying neighbor advertisements to eth1
07/03/2009 11:59:19.856628: INFO: Using channel 17
07/03/2009 11:59:19.856931: INFO: telnet console server running on port 6106
07/03/2009 11:59:19.860168: INFO: created tun device: tun0
07/03/2009 11:59:22.987345: INFO: interface device successfully initialized
07/03/2009 11:59:22.987427: INFO: starting radvd on device tun0
07/03/2009 11:59:32.668569: INFO: interface device successfully initialized
```

Fig. 4.2 IP-driver running

4.2 Configuring the network

With the downloaded Berkeley code, it is possible to create a simple network where a mote can send simple packets to the base station. But depending on the application area of the network, the user has to adapt the code to accomplish its aim.

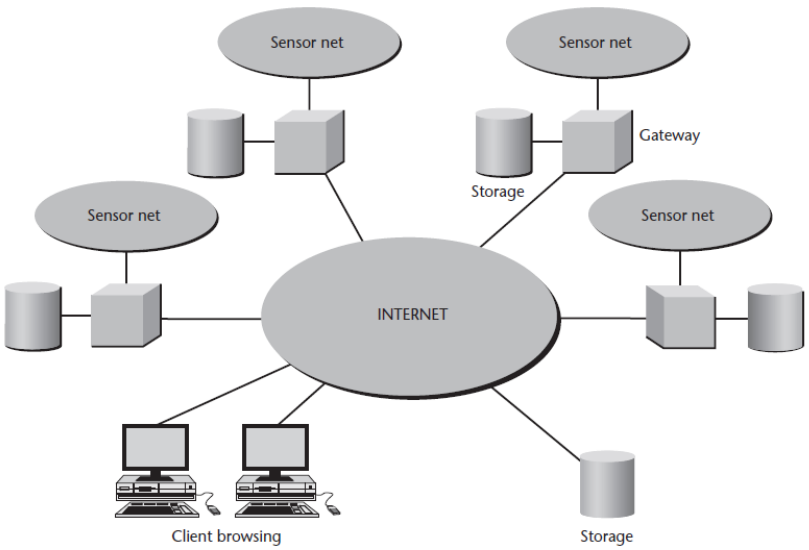

Fig. 4.3 Possible architecture of a 6LoWPAN network

4.2.1 Environmental application

The objective of the 6LoWPAN network is to provide a tool to supervise and control a determined environment, the EPSC campus. All the information required to accomplish this aim is collected trough the data sent by the motes.

By default, the UDPEcho code does not implement any sensor reading. It is necessary to modify it in order to allow the mote to read the data from the different sensors.

4.2.1.1 Implementation of the sensors

Each sensor has its own component, and the first step is to add the component into the configuration file *UDPEchoC.nc*:

```
components new HamamatsuS10871TsrC() as Sensor;
components new HamamatsuS1087ParC() as Sensor2;
components new SensirionSht11C() as Sensor3;
components new DemoSensorC() as Sensor4;
```

The *DemoSensorC()* component is the default sensor onboard, in this case, it is configured to be the sensor voltage reading.

Once the components have been added, it is necessary to wire the components and the interfaces:

```
UDPEchoP.ReadTSR -> Sensor.Read;
UDPEchoP.ReadPAR -> Sensor2.Read;
UDPEchoP.ReadExtTemp -> Sensor3.Temperature;
UDPEchoP.ReadHum -> Sensor3.Humidity;
UDPEchoP.ReadVolt -> Sensor4.Read;
```

In the module file, *UDPEchoP.nc*, those interfaces are called using an alias:

```
interface Read<uint16_t> as ReadTSR;
interface Read<uint16_t> as ReadPAR;
interface Read<uint16_t> as ReadExtTemp;
interface Read<uint16_t> as ReadHum;
interface Read<uint16_t> as ReadVolt;
```

For each interface it is required to implement all the events. The sensors have *Read* interface, which has an unique event, *readDone(),* that is, what is expected to do to signal the completion of the read.

```
event void ReadTSR.readDone(error_t result, uint16_t data)
{
  if (result == SUCCESS){
    stats.tsr = data;
    call ReadPAR.read();
  }
}
```

If the event has been successfully executed (`result == SUCCESS)`, the data are copied in a variable and the next sensor function is called:

```
call ReadPAR.read();
```

The sensors read in sequence and copy the data in different variables that form a structure.

All the data has to be sent in UDP packets, this implies that the file where the UDP structure is defined has to be also modified. This file is named *IPDispatch.h* and its directory is:

> $LOWPAN_ROOT/tos/lib/net/blip/IPDispatch.h

At the end of the file, an UDP structure is defined: *udp_statistics_t.*

```
typedef nx_struct {
  nx_uint16_t temp;
  nx_uint16_t hum;
  nx_uint16_t tsr;
  nx_uint16_t par;
  nx_uint16_t volt;
  nx_uint16_t sender;
} udp_statistics_t;
```

Each variable is an unsigned integer of 16 bits.

4.2.1.2 Establishment of the network

Once the motes have been built with their corresponding applications, that is, one mote working as a Base Station and, at least, one mote working as a UDPEcho and reading the data from the onboard sensors; they should be placed in an appropriate location considering the range specified in table 3.4.

Through the shell, start the driver with the commands explained in 4.1.3.3. If the Base Station mote is unplugged from the USB port, the driver with stop running and the so will the network.

When the remote nodes (UPDEcho motes) are switched on by putting the batteries into their box, they will automatically join to network by the neighbor discovery process. To observe how does it happen, Wireshark can be used.

Wireshark, originally named Ethereal, is a free packet sniffer computer application, used for network analysis. It has a graphical front-end, and many more information sorting and filtering options and allows the user to see all traffic being passed over the specific interface.

Start Wireshark on tun0 interface that has been created by starting the driver. The Base Station will start sending and receiving packets. The kind of packets than can be observed are:

- Router advertisement (RA): presence of the router in the network
- Router solicitation (RS): motes send request to the router to join the network or to indicate their presence
- ICMPv6 error (port unreachable): nobody is listening on the specified port
- UDP: containing the data from the mote

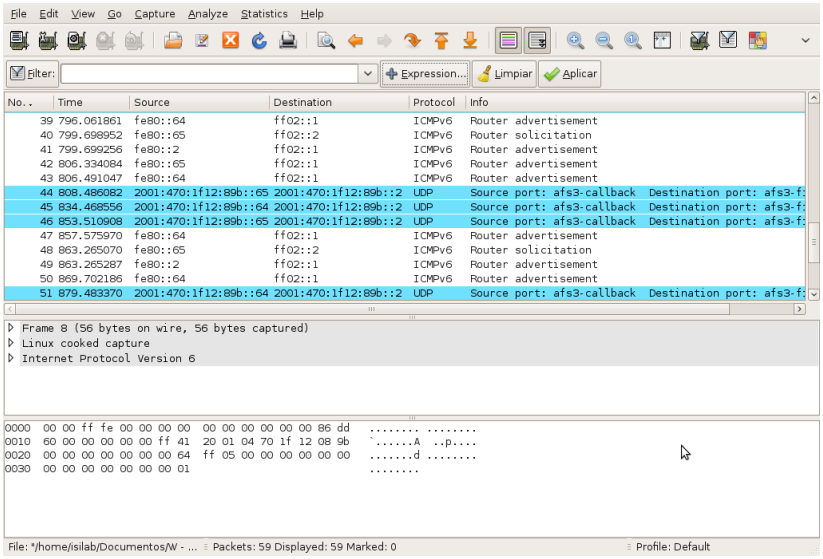

Fig. 4.4 Wireshark capture

4.2.1.3 Receiving the data

To receive the packets is necessary to open a socket at the corresponding port at which the information is sent.

Note that if the Base Station is not running the proper application to read the data an ICMPv6 packet will be sent advertising that the port was unreachable.

Berkeley application implements a socket with Python. The files are located at *util* folder from the UDPEcho application: *Listener.py* and *UdpReport.py*.

First, take a look at the UDPEcho code, where the port is selected:

```
#ifdef REPORT_DEST
    route_dest.sin6_port = hton16(7000);
    inet_pton6(REPORT_DEST, &route_dest.sin6_addr);
…
    call Status.bind(7001);
```

The UDPEcho mote sends all packets through the port 7001 to the Base Station port 7000, thus, the socket has to be opened at the receiving port:

