Bachelor's Thesis

Degree in Computer Engineering

Specialization
Information Technologies

Experimental implementation of an IoT platform for automatic actuation in a building

Defended on July 1st, 2019

Author:
Claudia Martínez Alquézar

Director:
Dr. Davide Careglio

Department:
Computer Architecture

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

Abstract

Experimental implementation of an IoT platform for automatic actuation in a building

by Claudia Martínez Alquezar

The Internet of Things (IoT), and specifically its use for Smart Buildings, has been on the rise in the last few years thanks to improvements in wireless communications and hardware that make it easier than ever to interconnect devices. One of the main points of interest of Smart Buildings in particular is the ability to improve energy efficiency and reduce waste, automatically adjusting the building's resources and providing users with higher comfort.

With this concept in mind, and trying to improve the proactiveness of Smart Buildings to more efficiently manage resources, the thesis "Design and simulation of an interoperable IoT platform for automatic actuation in buildings"[1] proposes a solution and tests it in a virtual scenario, modelling and simulating this implementation.

Furthering this idea, in this project we brought aspects of this theoretical solution to a real environment. We implemented a Wireless Sensor Network (WSN) through the building, with real sensors and gateways, in order to monitor and document the results of a real implementation and its viability. Our conclusions draw attention to the differences between the simulated and real sensor implementations, as well as the obstacles that have been found during the experiment, offering solutions for future designs.

Contents

1 Introduction

The Internet of Things (IoT) is described as a network of interconnected objects and devices, both physical and virtual, which interact and exchange data without the need of human-computer interaction[2][3]. These objects are connected to the Internet in order to communicate with the others, and make use of sensors to collect the data that will later be sent and received.[4]

The IoT market has been on the rise during the last few years[5][6], due in part to the improvements in technology, wireless communications and cheaper hardware[7], but also to the rapidly increasing number of connected devices and better global connectivity to the Internet - as of 2019, there are 4.39 billion internet users[8].

Smart Buildings are an emerging IoT technology due to their benefits regarding efficiency, in particular towards energy. Reducing energy consumption and operating costs is one of the main targets of Smart Buildings, as the building sector is considered one of the biggest contributors to world energy consumption and greenhouse gas emissions, and the single biggest contributor in Europe.[9][10]

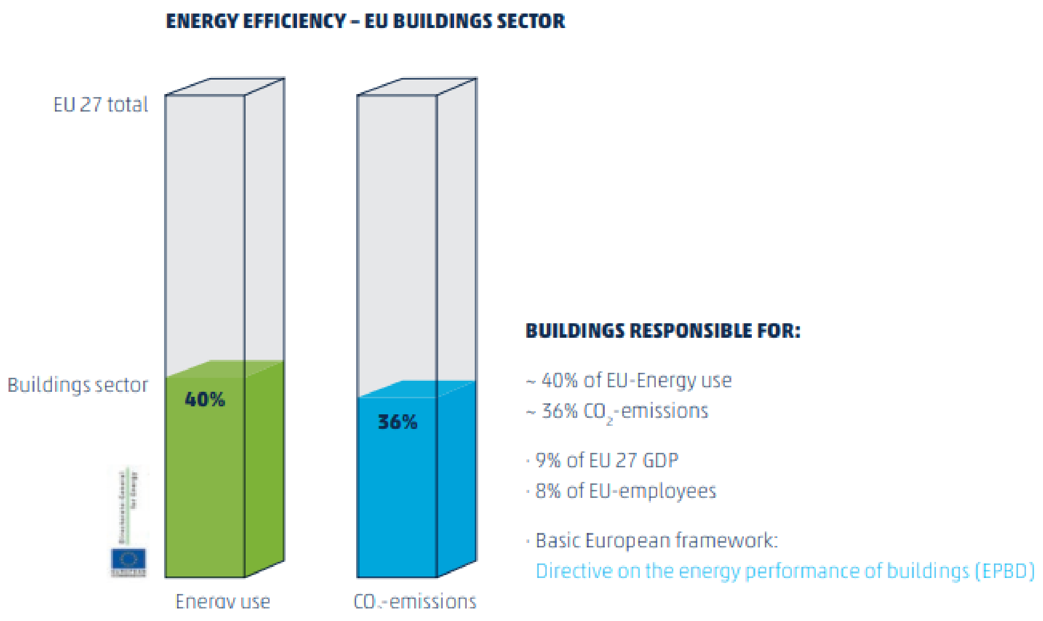


Figure 1: Energy consumption in buildings - Europe

Regarding these issues, David Sembroiz's thesis [1] brings up the idea that the lack of proactiveness and intelligence of Smart Buildings limits the amount of efficiency that can actually be achieved with their current implementation. He proposes to use a Smart Manager containing machine learning techniques to predict users' behaviour, anticipating their movements and needs. This would allow the system to schedule changes and avoid any unnecessary waste of energy.

The goal of his project was to achieve a more efficient resource management of the UPC Campus Nord D6 building by using this Smart Manager, together with sensors and IoT devices. He ran simulations to obtain an overall view of what results could be expected if this platform was actually implemented in the building. He concluded that predicting and scheduling actions was possible, leading to an increase in occupant comfort and decrease in energy consumption; and that a real deployment was feasible at the time of writing.

When the thesis was published, in 2016, IoT was rapidly gaining momentum and interest from both the tech industry and consumers, many of which bought their first IoT device this year. Particularly, the Amazon Echo, which was officially released in mid-2015, was the most popular IoT device with 2 million units sold in the first nine months of 2016 [11]. Smart speakers like the Echo can extend their usage by connecting to other IoT devices and allowing users to control them from the same hub, which provides an incentive to add more smart appliances to the household.

Since 2016 the IoT sector has experienced a boost in the number and variety of products offered, increasing the range of applications addressed by the field. This has resulted in the growth of the IoT market focused on the production of devices meant for homes. Nowadays this range of products goes from lights, locks or doors to all kinds of kitchen appliances. [12]

The increased interest in IoT also applies to Smart Buildings, and while the focus of Smart Building design used to be primarily oriented towards workplaces, they have started to be incorporated into homes, which could also benefit from their promises of sustainability, energy efficiency and comfort.

With this context in mind we can conclude that, nowadays, Smart Buildings are still a topic of interest and that the idea of implementing this kind of solution could prove very beneficial. With the advancements in IoT and related technologies, this implementation is not only possible but easier than when it was designed, and it could provide us with valuable information regarding the viability of deploying this kind of solution in other buildings or in a larger scale.

For these reasons, following Sembroiz's work, this project aims to test this solution in a real environment. To do so, the main platform should be deployed and implemented along with the needed infrastructure - sensors, gateways and middleware - in the D6 building, and the information obtained from the system should be recorded and studied. This project also aims to document what obstacles are found in a real-life implementation, and what aspects of the initial design and goals will have to be modified due to unexpected problems.

The results of the experiment will be monitored and compared to the ones obtained in the simulations, to determine if the virtual models generated in the theoretical design are similar to the real ones, and if the implementation of this system would actually be beneficial in regards to both comfort and energy efficiency.

1.1 Initial scope

The initial aim of the project was to carry out an experiment with the end goal of implementing the theoretical design in a real environment. To begin implementing the system, the original ServIoTicy platform and its documentation had to be studied before deploying the rest of the architecture. ServIoTicy would be vital in providing communication capabilities to homogenize all the received data and supply it to the system.

Next, the WSN deployment would start by programming the sensor nodes and gateways in order to collect the information and communicate with the ServIoTicy platform, as well as with each other. Once everything was working as described in the original design, the Smart Manager in the simulator would automate the control of virtual HVAC, lights and computers according to the real sensor data requested from ServIoTicy. The energy consumption results obtained from this automation would be compared with the original results and with the current consumption of energy of the building.

1.2 Final scope

To our knowledge, the ServIoTicy platform that the Smart Manager uses in the original design was already installed on a server located in the department, and ready to be used for this project. Once the development of the project started, it came to our attention that said server could not be started and thus the platform wasn't functional anymore.

Since the source code for the platform is publicly available on GitHub [13], we started a process of reinstallation on a local computer, but were faced with a multitude of problems. ServIoTicy hadn't been updated since 2015, and many of the packages that it depended on are outdated or unavailable, which created a series of failed dependencies that couldn't be fixed. Attempts to contact the creators and obtain an updated version also proved unfruitful, and a decision was taken to approach the project in a different way, to avoid using the platform and prevent causing further delays in the rest of tasks due to this issue.

The focus of the project shifted to the sensor network, specifically to the deployment process and the environmental information obtained from the data.

Once the WSN is implemented and working as intended, the sensors can provide values similar to the ones obtained in the theoretical thesis. The comparison between these real values and the simulated ones indicate how precise the simulations were and, consequently, how close the predictions about potential energy savings are to reality. With this information we can decide whether this design is recommended for real-life environments and what changes should be implemented in future projects to improve on it.

1.3 Goals

The main goal to be achieved in this project is the installation of a wireless sensor network in the UPC Campus Nord D6 building, which monitors temperature, humidity, luminosity and room occupancy. The purpose is to obtain real sensor data about this building's environment and compare it to previously simulated values.

First, the structure of the network is designed, giving special attention to energy consumption and communications range along other possible IoT issues. The network deployment is then carried out by configuring the devices and installing them through the building, after which the sensor data can be collected. These values are expected to provide more accurate information about the viability of installing a real automated system in this building.

1.4 Stakeholders

Developer
The project is developed by a single person. This is the person responsible for research, documentation and implementation, as well as for managing the project and accomplishing deadlines.

Director
The director of this project is Davide Careglio, Associate Professor at the Department of Computer Architecture (DAC).

Beneficiaries
While this project doesn't have direct beneficiaries, it will bring information about the possibility of a real implementation of the aforementioned thesis[1], and will help to see what benefits we can obtain from it. In the future, if this project is successful, this system can be applied to other buildings in order to achieve more efficiency in energy consumption and more comfort.

2 Background

In this section we provide an overview of the state of the technologies behind the subject at hand, all of them related to the overall field of the Internet of Things. We present the possible problems and how current improvements are used to manage them.

2.1 Smart Buildings

Automated buildings are able to use automation to control the building's operations, including heating, ventilation, air conditioning, lighting, security and other systems [14]. But using IoT, this automation can go beyond creating a simple management system to control the building. Instead, with IoT sensors, data about users' behavior and the environment can be collected and analyzed to automatically adjust different elements of the building, which in turn can improve users' comfort and resource management [15], as well as cut down the operational costs [16].

Smart Buildings, on top of that, should not only automate operations but also be able to learn and adapt to changes in user behavior, building usage, occupancy and weather conditions - without sacrificing comfort or energy efficiency [1]. With this definition in mind, there aren't many examples of fully Smart Buildings operating nowadays to which we can refer.

One of these few examples, and the most similar to our platform, is Intel's RR4 Design Facility in Bangalore [17]. This building is equipped with 9,000 sensors, used to track and automate temperature, lighting, energy consumption, and occupancy in the building. In order to improve efficiency, Intel implemented a machine learning algorithm that predicts the appropriate temperature for every area of the building. The algorithm takes into account parameters such as occupancy and ambient temperature, and runs every two minutes to keep predictions current.

2.2 Sensors

With today's rapid advancements in technology and specifically in electronic chips, sensors are improving constantly in aspects that make them ideal for IoT, while also becoming more affordable, a vital condition for Smart Buildings - as large amounts of sensors are needed to fully automate a building. It also reduces the cost of embedding sensors, making it easier for manufacturers to create affordable smart devices and everyday objects with "smart" capabilities.

Sensors are now smaller and more durable, which allows us to place them in virtually any place without taking up considerable space. They have more processing power - as of now, manufacturers have created chips with up to 18 processor cores [18]. More computing power means more capabilities, performance and efficiency, which also helps reduce the consumption of energy needed for the automation to work.

Moreover, there are many types of existing sensors that can be useful for IoT and Smart Buildings in particular. Some of them are: sensors for temperature, light level, gas, chemicals, smoke, occupancy, infrared radiation, motion detection and humidity [19].

Wireless Sensor Networks have different requirements to other wireless networks, based on their objectives and components. Sensor networks usually require large numbers of small sensors, which operate on limited battery power in order to allow the flexibility of sensor placement needed for good coverage of the area [20]. Each sensor can usually last between 90 and 120 hours using AAA batteries while in the active mode [21]. For this reason, energy saving is a critical part of sensor networks, and the main way of achieving this is using low consumption modes - the sensors are kept in sleep mode until woken up to gather sensor data.

2.3 Communication technology

There is a broad range of communication technologies that can be used in IoT, and which one is used depends largely on the project and situation. As previously mentioned, power consumption is one of the main concerns for IoT systems, as the devices are usually battery-powered and replacing batteries is costly and inconvenient - especially given that sensors can be placed in remote, hard to reach locations that difficult this task even more.

Communication elements are some of the most energy-consuming, and they tend to be active and listening most of the time in IoT devices, so using low-consumption communication technologies can lead to a considerable reduction in overall energy consumption.

2.3.1 RFID

An RFID (Radio Frequency Identification) consists of tags and readers, with the tag being a microchip attached to the object as its identifier. The RFID reader communicates with the tag using radio waves.

RFID systems send radio transmissions to the RFID tags while the tag emits a unique identification code, which is collected by a reader. The data collected from the tag can then be sent to a device or stored for later use.

The advantage of this technology is its automated identification and data capture, which is a big advantage in regards to IoT, since it would allow to identify and track any object globally using the tags [22].

2.3.2 Bluetooth

Bluetooth is a short-range wireless communication technology aimed at providing communications between internet devices as well as between devices and the internet, while also simplifying data synchronization between compatible devices.

Bluetooth Low Energy (BLE) is designed to provide more intelligent management of connections in order to significantly lower power consumption. This is achieved by sending small amounts of data when it is needed, while otherwise putting the connection to "sleep" mode when it is not used and reducing the amount of energy wasted [23].

2.3.3 802.15.4

The IEEE 802.15.4 standard defines the characteristics of the physical and MAC layers for Low-Rate Wireless Personal Area Networks. This low rate WAN offers a simple and flexible protocol stack while allowing simple installations, reliable data transfers, low cost and low energy consumption. [24].

802.15.4 only addresses the physical and MAC layers, while the layers on top can be implemented using protocols such as Zigbee, Z-Wave and 6LoWPAN. This lightweight protocol stack is useful for applications that can work with low data rates and low latency, and a common application of this technology are Wireless Sensor Networks (WSN).

An IEEE 802.15.4 network is built with different types of devices, including end devices, coordinators and PAN coordinators. All nodes in the same network use the same radio channel and PAN. As the protocols used are optimized for short and periodical data transfers, the nodes in 802.15.4 networks are usually kept in sleep (low consumption) mode, in which all radio functionalities are switched off. Sleeping devices can wake up on synchronous interruptions programmed every certain time, or when an asynchronous event occurs, like sensor detection. However network coordinators must always be awake to manage messages from the sleeping devices. [25]

2.3.4 Zigbee/Xbee

Zigbee is a standard based on IEEE 802.15.4 developed specifically to offer a simpler, lower cost, lower consumption communication technology compared to other WAN technologies like Bluetooth or Wi-Fi. This makes it ideal for wireless personal area networks where battery life is critical. However this is at the expense of factors like communication range and data transmission rate. [26]

This standard uses the ISM band, particularly 868 MHz in Europe, 915 MHz in the USA and 2,4 GHz globally. However most applications tend to use the 2,4 Ghz band, as it is free and available to use everywhere.

Zigbee performs communications through a single frequency, that is, it uses a single channel. Usually one channel can be chosen for a network from among 16 possible ones. A Zigbee network can be theoretically made up of up to 65535 devices, but while in a practical setting this number is lower, the protocol is still able to control a significant amount of devices in the same network.

Another important feature is that it allows interoperability between different devices from different manufacturers, which is useful to create heterogeneous sensor networks.

XBee is the trade name of a family of radio communication modules from Digi [27] and is based on the Zigbee standard. Digi offers many Xbee modules, and while most use Zigbee some others are significant modifications of the standard.

2.3.5 6LoWPAN

The 6loWPAN protocol began as an idea to take advantage of the Ipv6 addressing power and combine it with the flexibility of low consumption and low processing power devices, providing a way for these small devices to be used in Internet of Things applications.

It allows the efficient sending of IPv6 packets within small frames on the data link layer, over IEEE 802.15.4 based networks, by offering encapsulation and header compression mechanisms.

3 Architecture

This section presents the overall structure of the implementation, introducing the software and hardware used and their features, and detailing the purpose of each component.