```
port = 7000
s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
s.bind(('', port))
```

The code above is written in Python (*.py* file extension). For starting receiving packets, start the driver and then, execute the *Listener.py* by typing as superuser:

> ./Listener.py

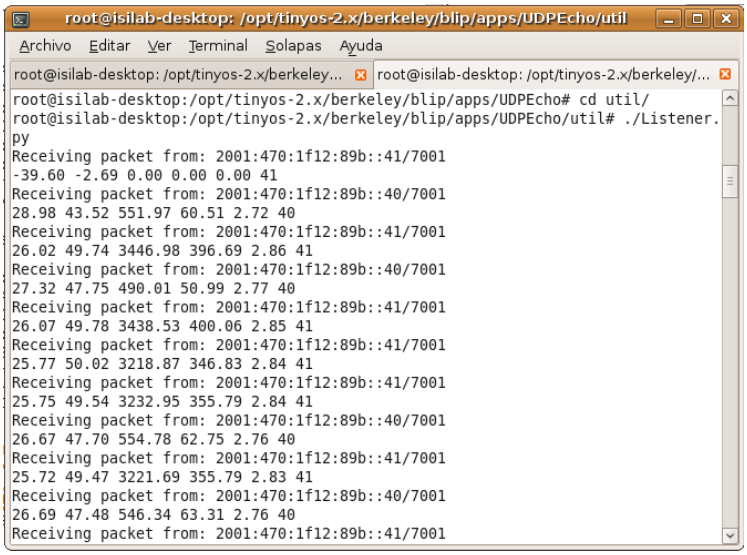

Fig. 4.5 Listener.py

Listener.py calls another Python file, *UdpReport.py*, which is responsible of creating an interface to the UdpReport message type. UdpReport's size is 32 bytes length.

Each field of the structure defined in the previous section, is read, and collected to form a message that will be displayed through the command shell where the Listener.py is running.

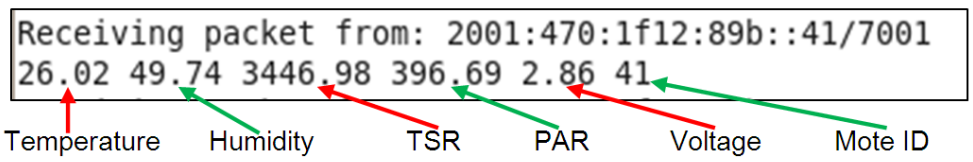

Fig. 4.6 UDP received data

4.3 Power management

The Base Station mote is always connected to USB port, no batteries are needed. But the UDPEcho motes need batteries or another power source such as a solar panel.

Between the features of any WSN are the low-power and energy efficiencies devices, that is why the consumption of the motes is an important topic.

The Berkeley WEBS application does not allow the implementation of the low power listening (LPL) interface to control the duty cycle, that is, to keep the mote with a very low consumption level while no data has to be sent.

Without this interface it is possible to reach a 50% duty cycle. The interface *SplitControl* allows starting and stopping the mote's radio, but requires a complete cycle to stop the radio and another to start it again. Using this method the batteries lifetime is almost doubled.

4.3.1 Measuring the consumption

To measure the current consumption of the devices it was measured the voltage of a resistor (nominal value 10 Ω) placed in series with the mote.

The real value of the resistor was measured with a digital multimeter and the voltage was measured with a digital oscilloscope.

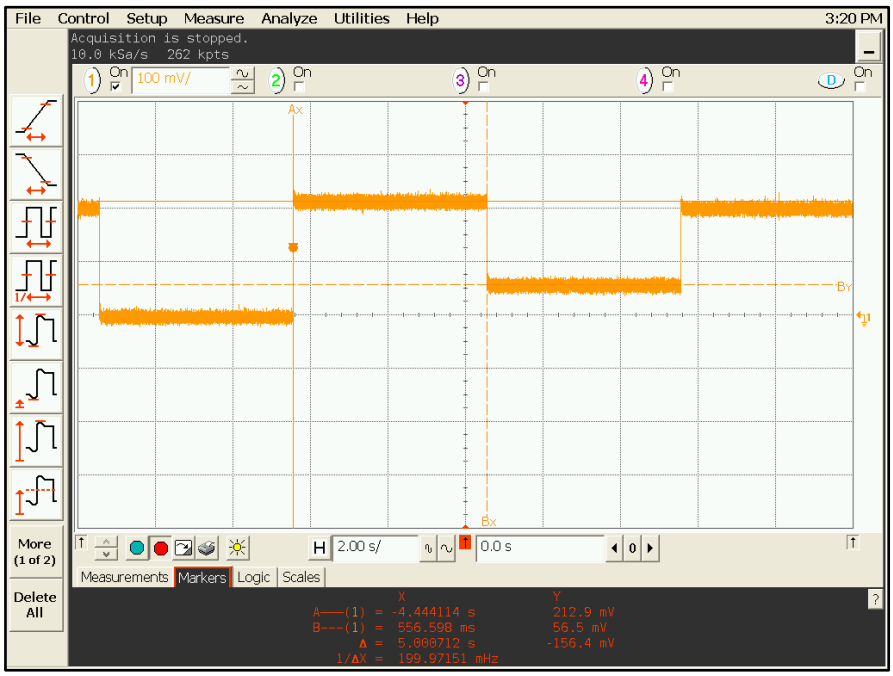

Fig. 4.7 Measured consumption

From the above figure, the average voltage corresponding to the maximum consumption is about:

$$V_{Havg} = 212.9\ mV$$

This corresponds to the consumption when the microcontroller, the radio and two of the three leds are on.

The voltage when the radio and the leds are off is not zero, the consumption of the microcontroller has to be considered. This value is obtained from table 3.4:

$$I_{Lavg} = 1800\ \mu A$$

Considering both values and the measured value of the resistance in series, it is possible to calculate the batteries lifetime.

4.3.2 Batteries lifetime

The most suitable power source for the motes is batteries. Each mote needs 3V to work, or, considering batteries, 2 units of AA batteries.

Nowadays, market is full of different brands that offer many different kinds of batteries, but, considering the low environmental impact of the network, the most suitable are rechargeable batteries. The reason to choose this kind of batteries is they can be recharged several times, which involves saving money in batteries and the possible contamination that batteries can cause if they are not recycled in the correct way.

Rechargeable batteries can be based on different technologies. The first ones where based on NiCd, but due to memory effect, after each recharge the life cycle was reduced. Now the principal technology is NiMH; they have no memory effect and higher charge and discharge capacity which makes them more useful than the previous ones.

One of the possible rechargeable batteries to be used is Ansmann AA-NiMH 2850 mAh.

Table 4.1 Battery characteristics

Maximum charge voltage	1.5 V
Nominal voltage	1.2 V
Nominal capacity	2850 mAh
Standard charge (charge current/charge time)	270 mA/16 h
Fast Charge (charge current/charge time)	2700 mA/1.1 h

With values from table 4.1 and the ones calculated in the previous section, it is possible to calculate the duration of the batteries.

Let's assume a DC of 50% and a period of 10 minutes:

$$I_{Pavg} = (V_{Havg}/R \times T\_H + I_{Lavg} \times T\_L)/(T\_H + T\_L\ ) \quad \textbf{(4.1)}$$

V_Havg was measured over R = 10 Ω (measured value: 9.927 Ω);

$$I_{Pavg} = 11.623\ mA$$

Considering the nominal capacity shown in table 4.1, it is obtained:

$$\frac{Capacity}{I_{Pavg}} = \frac{2850\ mAh}{11.623\ mA} = 245.19\ h$$

$$245.19\ h \cong 10\ days, 5\ hours, 12\ minutes$$

Hence, it is possible to have motes working with 50% DC for more than 10 days.

CHAPTER 5. 6LOWPAN DATABASE AND SERVER

Throughout this chapter it is explained how to save information in databases in order to enable future searches through the server, and how to make this information more appealing presenting it in graphs.

Before storing the information, it is required to install databases software. Furthermore, it is also necessary to choose an optimum way to develop the web through which the data would be searched. Considering this and the advantages of working with free and open software, the chosen software would be XAMPP.

Fig. 5.1 XAMPP

XAMPP is an open source cross-platform web server package. This package consists mainly of the Apache Server, MySQL database, PHP and Perl. XAMPP is used as a free web server capable of serving dynamic pages and is available not only for Linux but Windows, MacOS and Solaris.

- Apache HTTP Server: is a web server used to serve both static content and dynamic web pages on the World Wide Web. In February 2009 became the first web server to surpass the 100 million web site milestone.
- MySQL: is a powerful database management system which has more than 6 million installations. To administer MySQL databases one can choose between different tools such as command-line, MySQL Administrator and, the recommend one, phpMyAdmin.
- phpMyAdmin is a free web-based administration interface implemented in PHP.
- PHP: is a scripting language originally designed for producing dynamic web pages. Nowadays it has evolved and can be used in graphical applications or in a command-line interface. It can also be embedded into HTML code.
- Perl: is a high-level interpreted programming language. Perl borrows features from other languages including C, shell scripting and others.

5.1 XAMPP installation and configuration

To install XAMPP it is necessary to download .zip, .tar or .exe file; in this case, considering that Ubuntu is the operative system used, the .tar file is the one required. It can be downloaded from the official apache friends' website (http://www.apachefriends.org/en/xampp-linux.html) and its size is about 57 MB.

Once it has been downloaded, it has to be installed using the command shell, but before that, it has to be extracted.
Open a command shell. It is necessary to execute the commands as Ubuntu administrator, that is super user (write sudo su or sudo –s command and your password), and located at the Desktop directory or where the .tar.gz file has been placed, use:

> tar xvfz *filename* –C *route*

Where *filename* is the complete name of the .tar file and *route* is where it is going to be extracted. The default filename download from the link is *xampp-linux-1.7.1.tar.gz*, and the recommended route is */opt*. These are the variables used in this application.

Once it is done, the next step is to start XAMPP. In order to do it, use the command as super user:

> /opt/lampp/lampp start

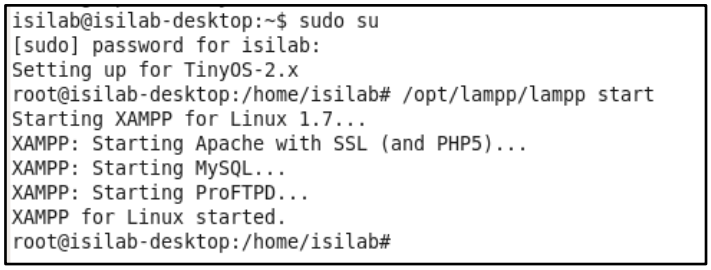

Fig. 5.2 XAMPP start

Now XAMPP has been installed and is running. It is also possible to create a desktop launcher to start or stop XAMPP. To do that, right click on the desktop and move down to create launcher, select type application and name it. At the command space write:

sudo */opt/lampp/share/xampp-control-panel/xampp-control-panel*

And click OK. It is necessary to do that with administrator privileges (sudo command). Now a new icon will appear at Desktop to allow start or stop XAMPP without opening a shell.

Fig. 5.3 XAMPP control panel

The next steps are referred to how to configure security and passwords. It is recommended using the same password previously defined for the super user in Ubuntu.

Open a browser and type *localhost*; the XAMPP page will appear, then, choose the language. On the left side of the main page there is a menu, choose *Security* to modify the default configuration.

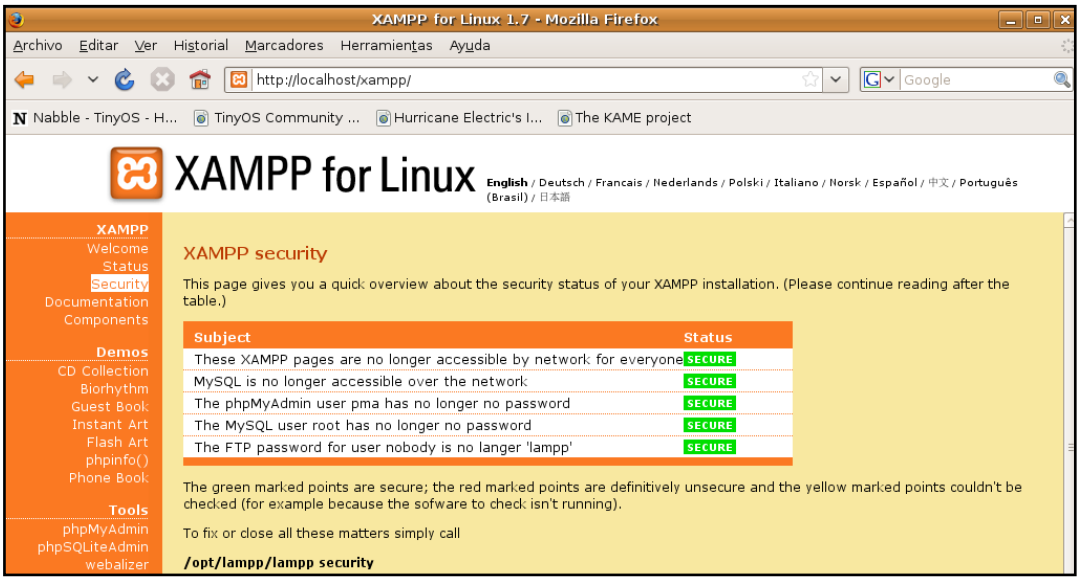

Fig. 5.4 XAMPP Security

By default, probably all the applications would be configured as unsecure (in a red box). It is strongly recommended to modify it by typing on a shell:

```
> /opt/lampp/lampp security
```

This will start an interactive application. It consists on a series of questions; just follow the steps to set all the necessary passwords:

- XAMPP password access
- MySQL password access, set the password for the root which is not available by default
- FTP password

After doing that, refresh the web page and an authentication required box will appear. Introduce the user and password previously defined and then check all the items are secure now.

When the passwords are set, a file is created containing the username and passwords that have been entered. If the xampp\security\xampp.users file is examined in a text editor, the password will be shown encrypted for security reasons.

5.2 Storing information in databases and tables

One of the options from the left menu at the main XAMPP web page is PhpMyAdmin. This application is included in the package and it is basically a tool to easily manage all databases and tables from MySQL. PhpMyAdmin allows the creation of databases and tables in a really intuitive manner. The main steps to create a database and a table are:

1) Enter a name for the database and click *create*.
2) Then, enter a name for each table included in the created database.
3) Fix the numbers of fields (rows) for each table and click *go*.
4) Indicate the type of each field (int, float, char ...). Look up the references for more information about each type.
5) Finally, click *save*.

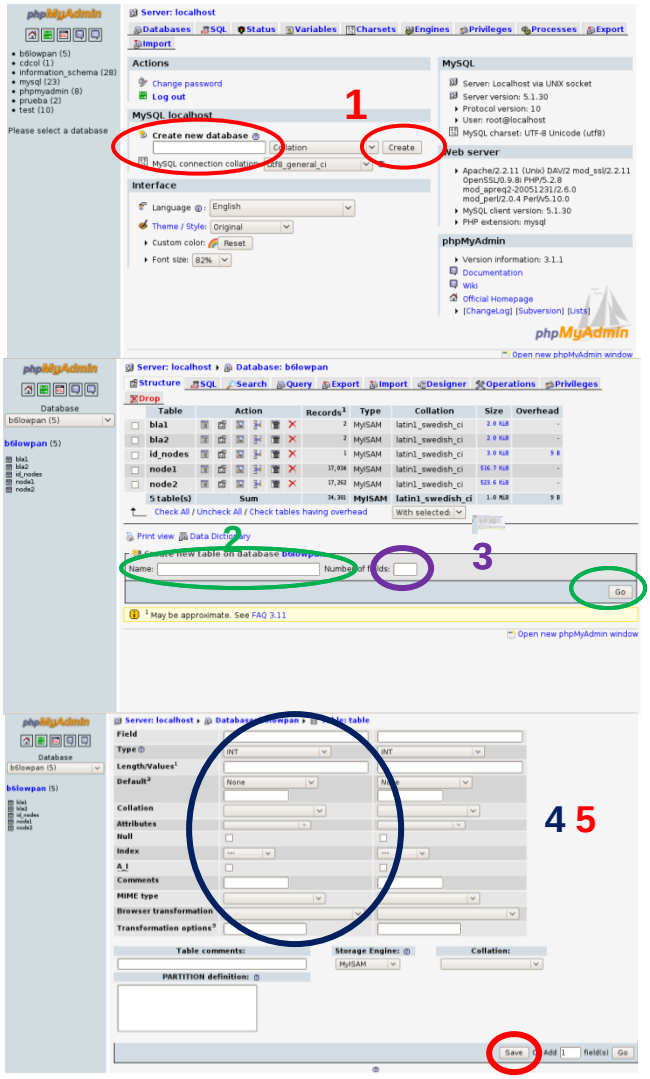

Fig. 5.5 Steps to create database and tables

A new database and an empty table have been created. The database defined in this case is able to follow the relational database model, which is based in storing data as relations. Each relation can be understood as a group of data, paying no attention to the order in which data has been stored.

The purpose of the relational model is to provide a declarative method for specifying data and queries: directly state what information the database contains and what information is required from it; and let the database management system take care of storing the data and retrieval system for getting queries answered.

Adding the information to the tables can be done in many different ways, depending on the final applications. The most suitable for this project is to add information with PHP code (File capturetest.php included in Annex F).

PHP implements some simple functions (such as *mysql_query*) to easily copy the information to a specified table. But before adding data to tables, it is necessary to have previously created the table.

Using one of the Python applications previously explained, all the information received from the motes on the computer is stored in a text file. The information format of this txt file has been established to facilitate later data storage.

```
28.26 50.09 315.41 52.11 2.53 40
28.27 48.48 318.23 48.75 2.59 41
```

Each line is referred to a different packet sent by the mote. Each packet contains a series of variables and each variable is separated from the others with a blank space. With the *gets* function, PHP code reads each line and executes a query to the specified database. Once, all the lines (packets) have been read and updated into the database, the text file is cleared.

The PHP code to add information to tables follows the next diagram:

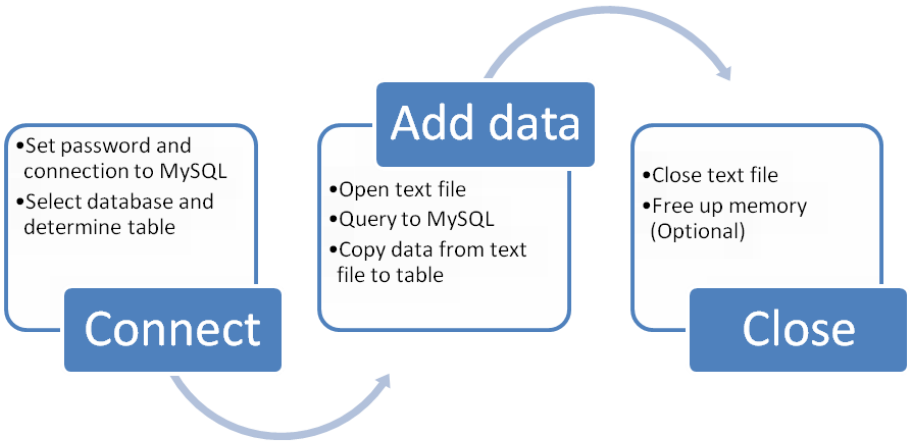

Fig. 5.6 Database upload diagram

Considering the period of the receiving data, it is necessary to execute this code in an appropriate time. The temporary window of the code and the period of the receiving data have to be very close to be efficient. This can be done by adding a refresh header at the top of the PHP file.

```
> header ('Refresh:180');
```

At the time of this writing, tests have been done with a period of 30 seconds and exactly the same temporary refresh for the code. It involves new data stored at the selected table each 30 seconds. Secondary tests have been done with new sent information every 3 minutes.

The temporary window in which data is sent can be established by the user depending on the application and its interests. Considering an environmental monitoring, receiving new data each roughly 10 minutes is more than enough. It is supposed that the environment will not change in such a quickly manner.

Having new data each 10 minutes means a total number of 144 packets each day if there are no losses.

5.3 Display information

When the information has been already stored in the database, it is necessary to choose an appropriate system to display data requested by the users connected to the server.

One of the best options is to use the JpGraph library (http://www.aditus.nu/jpgraph/). The library is completely written in PHP and can be used to create numerous types of graphics such as linear, bar and three-dimensional plots.

Download the *.tar.gz* file from the website and open a shell. As super user (sudo su) execute the following command:

> sudo tar xvfz *nameofthefile.tar.gz* –C *route*

Where *nameofthefile.tar.gz* is the name of the downloaded file and route is where ii is going to be installed. It is recommend to specify the */opt* route, the same route used for installing the XAMPP package.

Once this library has been installed, the next step is to write a second PHP file (File graphics.php). At the top of the file, it is necessary to add paths for the main libraries of JpGraph, they vary depending on the type of graphs required:

- jpgraph.php → basic library
- jpgraph_line.php → line plot extension
- jpgraph_date.php → classes to handle date scaling

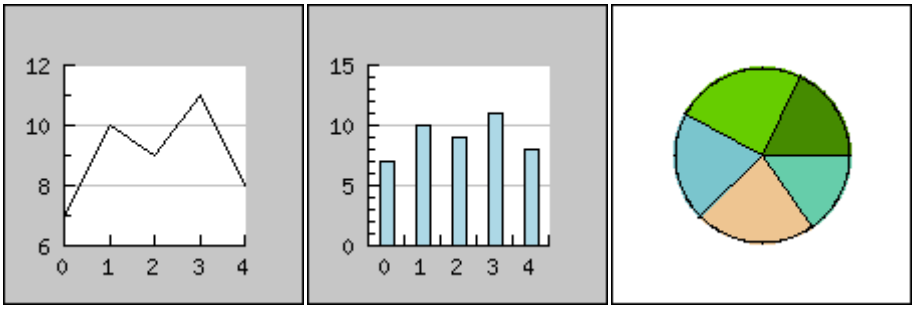

Fig. 5.7 Jpgraph examples

Included in this PHP file it is necessary to specify some parameters for each possible data and graph required:

- the database and table containing the information,
- the variables of the table which are required,
- the type of graph to display the values of the variable,
- margin, texts and size of the graphic.

It has to also include the connection and queries, and close the database.

Now it has been defined how to save data into tables, and how to display it in graphs. The last step is to create an HTML file (File *6lowpan.html*) to let the users consult all data by using the server.
It is necessary to add PHP code at the top and at the end of the file to open and close the connection to MySQL. The rest of the file is written in HTML code.

The next two images show the initial web page for the data server and the query page (File web.php).

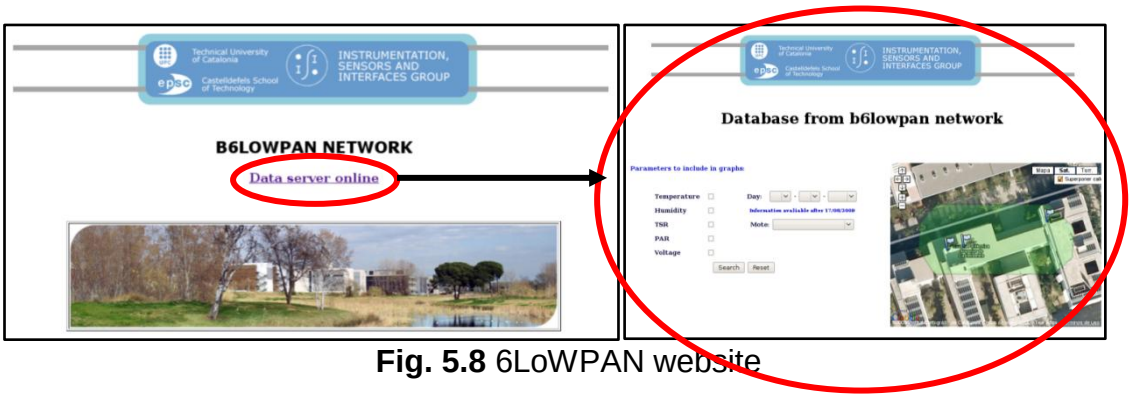

Fig. 5.8 6LoWPAN website

The first one is just a presentation tool for the *B6lowpan network*. It is linked to the second one, the database browser, where the user can select the variable to include in the graphs, define mote source and the date. There is also a map to easily locate the future position of each mote of the network, including information about IPv6 addresses and the future position of the base station.

It has to be mentioned that, during the tests, the motes and the base station where placed in different locations. All the nodes where placed at the isi-lab (laboratory 125).

Finally, once the user knows which mote and variable are required, by simply clicking on *Search* button, the graphic with the selected data will appear as shown in fig 5.8.

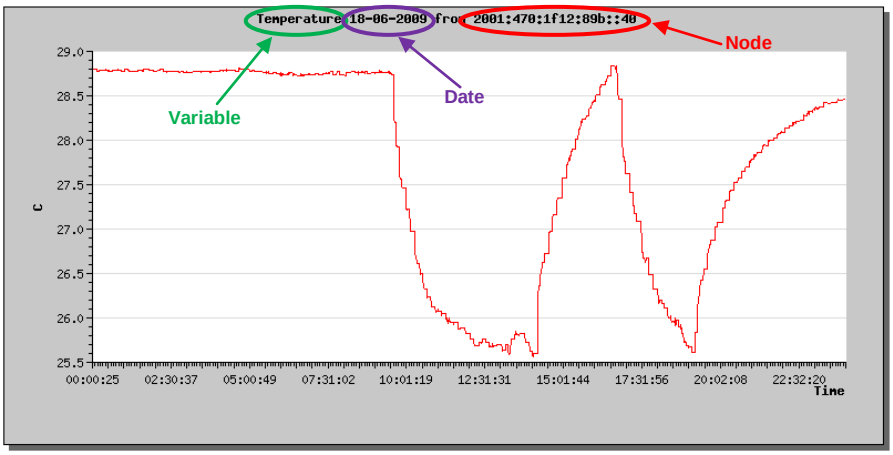

Fig. 5.9 Temperature graphic

At the top of the graph some information is displayed: which variable has been selected, the data and the sending node.

In case of a failure in base station failure, some data may seem constant for the user, because the file containing the new data has not been updated, but the application will still be working and printing graphics.

In case of batteries failure, all values will remain constants because the node is not able to sent new information to the base station. By choosing the voltage variable, it will be easy to discover if the values are reliable or not. When the voltage is under the 2.15 V value, the readings are not reliable, because the mote is not able to provide enough power to the sensors and the received data is wrong.

CONCLUSIONS

The aim of this project was the development of a wireless sensor network to supervise a determined environment, specifically, the EPSC campus.

This network has been developed with 6LoWPAN support, that is, each node has its own IPv6 address and can be connected to the Internet. The devices sent information to a central point were data is collected and available to be consulted through a website. The network has been running correctly during the last days and the website is available inside the EPSC network.