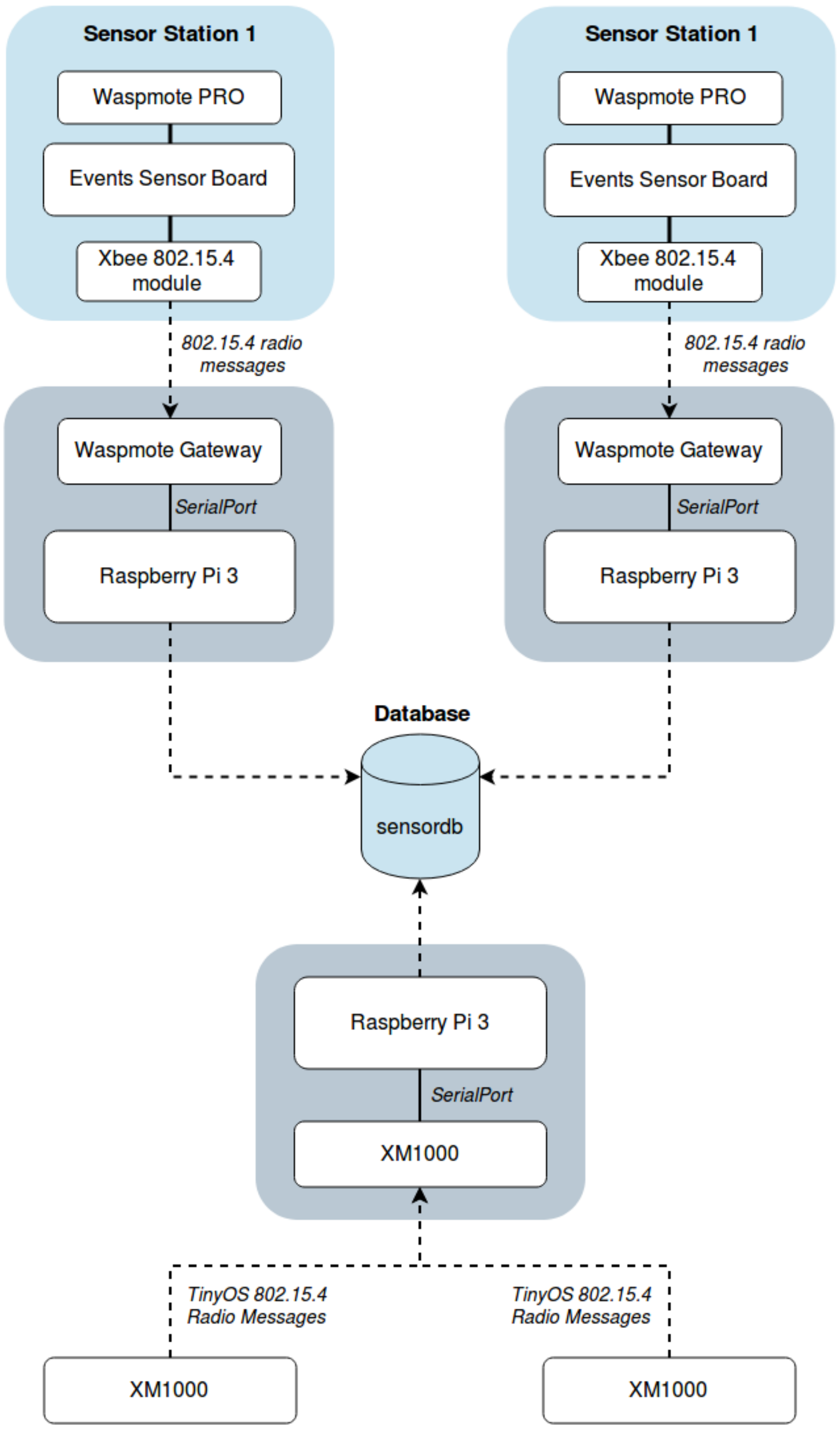


Figure 2: Architecture outline

The structure is composed by three small WSNs - two Waspmote Xbee networks and a XM1000 mote network - that combine into one. The sensors are spread through three floors and cover the Campus Nord D6 building, monitoring temperature, light, humidity and presence conditions. All the combined sensor data is transferred to a server and stored in the same database.

3.1 Software

In this section, we briefly describe the software used to implement our design.

- **TinyOS** [28] is an open source, BSD-licensed operating system designed for low-power wireless devices. It's used to flash the XM1000 motes with the desired software.
- **XM1000 Radio Sensing Software** [29] is the software that was developed specifically for the implementation of real XM1000 sensors in the first design. It's installed on the XM1000 motes.
- **Etcher** [30] is a cross-platform tool for flashing images to SD cards and USB drives. It's used to install Raspbian on the Raspberry Pi boards.
- **XCTU** [31] is a multi-platform application to interact with Digi RF modules. It's used to configure the Xbee modules.
- **Waspmote IDE** [32] is Waspmote's software development kit. It's used to write and upload code to the Waspmote boards.
- **Raspbian** [33] is the official OS for Raspberry Pi, Debian-based and highly optimized for this line. It's installed in our Raspberry boards.
- **Visual Studio Code** [34] is a code editor. It's used to program the scripts and any code that isn't developed on the Waspmote IDE.

3.2 Hardware

Most of the hardware used in this project was provided by the university. Table 1 lists the available types of hardware at the beginning of the project along with how many units were provided.

Device	**Units**
Raspberry Pi 3 Model B	3
Waspmote PRO (v.1.2)	2
XBee PRO S1 802.15.4 Wireless module	4
Waspmote Events Sensor Board v2.0	1
Waspmote Smart Cities PRO sensor board v1.4	1
Light Sensor (LDR)	2
Temperature Sensor (MCP9700A)	2
Humidity Sensor (808H5V5)	1
Presence Sensor (PIR)	1
Waspmote Gateway 802.15.4 PRO	2
AdvanticSys XM1000 mote	3
6600 mAh Rechargeable lithium-ion battery	2
2400MHz RP-SMA 2.1 dBi antenna	4

Table 1: Available hardware.

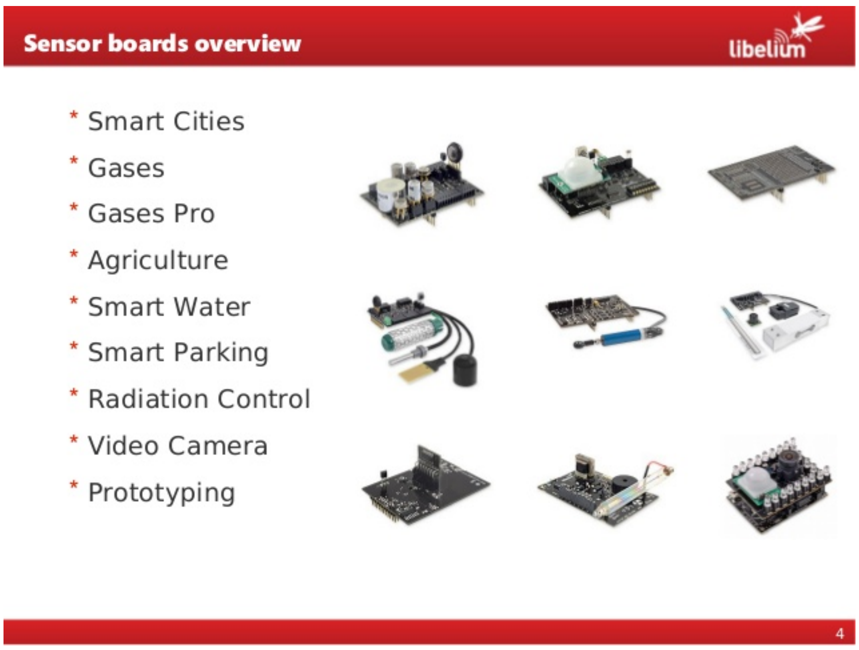
The original thesis used three real XM1000 motes in combination with the virtual ones, but the goal of this project was to replace those virtual sensors with physical ones. To do so we studied the Waspmote sensor boards and determined the best way to integrate them in the design, taking into account the differences in their functionality and aiming to provide as much homogeneity as possible in the overall installation. Ensuring that new sensors could be added and recognized by the WSN automatically and reliably was also a priority.

The final design of a WSN for the Waspmote devices uses *Sensor Stations*, comprised of 1 Waspmote PRO, 1 Sensor Board, and 1 Xbee module. These send the collected data to their assigned gateways, which consist of a Raspberry Pi and an Xbee module. The gateways process the sensor data received and store it in the database.

The XM1000 motes form their own sensor network, using one of them as a gateway. The network's data is collected by the gateway mote and forwarded to the Raspberry to which it is attached, and then stored in the database shared with the Waspmote network.

3.2.1 Waspmote

Waspmote [35] is an open source, modular, wireless sensor platform offered by Libelium [36]. They offer a series of advanced sensor boards and modules designed to be integrated in an easy and flexible way with the rest of the platform. Each sensor board is specialized in a different application of IoT and provides specific functions and sensors.