The objectives established at the beginning of the project have been reached. Other objectives were added during the development, for example, the database and the server, and they have also been performed.

Future improvements

Some improvements may be applied to the 6LoWPAN network in the near future.

At the end of August 2009 the new release of Berkeley WEBS will be available, and, included in this new release, there will be some interesting improvements such as TCP support and low power listening interface implemented among others.

UDPEcho motes not only sent information, they can also receive data from the Base Station. Included in Annex E it is a modification of *Listener.py* to sent data to the UDPEcho motes instead of reading what they sent. In future applications motes can be programmed to send data to the Base Station under request.

Other improvements can be implemented on the database and the server:

- Error control system to check possible mistakes in the form and displaying the graphs.
- Same variable from different motes displayed in the same graph, graph containing real time data... And study other options that JpGraph offers.
- Included in the query site there is a map that indicates the future location of the motes, but the API developed by Google offers other options. For more information about it see XX

As well as those changes are suggested, other sensors to control different environmental variables can be added to the motes using the external I/O.

BIBLIOGRAPHY

[1] Carlos De Morais Cordeiro and Dharma Prakash Agrawal, “Ad hoc and sensor networks”. Google books: http://books.google.es/books?id=D24L4ygKFngC&printsec=frontcover&hl=ca

[2] TinyOS main webpage with information and tutorials: http://www.tinyos.net/

[3] RFC 4919: http://tools.ietf.org/html/rfc4919

[4] RFC 4944: http://www.ietf.org/rfc/rfc4944.txt

[5] Arch Rock 6LoWPAN presentation: http://asdsupport.archrock.com/docs/Arch_Rock_Overview.pdf

[6] Internet of the Things: http://zachshelby.org/tag/ipso/

[7] Main Ubuntu commands: https://help.ubuntu.com/7.04/basic-commands/C/ar01s03.html

[8] The sensor network musem: http://www.snm.ethz.ch/Main/HomePage

[9] Berkeley WEBS’ blip project website: http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip

[10] Information about UDP sockets: http://www.chuidiang.com/clinux/sockets/udp/udp.php

[11] Online Python guide: http://www.ibm.com/developerworks/opensource/library/os-python8/

[12] Information about Google maps API: http://www.desarrolloweb.com/manuales/desarrollo-con-api-de-google-maps.html

[13] Hurricane Electric (tunnel broker): http://tunnelbroker.net

[14] Information about CVS in sourceforge: http://sourceforge.net/apps/trac/sourceforge/wiki/CVS

[15] Wikipedia: http://en.wikipedia.org/

[16] Sensirion SHT1x datasheet: http://www.sensirion.com/images/getFile?id=25

[17] Information and datasheet of Hamamatsu S1087 sensors: http://sales.hamamatsu.com/en/products/solid-state-division/si-photodiode-series/si-photodiode/part-s1087.php

[18] Arch rock Phynet base configuration: http://store.archrock.com/ProductDetails.asp?ProductCode=RSK-3000

[19] Jennic JN5139 kit: http://www.jennic.com/files/product_briefs/JN5139-EK000-PBv1.51.pdf

[20] Sensinode K210 NanoDevkit: http://www.sensinode.com/media/flyers/sensinode-k210-flyer-web.pdf

[21] Download XAMPP for linux: http://www.apachefriends.org/en/xampp-linux.html

[22] JpGraph main page: http://www.aditus.nu/jpgraph/

ANNEX A

How can I get IPv6?

First of all we need to go to http://tunnelbroker.net and register. Once we have done this, we will be able to create up to 4 tunnels.

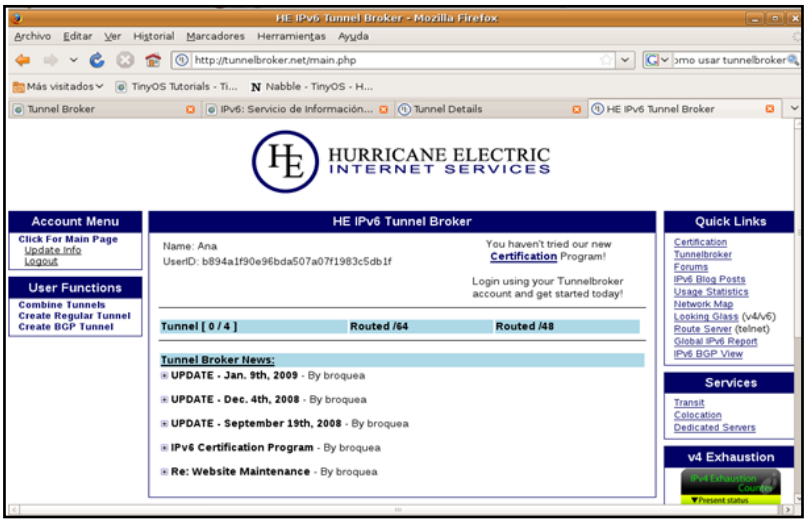

Figure A Hurricane electric website

To create a tunnel, choose “Create Regular Tunnel” option from the left menu, and we'll see the next figure:

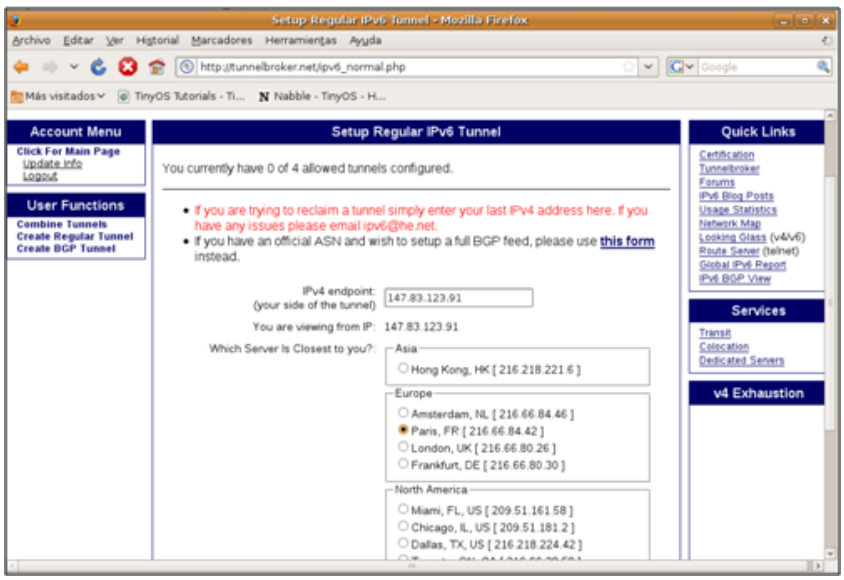

Figure B Create a tunnel

We have to specify our IPv4 endpoint (our actual IP address). We can obtain it by executing *ifconfig* on a terminal. Then, we choose the closest server, in this case, Paris' server will be selected.

If the tunnel has been created successfully we will see a message like "O.K.: Your tunnel has been allocated".

Once the tunnel has been created, we can click on "Tunnel Details" to see the main address of our tunnel. At the end of this page there is a menu to specify our OS, in this case, we select Linux-net-tools, and an example configuration will appear. With this example we can configure a script that will let us establish our Ipv6 address.

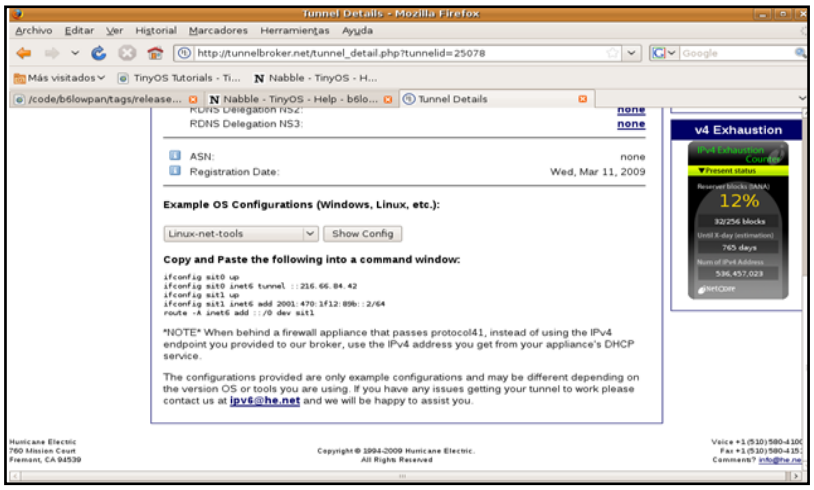

Figure C Configuration

Now create a file containing the specified commands and place it in a known route (i.e. Desktop). Here is an example of the script content:

```
#!/bin/bash

##IPv6 script##

ifconfig sit0 up
ifconfig sit0 inet6 tunnel ::216.66.84.42
ifconfig sit1 up
ifconfig sit1 inet6 add 2001:470:1f12:89b::2/64
route -A inet6 add ::/0 dev sit1
```

From a terminal, we change to the correct directory:

> *cd Escritorio*
> *more script* (with this command we can check the script file)
> *chmod +x script* (give permission to execute it)
> *sudo ./script* (execute the file)

Now we are able to have IPv6 connectivity. If we use *ifconfig* on the terminal, something similar to the next picture will appear.

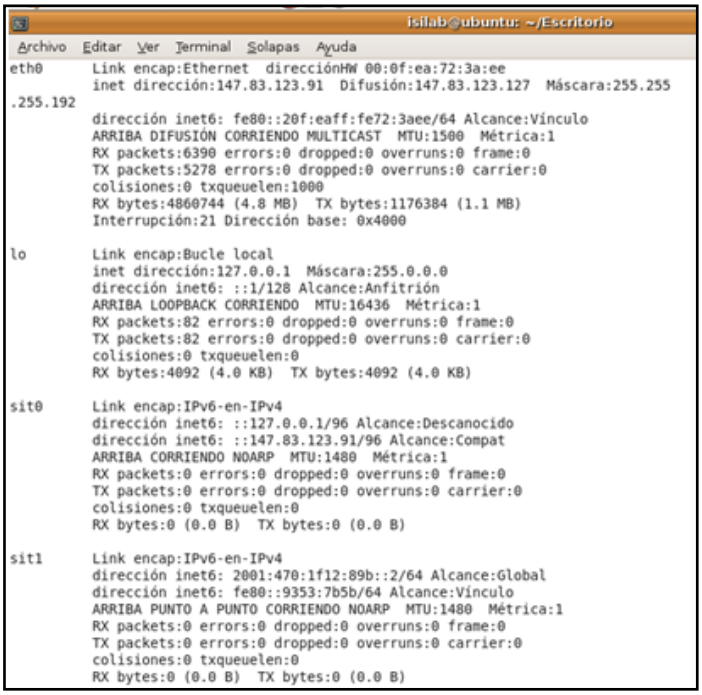

Figure D IP addresses

If we want to be sure that we have Ipv6, we have different options:

- http://www.kame.net/ →Go to this website and if you're able to see the dancing turtle means that you are using Ipv6
- http://www.6sos.org/conectividad.php# → Here you can also find another test for your connectivity
- http://ipv6.google.com/ → You can try to open google website Ipv6
- http://ipv6.he.net/ → This is also an Ipv6 page, and you can check your Ipv6 address.

ANNEX B

Installing TinyOs-2.x on Ubuntu 8.10

1. Go to System → Administration → Synaptic Package Manager. Type the administrator password (if required).

2. Go to Configuration → Repositories → Third party software → Add

3. Add the repository for TinyOS:

   deb http://tinyos.stanford.edu/tinyos/dists/ubuntu hardy main

   Hardy is the name of the previous Ubuntu version (8.04), the repository works with this name and not with Intrepid (the name of 8.10 version)

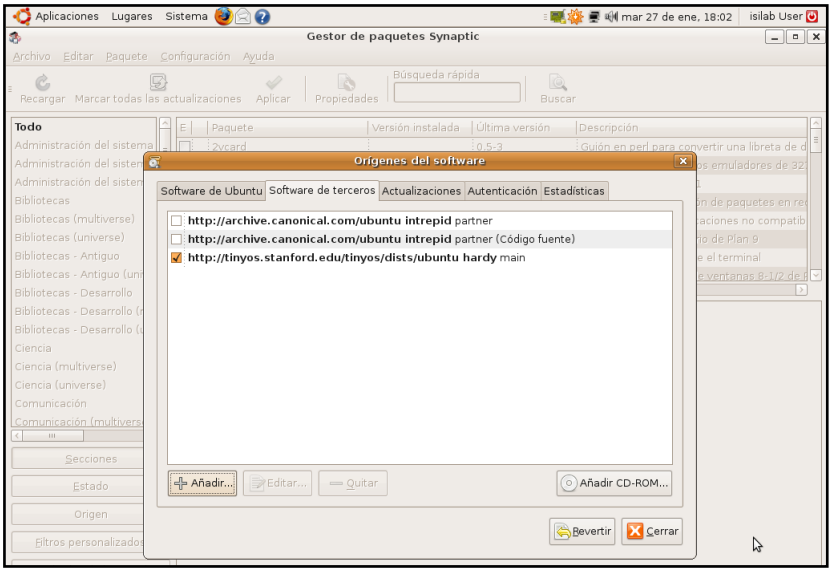

Figure E Adding repositories

4. Go to Applications → Accessories and open a terminal. With the following commands update the apt-cache and search the required packages:

   > sudo apt-get update (requires password)
   > sudo apt-cache search tinyos
   > sudo apt-get install tinyos-2.1.0

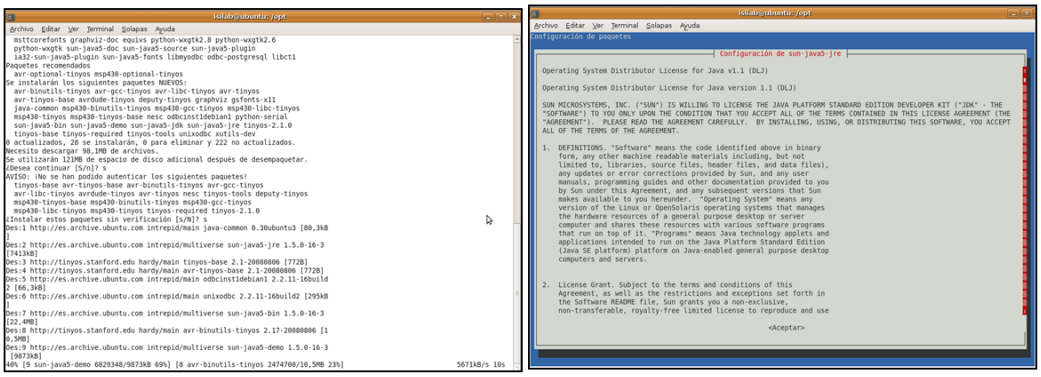

Figure F TinyOS and Java installation

5. The first image is TinyOS installation, and the second is Java installation that runs automatically during the first one. Accept Java Sun license to continue.

6. Install python:

   > sudo apt-get install python-dev

7. The last steps are referred to the configuration files. Edit tinyos.sh:

   > cd /opt/tinyos-2.x
   > sudo gedit tinyos.sh

   The file has been specified in 3.1.2.

8. Edit bash.bashrc file:

   > cd /etc
   > sudo gedit bash.bashrc

   Add the lines indicated in 3.1.2

9. Copy tinyos.sh in /etc/profile.d with Nautilus:

   > sudo nautilus

   A new window will appear, go to /opt/tinyos-2.x and copy the file. Then move to /etc/profile.d and paste it.

10. Check the installation and paths. Open the terminal and type:

    > tos-check-env

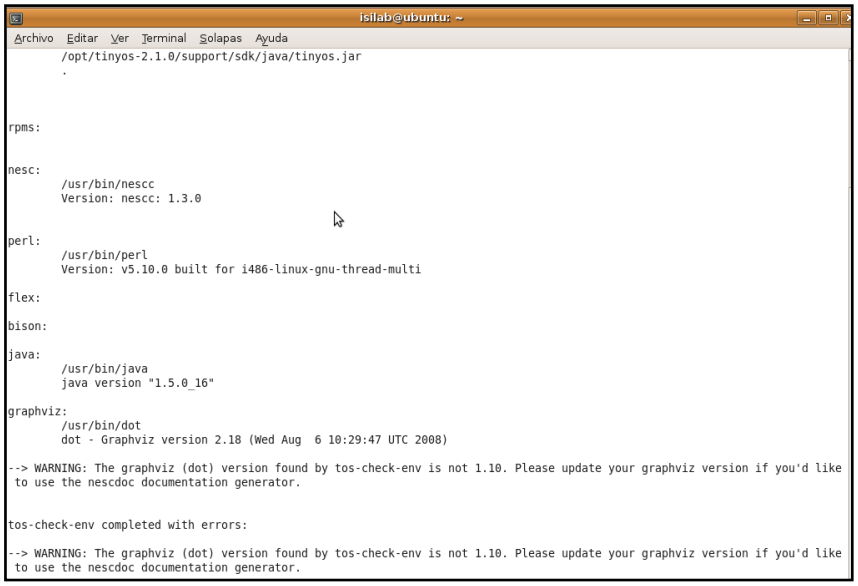

Figure G Checking TinyOS environment

ANNEX C

	JENNIC JN5139-EK000	ARCH ROCK PhyNet Base Configuration	SENSINODE K210 NanoDevkit
Cost	$449	$6,396 $3,996 (*PhyNet OEM Development Kit – IE)	997.5 € (+49€/ExtraSensor)
# nodes, routers, coordinators…	4 Sensor Boards, 2 High power modules	1 Server,2 Routers,10 Nodes *1 Server,1 Router,6 Nodes	2 NanoRouters N601 4 NanoSensors N711
OS	C, ZigBee stack	TinyOS	TinyOS, Nano
6lowpan, ZigBee, 802.15.4 support	Ok, 802.15.4/JenNet	6lowpan, 802.15.4	Ok
Current consumption [Typ,max]	JN5139-xxx-Myy Sleep = 2.6 uA Tx < 40/120 mA Rx < 40/45 mA	Sleep = 9uA Listen = 200uA Active [20,39] mA Repetidor 210 mA (OEMEngine)	Sleep = 0.5 uA Tx = 27 mA Rx = 27 mA (for complete RC230x)
V_{CC}	V_{CC} = [2.7; 3.6] V	V_{CC} = [2.75; 3.3] V	V_{CC} = [2; 3.6] V
Analog I/O	21 General and I/O Expansion Port	6/2	21 general I/O pins

Figure H Development kits I

	JENNIC JN5139-EK000	ARCH ROCK PhyNet Base Configuration	SENSINODE K210 NanoDevkit
Manufacturer support	Yes	Yes (web)	Yes (web)
Sniffer / IDE	Daintree / Jennet	- / IAR	- / NanoStack
API	Jenie (Network Stack API)	Software development kit (WinAVR)	NanoStack
Onboard sensors	Temperature, light and humidity	Temperature, light and humidity	N711 includes Temperature and Light sensors
Transceiver Output power and sensitivity	JN5139-xxx-Myy Tx 3/19 dBm Rx -97/-100 dBm	CC2420 (TI) Tx 0 dBm Rx -90 dBm	CC2431 (TI) Tx 0 dBm Rx -92 dBm
Modulation	-	O-QPSK	O-QPSK
Antenna connector	On board antenna / SMA / uFl	PCB or mini-coax u.FL connector.	SMD
Microcontroller	JN5139-xxx-Myy 16MHz 96kB RAM	TI MSP430 F1611 8MHz 10kB RAM	RC2301AT module with 8051 MCU 32MHz 8kB RAM
# bits	32 bits	16 bits	8 bits

Figure I Development kits II

ANNEX D

BaseStationC.nc

```
configuration BaseStationC {
}
implementation {
  components MainC, BaseStationP, LedsC;
  components CC2420MessageC as Radio;
  components SerialDispatcherC as SerialControl, Serial802_15_4C as
Serial;

  MainC.Boot <- BaseStationP;
  BaseStationP.RadioControl -> Radio;
  BaseStationP.SerialControl -> SerialControl;
  BaseStationP.UartSend -> Serial.Send;
  BaseStationP.UartReceive -> Serial.Receive;


  BaseStationP.RadioSend -> Radio;
  BaseStationP.RadioReceive -> Radio.Ieee154Receive;

  BaseStationP.RadioPacket -> Radio.Packet;
  BaseStationP.RadioIeeePacket -> Radio;

  BaseStationP.Leds -> LedsC;

  BaseStationP.PacketLink -> Radio;
  BaseStationP.LowPowerListening -> Radio;

  components ResetC;
  BaseStationP.Reset -> ResetC;

#ifndef SIM
  components SerialDevConfC as Configure;
  BaseStationP.ConfigureSend -> Configure;
  BaseStationP.ConfigureReceive -> Configure;

  components new TimerMilliC();
  BaseStationP.ConfigureTimer -> TimerMilliC;

  components IPAddressC;
  BaseStationP.IPAddress -> IPAddressC;

  components CC2420ControlC;
  BaseStationP.CC2420Config -> CC2420ControlC;
#endif
}
```

BaseStationP.nc

```
#include "CC2420.h"
#include "AM.h"
#include "Serial.h"
#include "devconf.h"
#include "lib6lowpan.h"

module BaseStationP {
  uses {
    interface Boot;
    interface SplitControl as SerialControl;
    interface SplitControl as RadioControl;
    interface Send as UartSend;
    interface Ieee154Send as RadioSend;
    interface Receive as UartReceive;
    interface Receive as RadioReceive;
    interface Packet as RadioPacket;
    interface Send as ConfigureSend;
    interface Receive as ConfigureReceive;
    interface Timer<TMilli> as ConfigureTimer;
    interface IPAddress;
    interface Ieee154Packet as RadioIeeePacket;
    interface PacketLink;
    interface LowPowerListening;
    interface CC2420Config;
    interface Leds;
    interface Reset;
  }
}
implementation
{
  enum {
    UART_QUEUE_LEN = 10,
    RADIO_QUEUE_LEN = 10,
  };
  uint16_t radioRetries = 10;
  uint16_t radioDelay   = 30;
  uint16_t serial_read;
  uint16_t radio_read;
  uint16_t serial_fail;
  uint16_t radio_fail;
  bool echo_busy;
  message_t echo_buf;
  config_reply_t *reply;
  message_t  uartQueueBufs[UART_QUEUE_LEN];
  message_t  *uartQueue[UART_QUEUE_LEN];
  uint8_t    uartIn, uartOut;
  bool       uartBusy, uartFull;
  message_t  radioQueueBufs[RADIO_QUEUE_LEN];
  message_t  *radioQueue[RADIO_QUEUE_LEN];
  uint8_t    radioIn, radioOut;
  bool       radioBusy, radioFull;
  task void uartSendTask();
  task void radioSendTask();
```

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

  void failBlink() {
    call Leds.led2Toggle();
  }
#define CHECK_NODE_ID if (0) return
  task void configureReply() {
    if (echo_busy) return;
    reply->addr = call IPAddress.getShortAddr();
    reply->serial_read = serial_read;
    reply->radio_read = radio_read;
    reply->serial_fail = serial_fail;
    reply->radio_fail = radio_fail;
    echo_busy = TRUE;
    call ConfigureTimer.startOneShot(100);
  }
  event void Boot.booted() {
    uint8_t i;
    CHECK_NODE_ID;
    for (i = 0; i < UART_QUEUE_LEN; i++)
      uartQueue[i] = &uartQueueBufs[i];
    uartIn = uartOut = 0;
    uartBusy = FALSE;
    uartFull = TRUE;
    for (i = 0; i < RADIO_QUEUE_LEN; i++)
      radioQueue[i] = &radioQueueBufs[i];
    radioIn = radioOut = 0;
    radioBusy = FALSE;
    radioFull = TRUE;
    echo_busy = FALSE;
    serial_read = 0;
    radio_read = 0;
    serial_fail = 0;
    radio_fail = 0;
    call RadioControl.start();
    call SerialControl.start();
    reply = (config_reply_t *)(&(echo_buf.data));
  }
  event void RadioControl.startDone(error_t error) {
    CHECK_NODE_ID;
    if (error == SUCCESS) {
      radioFull = FALSE;
#ifdef LPL_SLEEP_INTERVAL
      call
LowPowerListening.setLocalSleepInterval(LPL_SLEEP_INTERVAL);
#endif
    }
  }
  event void SerialControl.startDone(error_t error) {
    CHECK_NODE_ID;
    if (error == SUCCESS) {
      uartFull = FALSE;
    }
    reply->error = CONFIG_ERROR_BOOTED;
    post configureReply();
  }
```

```
  event void SerialControl.startDone(error_t error) {
    CHECK_NODE_ID;
    if (error == SUCCESS) {
      uartFull = FALSE;
    }
    reply->error = CONFIG_ERROR_BOOTED;
    post configureReply();
  }
  event void SerialControl.stopDone(error_t error) {}
  event void RadioControl.stopDone(error_t error) {}
  uint8_t count = 0;
  message_t* receive(message_t* msg, void* payload, uint8_t len);
  event message_t *RadioReceive.receive(message_t *msg,
                                        void *payload,
                                        uint8_t len) {
    CHECK_NODE_ID NULL;
    dbg("base", "radio message received (%i)\n", len);
    return receive(msg, payload, len);
  }
  message_t* receive(message_t *msg, void *payload, uint8_t len) {
    message_t *ret = msg;
    CHECK_NODE_ID NULL;
    atomic {
      if (!uartFull)
      {
        ret = uartQueue[uartIn];
        uartQueue[uartIn] = msg;
        uartIn = (uartIn + 1) % UART_QUEUE_LEN;
        if (uartIn == uartOut)
          uartFull = TRUE;
        if (!uartBusy)
          {
            post uartSendTask();
            uartBusy = TRUE;
          }
      }
      else
      dropBlink();
    }
    return ret;
  }
  task void uartSendTask() {
    uint8_t len;
    message_t* msg;
    atomic
      if (uartIn == uartOut && !uartFull)
      {
        uartBusy = FALSE;
        return;
      }
    msg = uartQueue[uartOut];
    len = call RadioPacket.payloadLength(msg) - sizeof(am_header_t);

    if (call UartSend.send(call
RadioIeeePacket.source(uartQueue[uartOut]),
                           uartQueue[uartOut], len) == SUCCESS) {
      call Leds.led1Toggle();
    }    else
      {
      failBlink();
      post uartSendTask();
```

```
  }
  event void UartSend.sendDone(message_t* msg, error_t error) {
    if (error != SUCCESS)
      failBlink();
    else
      atomic
      if (msg == uartQueue[uartOut])
        {
          if (++uartOut >= UART_QUEUE_LEN)
            uartOut = 0;
          if (uartFull)
            uartFull = FALSE;
        }
    post uartSendTask();
  }
  event message_t *UartReceive.receive(message_t *msg,
                                       void *payload,
                                       uint8_t len) {
    message_t *ret = msg;
    bool reflectToken = FALSE;
    CHECK_NODE_ID msg;
    dbg("base", "uartreceive len %i of 0x%x\n", len, call
SerialAMPacket.destination(msg));
#if defined(PLATFORM_TELOS) || defined(PLATFORM_TELOSB) ||
defined(PLATFORM_EPIC)
    WDTCTL = WDT_ARST_1000;
#endif
    atomic
      if (!radioFull)
      {
        reflectToken = TRUE;
        ret = radioQueue[radioIn];
        radioQueue[radioIn] = msg;
        if (++radioIn >= RADIO_QUEUE_LEN)
          radioIn = 0;
        if (radioIn == radioOut)
          radioFull = TRUE;

        if (!radioBusy)
          {
            post radioSendTask();
            radioBusy = TRUE;
          }
      }
      else
        dbg("base", "no enqueue\n");
    return ret;
  }
  task void radioSendTask() {
    uint8_t len;
    ieee154_saddr_t addr;
    message_t* msg;
    dbg ("base", "radioSendTask()\n");
    atomic
      if (radioIn == radioOut && !radioFull)
      {
        radioBusy = FALSE;
        return;
      }

    msg = radioQueue[radioOut];
```

```
    len = call RadioPacket.payloadLength(msg);
    addr = call RadioIeeePacket.destination(msg);

    if (addr != 0xFFFF) {
      call PacketLink.setRetries(msg, radioRetries);
      call PacketLink.setRetryDelay(msg, radioDelay);
    } else {
      call PacketLink.setRetries(msg, 0);
    }
#ifdef LPL_SLEEP_INTERVAL
    call LowPowerListening.setRxSleepInterval(msg,
LPL_SLEEP_INTERVAL);
#endif
    dbg("base", "radio send to: 0x%x len: %i\n", addr, len);
    if (call RadioSend.send(addr, msg, len) == SUCCESS)
      call Leds.led0Toggle();
    else
      {
      failBlink();
      post radioSendTask();
      }
  }
  event void RadioSend.sendDone(message_t* msg, error_t error) {
    CHECK_NODE_ID;
    if (error != SUCCESS)
      failBlink();
    else
      atomic
      if (msg == radioQueue[radioOut])
        {
          if (++radioOut >= RADIO_QUEUE_LEN)
            radioOut = 0;
          if (radioFull)
            radioFull = FALSE;
        }
    post radioSendTask();
  }
  event message_t *ConfigureReceive.receive(message_t *msg,
                                            void *payload,
                                            uint8_t len) {
    config_cmd_t *cmd;
    uint8_t error = CONFIG_ERROR_OK;
#if defined(PLATFORM_TELOS) || defined(PLATFORM_TELOSB) ||
defined(PLATFORM_EPIC)
    WDTCTL = WDT_ARST_1000;
#endif
    if (len != sizeof(config_cmd_t) || msg == NULL) return msg;
    cmd = (config_cmd_t *)&msg->data;
    switch (cmd->cmd) {
    case CONFIG_ECHO:
      break;
    case CONFIG_SET_PARM:
      call CC2420Config.setChannel(cmd->rf.channel);
      call IPAddress.setShortAddr(cmd->rf.addr);
      call CC2420Config.sync();
      radioRetries = cmd->retx.retries;
      radioDelay   = cmd->retx.delay;
      break;
    case CONFIG_REBOOT:
      call Reset.reset();
      break;
```

```
case CONFIG_KEEPALIVE:
      return msg;
    }
    if (!echo_busy) {
      reply->error = error;
      post configureReply();
    }
    return msg;
  }


  event void CC2420Config.syncDone(error_t error) {

  }

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

  event void ConfigureTimer.fired() {
    call Leds.led2Toggle();
    if (call ConfigureSend.send(&echo_buf, sizeof(config_reply_t))
!= SUCCESS)
      echo_busy = FALSE;
  }
#endif
}
```