Sensor boards overview

- Smart Cities
- Gases
- Gases Pro
- Agriculture
- Smart Water
- Smart Parking
- Radiation Control
- Video Camera
- Prototyping

4

Figure 3: Waspmote sensor boards

Waspmote hardware architecture has been devised to work with extremely low energy consumption, which is vital in the design of IoT solutions.

Attaching a battery to the motes allows us to place the sensors anywhere in the building, without being restricted by the requirement of having an accessible energy source. But relying on batteries means that at some point they will have to be replaced or recharged, regularly losing access to the sensor's functionality and causing inconveniences for the users. The low consumption of Waspmote hardware provides batteries with a longer lifespan, thus increasing the sensors' autonomy.

In the following sections we describe the Waspmote sensor boards and modules used in this project.

3.2.1.1 Waspmote PRO

The Waspmote is the main part of the platform, a modular device that allows the user to attach different modules to it by plugging them into the available sockets (see figure 4). The idea behind this design is to integrate only the necessary modules, which can be changed based on needs.[37]

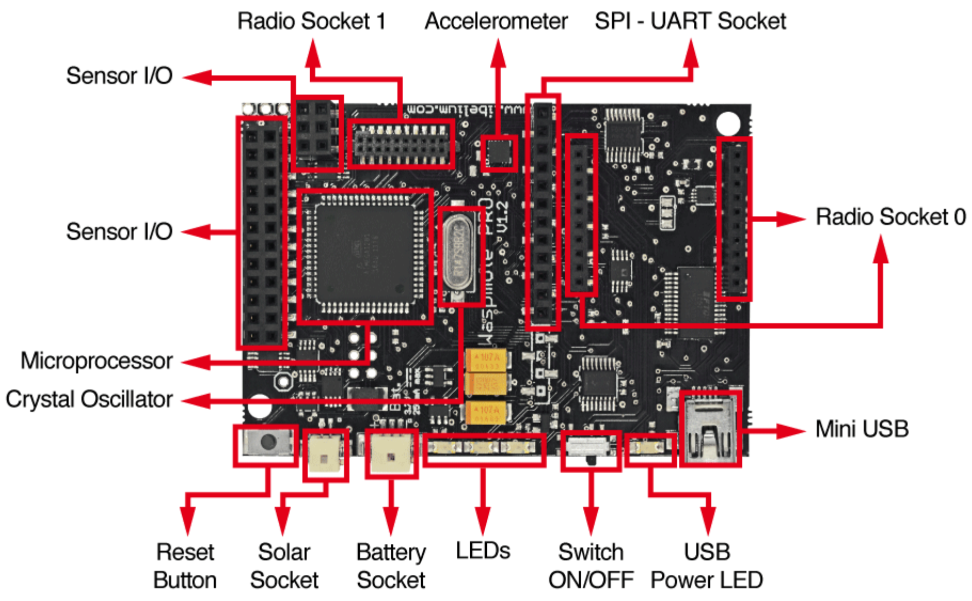


Figure 4: Waspmote PRO v1.2 top side and sockets

The two broad categories of available modules are sensor boards, which can monitor and collect data from environmental parameters; and radio modules, that allow the mote to communicate with other devices, transmit data and create sensor networks. Individual sensors can also be plugged into the microcontroller using the available sockets and pins.

Attaching modules provides the user with access to all their functionalities. The mote can then be programmed in a computer using the Waspmote IDE and the appropiate libraries [32], and the code is loaded to the mote via USB port.

The main specifications of the mote are the following:

- Microcontroller: ATmega1281
- Frequency: 14.74 MHz
- SD card: 2 GB
- Dimensions: 73.5 x 51 x 13 mm
- Temperature range: [-10 ºC, +65 ºC]
- Clock: RTC (32 kHz)

More information about the microcontroller's specifications can be found on its technical guide [37].

Waspmote can communicate with other external devices using its different input/output ports, which are:

Analog: The mote has 7 accessible analog inputs in the sensor connector, each of them directly connected to the microcontroller. The reference voltage value is 0V (GND), while the maximum value of input voltage is 3.3V.

Digital: The 8 digital pins can be configured as input or output depending on the needs of the application. A voltage value of 0V corresponds to logic value 0, while a voltage of 3.3V corresponds to a logic 1.

UART: There are two UARTs in Waspmote, UART0 and UART1. Several ports might also be connected to these UART ports through two different multiplexers, one for each. UART0 is shared by the USB port and the Socket0, which is used for radio modules. By default, the data signal in this UART is always switched to Socket0, and is temporarily switched when the USB needs to send info through the UART0. UART1 is shared by four ports (Socket1, GPS, Auxiliar1 and Auxiliar2). The port that needs to be used in an application can be configured in the program.

I2C: The I2C bus is used when two devices (accelerometer and RTC) are connected in parallel.

SPI: The microcontroller's SPI port is used to communicate with the micro SD card.

USB: The USB port is used for communication with a computer or compatible USB devices, which allows the microcontroller's program to be loaded. USB communications use UART0.

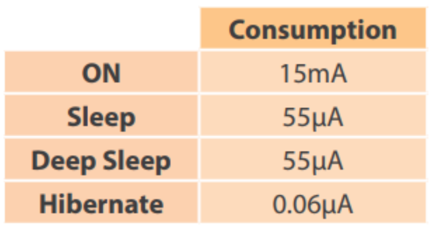
While the analog and digital pins allow the connection of simple sensors, the integration of sensors requiring some type of electronic adaptation stage or signal processing is done by the various sensor boards that can be attached. The connection between these boards and the mote uses the two 2x12 and 1x12 connectors seen in Figure 4 named *Sensor I/O* and *SPI - UART Socket* respectively. There are 8 sensor boards available for Waspmote PRO v1.2:

- Gases
- Gases PRO
- Events
- Smart Water

- Smart Water Ions
- Smart Cities
- Smart Parking
- Agriculture

This project uses the Events and Smart Cities sensor boards, each of them plugged into one of the Waspmote PRO microcontrollers.

In addition to the sensors and boards that can be attached, the microcontroller itself has a built-in Real Time Clock that provides a few functionalities. The RTC keeps Waspmote continually informed of the time, which allows the microcontroller to be programmed to perform time-related actions such as sleeping for certain amounts of time before continuing the execution of a program, or perform actions at specific intervals. Both relative and periodic alarms can be programmed in the RTC, allowing precise control of when the mote should wake up to capture sensor values and perform actions. This allows Waspmote to be in energy saving modes (Deep Sleep, Hibernate) and only waking up when required, reducing the consumption of energy. In figure 5 we can see the difference in power consumption between the different energy modes.



	Consumption
ON	15mA
Sleep	55µA
Deep Sleep	55µA
Hibernate	0.06µA

Figure 5: Waspmote PRO v1.2 power consumption

ON: Normal operation mode.

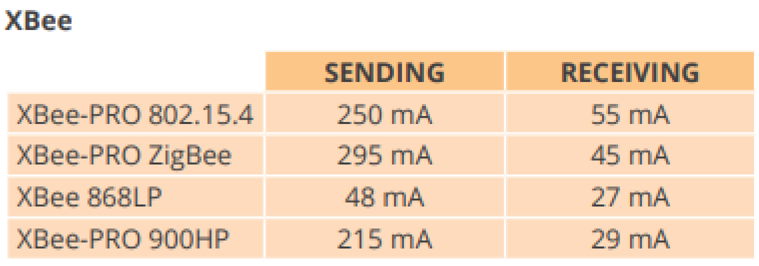
Sleep: The program is paused and the microcontroller passes to a latent state, from which it can be woken up by all asynchronous interruptions and by the synchronous interruption generated by the Watchdog. The duration interval of this state ranges from 32ms to 8s.

Deep Sleep: The program is paused and the microcontroller passes to a latent state from which it can be woken up by all asynchronous interruptions and by the synchronous interruption triggered by the RTC. The interval of this cycle can range from seconds to days.

Hibernate: The program stops and the microcontroller and Waspmote modules are completely disconnected. In this state, the only way to reactivate the device is by a previously programmed alarm in the RTC. The interval of this cycle can range from seconds to days. All devices except the RTC are disconnected from the battery, which causes this mode to have an almost negligible amount of energy consumption.

In this project, we use the Deep Sleep mode and the synchronous interruptions caused by the RTC in order to reduce energy consumption. The Waspmotes are in Deep Sleep mode most of the time, only awaking to collect data from the sensors at certain intervals. Additionally, this data isn't sent every time it is collected, instead it only does so in the event that the values have changed in a significant way compared to the last sample.

Communication and radio modules consume a large amount of energy (as it can be seen in Figure 6) - in fact, they amount for most of the Waspmote's power consumption, and sending values continuously would deplete the batteries too often. Thus, these adjustments were added to achieve a balance between precision and battery life. The specific adjustments are explained in Section 4.1. Our Waspmotes are powered by 6600 mAh rechargeable lithium-ion batteries provided by Libelium.


XBee

	SENDING	RECEIVING
XBee-PRO 802.15.4	250 mA	55 mA
XBee-PRO ZigBee	295 mA	45 mA
XBee 868LP	48 mA	27 mA
XBee-PRO 900HP	215 mA	29 mA

Figure 6: Power consumption of Xbee radio modules

The RTC also allows the user to obtain the current date and time at any given moment during the program. This project uses this functionality in order to collect information about the date and time right after the sensor data is sampled, and sends this information together with the data towards the WSN. This information is vital in order to study the evolution of environmental temperature and luminosity over time.

The RTC also has an internal temperature sensor, which provides data about the temperature of the board, and can be accessed by Waspmote through the I2C bus. However, the RTC's internal temperature sensor is not meant for air temperature sensing - instead, Sensor Boards are the recommended choice to collect this kind of data, and those are the ones that this project utilizes.

After an overall view of the main microcontrollers and their features, in the following sections we describe the boards and modules attached to them for this project.

3.2.1.2 Smart Cities Board

The Smart Cities board provides monitoring functionalities aimed at indoor environments and outdoor locations. Apart from the humidity, luminosity and temperature sensors, present in all the Libelium boards, the Smart Cities board includes three sensors destined to monitor cracks in buildings and structures, a linear displacement sensor (SLS095) for crack width, a single strand strain gauge for crack detection and a multiple strand strain gauge for crack propagation.[38]

For this project, we use the humidity, luminosity and temperature sensors, as those were the ones we were provided with and we have no use for the other sensors in this design.

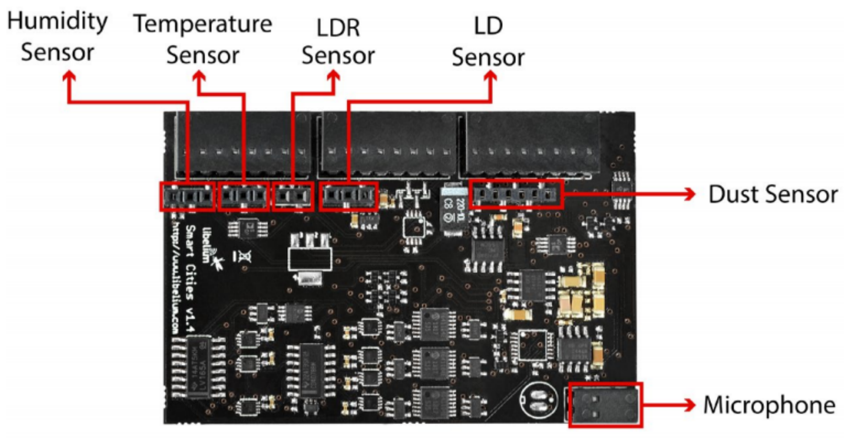


Figure 7: Smart Cities PRO Sensor Board top side and sockets

This board requires both the 3.3V and 5V power supplies output from the mote. The connected sensors are powered through three solid state switches, that allow the user to activate or deactivate the supply voltages for each sensor or group of sensors separately. These switches can be controlled from the program using the *SensorCities.setBoardMode* and *SensorCities.setSensorMode* functions, included in the necessary *WaspSensorCities* API library.

3.2.1.3 Events Board

The Events sensor board allows the simultaneous connection of up to 8 sensors. One of the main features of this board, compared to the Smart Cities board, is that the sensors are active on the board while Waspmote is in Sleep or Deep Sleep mode. This means that it can be programmed so that when a sensor picks up a value higher than a previously programmed threshold, a signal is generated which wakes the mote from its low consumption status and tells it which sensor has generated the signal. The value of these thresholds is programmed by the microcontroller through the I2C bus, as the system is controlled through digital potentiometers.

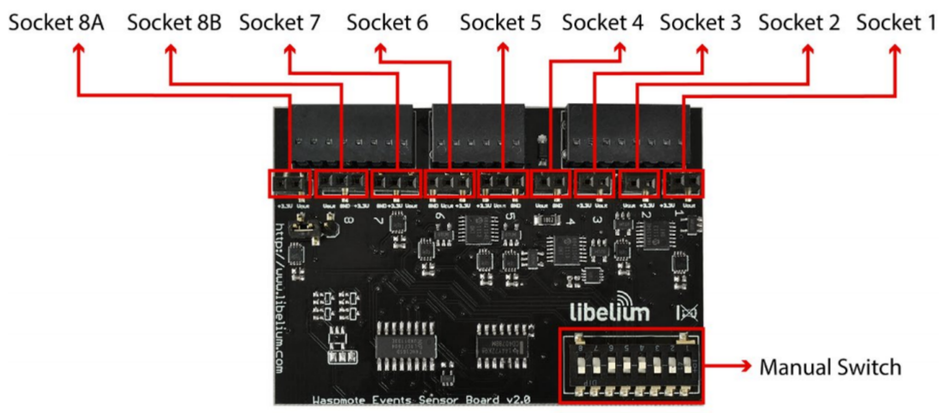


Figure 8: Events Sensor Board top side and sockets

The Events Sensor Board also includes two components which must be manually configured by the user: a manual switch for controlling the power of each sensor and a jumper to choose the output of one of the connectors. The power of each sensor must be enabled or disabled manually using a three-position 8 switches socket from which each of the connectors is controlled (see Figure 8). Additionally, this allows extra energy savings by disabling the sensors that aren't being used. In our case we only enabled the sockets needed for our 3 sensors: connector 5 (for the temperature sensor), connector 2 (for the light sensor) and connector 7 (for the presence sensor).

Each type of sensor must be connected to its corresponding socket. Information about the correspondence between sensors and sockets can be found in the board's technical guide [39]. The light sensor could be installed on sockets 1, 2, or 3 depending on the range of luminosity intensity to be measured. Due to the environment having generally a medium to high luminosity level, we selected socket 2.

All the instructions to handle the sensors and interruption functionalities built into the board are included in the *SensorEventv20* API library.

3.2.1.4 Sensors

This sections describes the sensors attached to the Events and Smart Cities sensor boards and their specifications.

Luminosity sensor (LDR)

- Resistance in darkness: 20MΩ
- Resistance in light (10lux): 5 ~20Ω

- Spectral range: 400 ~700nm
- Operating Temperature: -30ºC ~+75ºC
- Minimum consumption: 0uA approximately

This is a resistive sensor whose conductivity varies depending on the intensity of light received on its photosensitive part. Its spectral range (400nm – 700nm) coincides with the human visible spectrum, so it can be used to detect light and darkness in a similar way to the human eye.

This sensor is attached both to the Smart Cities and Events boards.

Temperature sensor (MCP9700A)

- Measurement range: -40ºC ~+125ºC
- Output voltage (0ºC): 500mV
- Sensitivity: 10mV/ºC
- Accuracy: ±2ºC (range 0ºC ~+70ºC), ±4ºC (range -40 ~+125ºC)
- Typical consumption: 6μA
- Power supply: 2.3 ~5.5V
- Operation temperature: -40 ~+125ºC

This is an analog sensor which converts a temperature value into a proportional analog voltage, using an internal diode to measure temperature. Its electrical characteristics have a temperature coefficient that provides a change in voltage based on the relative ambient temperature from -40°C (100mV) to 150°C (1.75V). The change in voltage is scaled to a temperature coefficient of 10.0 mV/°C, resulting in an output voltage of 0°C scaled to 500 mV. [40].

This sensor is attached both to the Smart Cities and Events boards.

Humidity sensor (808H5V5)

- Measurement range: 0 ~100%RH
- Output signal: 0,8 ~3.9V (25ºC)
- Accuracy: <±4%RH (a 25ºC, range 30 ~80%), <±6%RH (range 0 ~100)
- Typical consumption: 0.38mA
- Power supply: 5VDC ± 5%
- Operation temperature: -40 ~+85ºC

This is an analog sensor that provides a voltage output proportional to the relative humidity in the atmosphere with an accuracy of ±6%RH along the whole range and of ±4%RH between 30% and 80% humidity [41].

This sensor is attached to the Smart Cities board, but used to adjust the temperature readings instead of providing data on its own.