UDPEchoC.nc

```
#include <6lowpan.h>
#include "TestDriver.h"

configuration UDPEchoC {
} implementation {
  components MainC, LedsC;
  components UDPEchoP;
  components new HamamatsuS10871TsrC() as Sensor;
  components new HamamatsuS1087ParC() as Sensor2;
  components new SensirionSht11C() as Sensor3;
  components new DemoSensorC() as Sensor4;

  UDPEchoP.Boot -> MainC;
  UDPEchoP.Leds -> LedsC;
  UDPEchoP.ReadTSR -> Sensor.Read;
  UDPEchoP.ReadPAR -> Sensor2.Read;
  UDPEchoP.ReadExtTemp -> Sensor3.Temperature;
  UDPEchoP.ReadHum -> Sensor3.Humidity;
  UDPEchoP.ReadVolt -> Sensor4.Read;

  components new TimerMilliC();
  components IPDispatchC;

  UDPEchoP.RadioControl -> IPDispatchC;
  components new UdpSocketC() as Echo,
    new UdpSocketC() as Status;
  UDPEchoP.Echo -> Echo;

  UDPEchoP.Status -> Status;
  UDPEchoP.StatusTimer -> TimerMilliC;
  UDPEchoP.IPStats -> IPDispatchC.IPStats;
  UDPEchoP.RouteStats -> IPDispatchC.RouteStats;
  UDPEchoP.ICMPStats -> IPDispatchC.ICMPStats;

  components RandomC;
  UDPEchoP.Random -> RandomC;
  components UDPShellC;

#ifdef DBG_TRACK_FLOWS
  components TestDriverP, SerialActiveMessageC as Serial;
  components ICMPResponderC, IPRoutingP;
  components new TimerMilliC() as Mark;
  TestDriverP.Boot -> MainC;
  TestDriverP.SerialControl -> Serial;
  TestDriverP.ICMPPing -> ICMPResponderC.ICMPPing[unique("PING")];
  TestDriverP.CmdReceive -> Serial.Receive[AM_TESTDRIVER_MSG];
  TestDriverP.IPRouting -> IPRoutingP;
  TestDriverP.DoneSend -> Serial.AMSend[AM_TESTDRIVER_MSG];
  TestDriverP.RadioControl -> IPDispatchC;
  TestDriverP.MarkTimer -> Mark;
#endif
}
```

UDPEchoP.nc

```
#include <IPDispatch.h>
#include <lib6lowpan.h>
#include <ip.h>
#include "UDPReport.h"
#include "PrintfUART.h"
#include "SBT80ADCmap.h"
#define REPORT_PERIOD 300L

module UDPEchoP {
  uses {
    interface Boot;
    interface SplitControl as RadioControl;
    interface UDP as Echo;
    interface UDP as Status;
    interface Leds;
    interface Timer<TMilli> as StatusTimer;
    interface Read<uint16_t> as ReadTSR;
    interface Read<uint16_t> as ReadPAR;
    interface Read<uint16_t> as ReadExtTemp;
    interface Read<uint16_t> as ReadHum;
    interface Read<uint16_t> as ReadVolt;
    interface Statistics<ip_statistics_t> as IPStats;
    interface Statistics<route_statistics_t> as RouteStats;
    interface Statistics<icmp_statistics_t> as ICMPStats;
    interface Random;
  }
} implementation {
  bool timerStarted;
  udp_statistics_t stats;
  struct sockaddr_in6 route_dest;
#define CHECK_NODE_ID
  enum {
    STATUS_SIZE = // sizeof(ip_statistics_t) +
    sizeof(route_statistics_t) +
    sizeof(icmp_statistics_t) + sizeof(udp_statistics_t),
  };
  event void Boot.booted() {
    CHECK_NODE_ID;
    call RadioControl.start();
    timerStarted = FALSE;
    call IPStats.clear();
    call RouteStats.clear();
    call ICMPStats.clear();
    printfUART_init();
    stats.temp = 0;
    stats.hum = 0;
    stats.tsr = 0;
    stats.par = 0;
    stats.volt = 0;
#ifdef REPORT_DEST
    route_dest.sin6_port = hton16(7000);
    inet_pton6(REPORT_DEST, &route_dest.sin6_addr);
    call StatusTimer.startOneShot(call Random.rand16() % (1024 *
REPORT_PERIOD));
#endif
    dbg("Boot", "booted: %i\n", TOS_NODE_ID);
    call Echo.bind(7);
    call Status.bind(7001);
}
```

```
  event void RadioControl.startDone(error_t e) {  }

  event void RadioControl.stopDone(error_t e) {  }

  event void Status.recvfrom(struct sockaddr_in6 *from, void *data,
                             uint16_t len, struct ip_metadata *meta)
{ }
  event void Echo.recvfrom(struct sockaddr_in6 *from, void *data,
                           uint16_t len, struct ip_metadata *meta) {
    call Leds.led1Toggle();
    CHECK_NODE_ID;
//    call Echo.sendto(&route_dest, &data, len);
  }
  event void ReadTSR.readDone(error_t result, uint16_t data)  {
    if (result == SUCCESS){
      stats.tsr = data;
      call ReadPAR.read();
    }
  }
  event void ReadPAR.readDone(error_t result, uint16_t data)  {
    if (result == SUCCESS){
      stats.par = data;
      call ReadExtTemp.read();
    }
  }
  event void ReadExtTemp.readDone(error_t result, uint16_t data) {
    if (result == SUCCESS){
      stats.temp = data;
      call ReadHum.read();
    }
  }
  event void ReadHum.readDone(error_t result, uint16_t data) {
    if (result == SUCCESS){
      stats.hum = data;
      call ReadVolt.read();
    }
  }
  event void ReadVolt.readDone(error_t result, uint16_t data) {
    if (result == SUCCESS){
      stats.volt = data;
    }
  }
  event void StatusTimer.fired() {
    uint8_t status[STATUS_SIZE];
    nx_struct udp_report *payload;
    CHECK_NODE_ID;
    if (!timerStarted) {
      call StatusTimer.startPeriodic(1024 * REPORT_PERIOD);
      timerStarted = TRUE;
    }
    call ReadTSR.read();
    payload = (nx_struct udp_report *)status;
    stats.sender = TOS_NODE_ID;
    call RouteStats.get(&payload->route);
    call ICMPStats.get(&payload->icmp);
    memcpy(&payload->udp,&stats,sizeof(udp_statistics_t));
    call Status.sendto(&route_dest, status, STATUS_SIZE);
  }
}
```

ANNEX E

Listener.py

```
#!/usr/bin/python

import MySQLdb
import socket
import string
import UdpReport
import re
import sys

port = 7000

if __name__ == '__main__':

    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.bind(('', port))

    while True:
      (data, addr) = s.recvfrom(1024)
      print 'Receiving packet from: %s/%d' % (addr[0],addr[1])
        if (len(data) > 0):

            rpt = UdpReport.UdpReport(data=data,
data_length=len(data))

            print rpt
            myfile = file("test.txt","a")
            print >> myfile, rpt
            myfile.close()
```

Send.py

```
#!/usr/bin/python

import socket
import re
import sys

PORT = 7
DEST_ADDR = ('2001:470:1f12:9d1::41', PORT)

if __name__ == '__main__':

    data = 'bb'
    dest = DEST_ADDR

    print 'Opening socket...'
    s = socket.socket(socket.AF_INET6, socket.SOCK_DGRAM)
    s.sendto(data,dest)
    s.close()
```

ANNEX F

Capturetest.php

```
<?php

header('Refresh:180');
$b = time();
$d1 = date("y/m/d",$b);
$d2 = date("G:i:s",$b);
$password='******';
$bd='b6lowpan';
$table1='node1';
$table2='node2';

$link = mysql_connect('localhost', 'root', $password);

            if (!$link) {
                  die('Not possible to connect with MySQL: ' .
mysql_error());
            }

            $db_selected = mysql_select_db($bd,$link);
            if (!$db_selected) {
                  die ('Not possible to connect with database: ' .
mysql_error());
            }

$file = "/opt/tinyos-2.x/berkeley/blip/apps/UDPEcho/util/test.txt";
$fp1 = fopen($file,'r');

            while (!feof($fp1)) {
                  $info = fgets($fp1);
                  echo "Leido $info";
                  list ($var1, $var2, $var3, $var4, $var5, $var6) =
explode(" ", $info);

                  if ($var6 == 40) {
                        mysql_query("INSERT INTO $table1 VALUES
('$var1', '$var2', '$var3', '$var4', '$var5', '$var6', '$d1',
'$d2')");
                  }

                  else if ($var6 == 41) {
                        mysql_query("INSERT INTO $table2 VALUES
('$var1', '$var2', '$var3', '$var4', '$var5', '$var6', '$d1',
'$d2')");
                  }
            }
fclose($fp1);

$fp1 = fopen($file,'w');
fclose($fp1);
?>
```

ANNEX G

Sensors calibration

Temperature and humidity sensor

SHT10 is a digital output sensor, and, as indicated in the datasheet, it is fully calibrated. It is not necessary to do any calibration for this sensor, but a comparison between the measured obtained with it and the ones obtained with a probe has been done.

The probe is a thermometer hygrometer available at the laboratory (model: DO 9406).

Parameter	Sensor measurement	Probe measurement	Error	Relative error
Temperature (ºC)	27.37	27.2	0.17	0.62 %
Humidity (%RH)	49.76	46	3.76	7.55 %

The probe was not able to measure the relative humidity at the moment of the measurements. The comparison was made with the RS thermometer/ hygrometer available at the laboratory which accuracy is not as good as a probe, but can be used to have a reference.

As can be seen in the above table, the value obtained for the temperature are similar to the one specified by the manufacturer in figure 3.7. And is really close in case of the relative humidity.

Light sensors

S1087 for sensing photosynthetically active radiation (PAR) and the S1087-01 for sensing the entire visible spectrum (TSR) are manufactured by Hamamatsu, which does not provide information about calibration, but provides a graph from which the equations can be obtained (figure 3.9)

The TSR can be calibrated comparing data obtained with a luxometer, but in case of PAR, there is no instrument to measure it available at the laboratory. There is a way to know the relation between both variables, calculating the area of the sensors (integrating both equations obtained from the figure).

The result obtained is that PAR is almost 20% of the total solar radiation.

Parameter	Sensor measurement	Luxometer measurement	Error	Relative error
TSR	507.20	415.7	91.5	18.04 %
PAR	67.33	20% of 415.7	15.81	23.48 %

The relative error is really high, but it is due to the large response time of the sensors and the high probability to have variations in the scenario. For example, a person walking round the sensor can make the measurements invalid.