Presence sensor (PIR)

- Consumption: 100μA
- Range of detection: 6 ~7m
- Spectral range: ~10μm

This is a pyroelectric sensor mainly consisting of an infra-red receiver and a focusing lens that monitors the variations in levels of detected infra-red light, signaling a presence by setting its output signal high ('1' value), and a lack of presence by setting it low ('0' value). The 10μm spectrum corresponds to the radiation of heat from the majority of mammals as they emit temperatures around 36°C, making it useful to detect human presence.

This sensor is attached to the Events board.

3.2.1.5 Waspmote Gateway

This device acts as a data bridge or access point between the WSN and the receiving equipment. Depending on the model, it can work with different wireless protocols:

- XBee 802.15.4/ZigBee
- Sigfox
- LoRaWAN
- LoRa 868
- WiFi
- Bluetooth

In this project we use the Waspmote Gateway 802.15.4 PRO model, which comes with an attached XBee PRO S1 Wireless module (see Section 3.2.2) and allows us to use the low-cost IEEE 802.15.4 communication protocol [42].

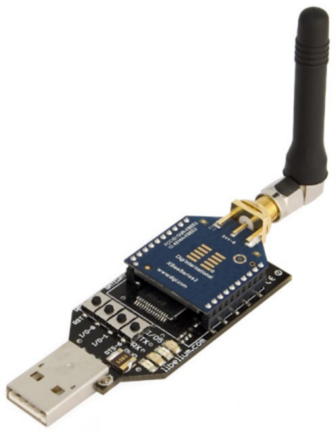


Figure 9: Waspmote Gateway 802.15.4/ZigBee with an attached radio module

The gateway is installed in the receiving equipment, which is responsible for processing, storing and using the data received depending on the specific needs of the application. This equipment can be any kind of device compatible with USB connectivity and with an available type A USB port. Once attached, a new communication serial port appears in the receiving device, directly connecting to the XBee module's UART. In this project the receiving equipment is a Raspberry Pi 3 Model B single-board computer, which processes the packets received from the Gateway and stores the collected sensor data in a database.

The Waspmote Gateway also acts a shield for the Xbee module, which allows us to connect it to a computer in order to modify the Xbee's configuration and update its firmware. These actions can be performed by using the XCTU application [31] provided by Digi [27], and are necessary to correctly set up the WSN and have more control over it, as well as to troubleshoot connection problems between the modules.

Both Waspmote Gateways use the Xbee modules and are attached to the two Raspberry Pi boards.

3.2.2 Xbee

As it has been previously mentioned, the Xbee modules used are XBee S1 802.15.4 PRO which support the IEEE 802.15.4 standar [43]. On top of the functionalities contributed by the standard, the XBee modules add new ones:

> **Node discovery**: The nodes can discover other nodes on the same network, which allows a node discovery message to be sent. When received, the rest of the network nodes respond with their data (Node Identifier, @MAC, @16 bits and RSSI).

Duplicate packet detection: reducing duplicate packets can make the network be more efficient.

This module has the following general specifications:

- Operating frequency band: ISM 2.4 GHz
- Ethernet protocols: UDP/TCP, DHCP client
- Operating temperature: -40 to 85ºC
- Antenna options: Integrated whip antenna, embedded PCB antenna, U.FL connector, RPSMA connector
- 5V/2.5A DC power input

Using Xbee allows for automatic and reliable recognition of new sensors by the current WSN without needing a complex configuration and setup for each sensor. As long as the sensors have an Xbee module and use the correct PANID/Channel, they can communicate with every other module on the same configuration and be automatically recognized by the WSN.

To achieve serial communication, the UART of both devices (the microcontroller and the XBee module) must be configured with compatible settings for the baud rate, parity, start bits, stop bits, and data bits. The UART settings of the XBee can be configured physically using a shield with USB connection using the XCTU application, or locally and remotely via AT commands.

Figure 10: Xbee PRO S1 Wireless module

The operating mode of an XBee radio module establishes the way that any device attached to the XBee communicates with the module through the UART or serial interface. Radio modules can work in three different operating modes:

Application Transparent (AT) operating mode (0)
API operating mode (1)
API escaped operating mode (2)

Transparent mode has limited functionality, simply allowing the radio to pass information along exactly as it receives it. In API mode, a protocol determines the way information is exchanged, and data is communicated in packets (API frames). This mode adds an additional header to the frame, including extra information that allows identifying the origin and destination nodes of a packet. This is useful to create sensor networks and perform tasks such as collecting data from multiple locations or controlling devices remotely.

These API frames are used to communicate with the module or with other modules in the network, and they have the following structure:

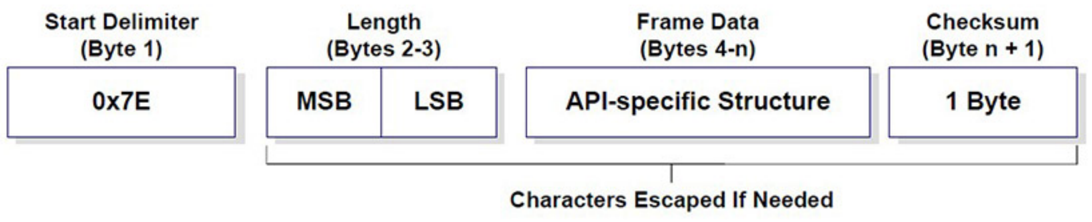


Figure 11: API frame structure

API 2 is the mode we use in this project, as we want to create sensor networks and be able to control the devices without having physical access to them. The only difference between API 1 and API 2 is that API 2 requires that frames use escape characters (bytes).

The module's radio firmware can also be modified using XCTU. New radio firmware versions are periodically released to fix bugs or improve functionality. All of our Xbee modules have been updated with the latest firmware version, 10EF.

While the Xbee modules have a long list of configurable settings, the specific ones needed to identify network nodes and form the WSN are the following:

Channel (CH): This controls the frequency band used by Xbee to communicate. XBee S1 802.15.4 operates on the 2.4GHZ 802.15.4 band, and the channel further calibrates the operating frequency within that band. Every XBee in the same network must operate on the same channel.
PAN ID (ID). The personal area network ID is an hexadecimal value between 0 and 0xFFFF. In order to communicate with each other, Xbees must have the same PANID. It is important to change it from the default to avoid accidentally operating on the same network as other Xbees in range.
Destination Address (DH, DL): This parameter (consisting of two values - Destination Address High and Destination Address Low) defines the Xbee module to which the current Xbee will send the data packets. This can be set to the destination's MY address or to its Serial Number. The destination address can also be set to 0xFFFF to enable broadcast

transmission, however, transfer rate drops significantly with this option enabled.

MY address (MY): This parameter defines the module's address and can be any value between 0x0000 and 0xFFFF. This address doesn't have to be unique - having two or more modules with the same address allows other nodes to send packets to all of them, specifying their shared address in the DH,DL parameters.

Coordinator Enable (CE): This parameter allows us to change the role of the Xbee module between Coordinator or End Device. Coordinator is the central node of a network - it syncs the network, establishes the channel frequency and allows other nodes to connect. It must always be available and thus can't enter sleep mode. End Devices on the other hand are remote nodes that can connect to the coordinator and other nodes withing the network. The end device's parameters can be changed remotely from the Coordinator once it joins the network.

In this project we create two small Xbee sensor networks, consisting of 1 Gateway and 1 *Sensor Station*, each of them equipped with an Xbee module. The Sensor Stations only send data to their assigned gateway, so the Xbee modules on the Station/Gateway pairs must have the correct Channel, PANID, Destination Address and MY Address to communicate between them. The Xbees on the Sensor Stations are configured as End Devices, while the ones on the Gateways are Coordinators. With this initial setup, more end devices could be easily added to each network, creating a star network topology.

Another important specification in Xbees - and, generally, in IoT nodes - is the communication range. While the Xbees' outdoor range is quite high, their indoor range is drastically reduced. Taking into account that the nodes are distributed in a building with multiple floors, the amount of walls and other obstacles between the nodes reduces the range enough to cause problems. To solve this, each Xbee node is connected to a RP-SMA 2.1 dBi antenna that significantly boosts the range. In Table 2 we can see an approximate calculation of the final ranges.

Without antenna	
Indoor/urban range	Up to 60 m
Outdoor RF line-of-sight range	Up to 750 m
With 2.1 dBi antenna	
Indoor/urban range	Up to 220 m
Outdoor RF line-of-sight range	Up to 2700 m

Table 2: Xbee PRO S1 range estimations

3.2.3 Raspberry Pi 3 Model B

Raspberry Pi is a series of low-cost, single-board computers developed by the Raspberry Pi Foundation [44]. It is widely used for IoT solutions since it offers a complete Linux server in a small platform.

In this project, the Raspberry Pi 3 Model B computers act as gateways between the sensor network and the Internet, allowing us to store and manage all the collected information.

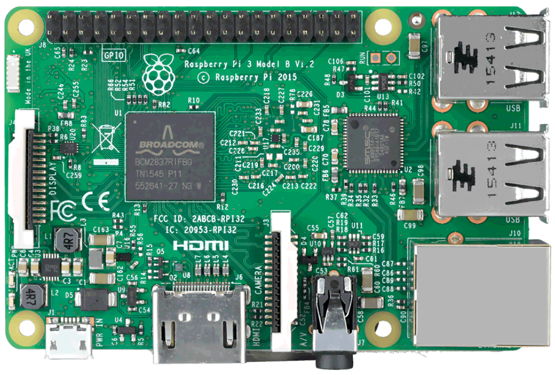


Figure 12: Raspberry Pi 3 Model B

The specifications for this model are as follows:

- 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2
- Gigabit Ethernet over USB 2.0
- 4 USB 2.0 ports
- Micro SD port
- 5V/2.5A DC power input

The USB ports provide a simple way to connect the Waspmote Gateways and receive the frames, which the Raspberry immediately processes. Once the sensor data is extracted, it is stored in a database for later use. Two of the available Raspberrys act as Gateways for our Xbee networks.

The third Raspberry works as gateway for the XM1000 mote network. The data retrieved from this network is collected by the Base Station, an XM1000 mote that acts as a gateway for the other two. This mote is connected to the Raspberry, where the data is processed and stored.

3.2.4 XM1000

The XM1000 is an advanced mote manufactured by AdvanticSys which contains integrated temperature, humidity and light sensors. Its main specifications are:

- IEEE 802.15.4 WSN mote
- TI MSP430F2618 Microcontroller, CC2420 RF [45]
- Compatible with TinyOS 2.x and ContikiOS
- User and Reset Buttons
- 3 LEDs
- USB Interface
- 3V (Using 2xAA batteries)

And the sensors' specifications are:

- **Light 1 - Hamamutsu S1087 Series:** Visible Range (560 nm peak sensitivity wavelength)
- **Light 2 - Hamamutsu S1087 Series:** Visible and Infrared Range (960 nm peak sensitivity wavelength)
- **Temperature and Humidity - Sensirion SHT11:**

  Temperature Range: -40 to 123.8 C
  Humidity Range: 0 to 100% RH

It supports TinyOS and ContikiOS, the most commonly used Operating Systems for nodes inside Wireless Sensor Networks. It is also based on TelosB specification, which makes it compatible with software available for that platform with just a few changes.

These were the physical motes used along the virtual ones in the theoretical design, and in this project they have been implemented using the specification that was described in the thesis, with few modifications. Making use of these motes' USB and antenna communication capabilities, one of them is set up as a gateway while the rest work as transmitters using 802.15.4 communication. The gateway interconnects the other two motes, collects their sensor data and forwards it to the Raspberry to which it is connected.

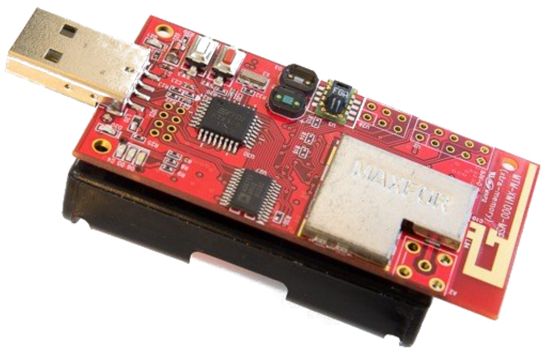


Figure 13: XM1000 mote

The motes are flashed with the provided sampling software [29] using TinyOS. This software samples the sensor data every *t* seconds and sends it via radio messages, which are then received by the gateway mote and forwarded.

The transmitter motes are powered by AA batteries which allows us to place them anywhere as long as the range reaches the gateway mote. Taking this into account, the provided software already implements a low power consumption model, to extend the motes' autonomy and lifespan.

4 Implementation

This section aims to explain in detail the process followed in order to implement the design in a real environment. Furthermore, this could also serve as a guide to recreate this implementation elsewhere, provided that we have the necessary hardware (which has been described in Section 3). It describes the steps to program the hardware, set up motes, gateways and databases; and place the devices in the building. It also includes descriptions of the software developed for this project. The code written for this project can be found on a GitHub repository [46].

4.1 Programming the Waspmote sensor boards

The Waspmote IDE is used to write and upload code to the boards. The Waspmote API is used in conjunction with the IDE to make use of the official libraries provided by Libelium.

4.1.1 Events board

This board is equipped with temperature, light, and presence sensors.

```
#include <WaspSensorEvent_v20.h>
#include <WaspXBee802.h>
#include <WaspFrame.h>

char RX_ADDRESS[] = "0013A200412539CE";
char WASPMOTE_ID[] = "SensorStation_2";

float ldr_value, ldr_sent, ldr_mapped;
float temperature_value, temperature_sent;
float presence_value, presence_sent;
ldr_sent = temperature_sent = presence_sent = 0;

float c1=10;
float c2=19.5;
float diff_ldr=80;
float diff_temp=1.0;
```

At the beginning, the necessary libraries are included. In this case we included the libraries for the Events Sensor Board, the Xbee module, and the Waspmote frames; given that we need to sense data, pack it into a frame and send it over to another Xbee.

Next, the variables are declared. *RX_ADDRESS[]* contains the MAC address of the destination Xbee module, which belongs to a gateway. *WASPMOTE_ID[]* contains the mote's ID, which is sent in the data frame and is used later to identify which values came from each sensor station. The different float variables for each sensor are used in the main program to store the current value and compare it to the previously sent one. *c1* And *c2* are coefficients used to adjust the raw readings of the sensors. The *diff* variables are used in the evaluation of current values against the previous ones.

```
void setup()
{
  frame.setID( WASPMOTE_ID );

  USB.ON();
  SensorEventv20.ON();
}
```

The *frame.setID()* function stores the Waspmote ID in the EEPROM memory, in order to set the corresponding field in the frame's header when the frame is created. Next, the Events sensor board and USB are initialized using the *ON()* function.

```
float map(long x, long in_min, long in_max, long out_min, long out_max)
{
  float value = (x-in_min) * (out_max-out_min) / (in_max-in_min) + out_min;
  if (value>out_max) return out_max;
  if (value<out_min) return out_min;
  else return value;
}
```

The *map* function re-maps a number from one range to another, and is used to adapt the sensor values (which have different ranges depending on the board) to a homogenized range.

```
void loop()
{
  USB.println(F("Entering sleep mode"));
  PWR.deepSleep(00:00:00:10, RTC_OFFSET, RTC_ALM1_MODE2, SENS_OFF);
  USB.ON();
  USB.println(F("Woke up"));

  RTC.getTime();
  (...)
```

Here, the main program begins. Using the *PWR.deepSleep* function, the Waspmote is put in deep sleep mode for 10 seconds. The parameters are Time, Offset, Mode, and Sleep Option respectively [47].
RTC_OFFSET adds the input time to the current time in the RTC and the result is set as the alarm time.
RTC_ALM1_MODE2 uses date, hours, minutes and seconds to match with the alarm.
ALL_OFF switches off the switches related to the sensor board.
Once the Waspmote wakes up by the RTC interruption, it initializes the USB and RTC and gets the current time.

```
(...)
ldr_value = SensorEventv20.readValue(SENS_SOCKET1, SENS_RESISTIVE);
temperature_value = SensorEventv20.readValue(SENS_SOCKET5);
presence_value = SensorEventv20.readValue(SENS_SOCKET7);

temperature_value = c1 + c2 * temperature_value;

ldr_value = (1.5 * ldr_value * 6250) / 4096;
ldr_mapped = map(ldr_value, 35, 0, 0, 1600);
(...)
```

The values of the luminosity, temperature and presence sensors are read using the Events sensor board's *SensorEventv20.readValue* function and stored in their respective variables. The *SENS_SOCKET* parameter allows us to select the sockets corresponding to each sensor, from which the values will be read. *SENS_RESISTIVE* is a special parameter for resistive sensors that returns the resistance of the sensor in kΩ.

Reading the sensor values directly from the board doesn't provide correct results, as they need to be converted. The temperature value must be adjusted using the following formula:

$$Temp = c1 + c2 * RawVal$$

The temperature coefficients c1 and c2 are found in the MCP9700A sensor's documentation [40].

The luminosity sensor value must be converted and then mapped to a homogeneous range that is used on every sensor. To convert the raw value, we use the following formula:

$$Luminosity = (1.5 * RawLDR * 6250)/(4096)$$

```
if (((abs(ldr_mapped - ldr_sent)) >= diff_ldr) or
((abs(temperature_value - temperature_sent)) >= diff_temp) or
(presence_value != presence_sent))
{
    xbee802.ON();

    frame.createFrame(ASCII);

    frame.addSensor(SENSOR_LUM, ldr_mapped);
    frame.addSensor(SENSOR_TCA, temperature_value);
    frame.addSensor(SENSOR_PIR, presence_value);
    frame.addSensor(SENSOR_TIME, RTC.hour, RTC.minute, RTC.second );

    error = xbee802.send( RX_ADDRESS, frame.buffer, frame.length );
   }
  }

  ldr_sent = ldr_mapped;
  temperature_sent = temperature_value;
  presence_sent = presence_value;
}
```

Once the measurements are done, this *if* statement evaluates whether the collected values have changed significantly since the last time they were sent. It compares the *diff_ldr* and *diff_temp* variables to the difference between both values, and proceeds if the result is equal or higher.

For the presence sensor, checking if these values are different is enough, given that it can only have "0" or "1" outputs.

If the program continues, the Xbee module is turned on (it was previously switched off by the deepSleep function). The *frame.createFrame* function creates an ASCII frame, and all the obtained sensor information is added to the frame using the *frame.addSensor* function. The current time is also added.

Finally, the current values are stored and the program loops to the beginning.

4.1.2 Smart Cities board

This board is equipped with temperature, humidity, and light sensors. Most part of the program is similar and uses the same functionalities as the Events sensor board, so some explanations have been skipped. Refer to the previous Section 4.1.1 for details.

```
#include <WaspSensorCities.h>
#include <WaspXBee802.h>
#include <WaspFrame.h>

char RX_ADDRESS[] = "0013A2004125398D";

char WASPMOTE_ID[] = "SensorStation_1";
(...)

void setup()
{
  frame.setID( WASPMOTE_ID );

  USB.ON();
}
(...)
```

The libraries included in this code are the ones for the Smart Cities sensor board, the Xbee module, and the Wapmote frames. *RX_ADDRESS[]* contains the MAC address of the destination Xbee module and *WASPMOTE_ID[]* contains this mote's ID. The *frame.setID()* function stores the Waspmote ID in the EEPROM memory and the USB is initialized.

```
void loop()
{
  USB.println(F("Entering sleep mode"));
  PWR.deepSleep(00:00:00:10, RTC_OFFSET, RTC_ALM1_MODE2, SENS_OFF);
  USB.ON();
  USB.println(F("Woke up"));
  RTC.ON();
  RTC.getTime();

  SensorCities.setSensorMode(SENS_ON, SENS_CITIES_LDR);
  SensorCities.setSensorMode(SENS_ON, SENS_CITIES_HUMIDITY);
  SensorCities.setSensorMode(SENS_ON, SENS_CITIES_TEMPERATURE);

  ldr_value = SensorCities.readValue(SENS_CITIES_LDR);
  humidity_value = SensorCities.readValue(SENS_CITIES_HUMIDITY);
  temperature_value = SensorCities.readValue(SENS_CITIES_TEMPERATURE);
  (...)
```

The main program begins and the Waspmote is put in deep sleep mode for 10 seconds. Once it wakes up, the necessary board sensors are initialized using the *SensorCities.setBoardMode* function for the light, humidity and temperature sensors. Next, the values are read using the *SensorCities.readValue* function provided by the Smart Cities board.

```
if (((abs(ldr_mapped - ldr_sent)) >= diff_ldr) or
((abs(temperature_value - temperature_sent)) >= diff_temp) or
((abs(humidity_value - humidity_sent)) >= diff_temp)) {
   xbee802.ON();

   frame.createFrame(ASCII);

   frame.addSensor(SENSOR_LUM, ldr_mapped);
   frame.addSensor(SENSOR_HUMA, humidity_value);
   frame.addSensor(SENSOR_TCA, temperature_value);
```

```
    xbee802.send( RX_ADDRESS, frame.buffer, frame.length );
  }
  (...)
```

If the values have changed significantly since the last update, the program continues. The Xbee module is switched on and the sensor values are added to the frame with the *frame.addSensor* function, before sending the frame. The program loops back to the beginning.

4.2 Creating a WSN using Xbee

The Xbees provide connectivity between the Waspmote Sensor Stations and the gateways, creating the sensor network. Xbees are especially useful in this regard - with the right configuration we can introduce new nodes that are automatically recognized and added to the network.

With the 4 Xbee modules we have, we construct 2 small sensor networks that work in identical ways. The Xbee connected to the Sensor Station transmits data to the other one, which is connected to the Gateway and forwards the received packets through serialport.

To begin with, every Xbee is attached to a Waspmote Gateway which functions as a shield to allow USB communication with the computer. This is only for configuration purposes, done by using XCTU [31] on every module. Once connected, the module can be selected on XCTU to adjust its parameters.

All the modules share some of the same configuration parameters:
Baudrate is set to 115200 and **Parity** to None, to be compatible with the serialPort communication.
API mode is set to API 2 (escaped operating mode).

The modules in the same network are configured with the same **Channel** and **PANID**. This places them in the same network, but they still can't talk to each other to exchange data packets. Each module is also assigned a **MY address**, and the **Destination Address** is set to the MY address of the other node, indicating where the packets should be sent and allowing data transmission. The specific configuration parameters used for each network can be seen in Tables 3 and 4.

Channel	11
PAN Id	1
End Device	DH: 0
	DL: 2
	MyAdd: 1
Coordinator	DH: 0
	DL: 1
	MyAdd: 2

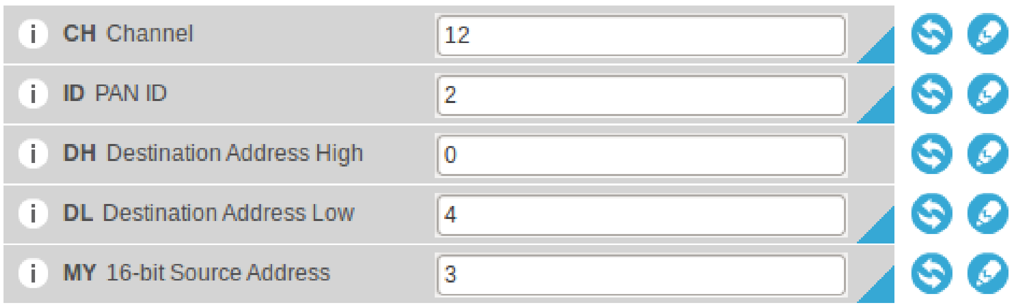
Table 3: Xbee configuration parameters - 1

Channel	12
PAN Id	2
End Device	DH: 0
	DL: 3
	MyAdd: 4
Coordinator	DH: 0
	DL: 4
	MyAdd: 3

Table 4: Xbee configuration parameters - 2

In addition, the Xbees connected to Sensor Stations are set as End Devices while the gateway Xbees are set as Coordinators. This affects sleep modes - the Coordinators can't enter sleep mode, so they won't accidentally stop receiving frames.

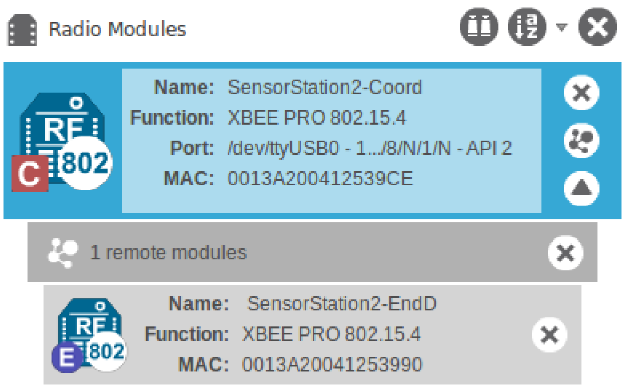
In the following figure we can see part of the Coordinator 2 configuration as an example:



CH Channel	12
ID PAN ID	2
DH Destination Address High	0
DL Destination Address Low	4
MY 16-bit Source Address	3

Figure 14: Coordinator 2 configuration

Once the paired modules are configured, XCTU lets us scan for radio nodes in the same network and, if found, they are added to the module list. This allows us to verify that the communication between the nodes is working, and additionally gives us access to modify the other network nodes configuration remotely.


Radio Modules

Name: SensorStation2-Coord
Function: XBEE PRO 802.15.4
Port: /dev/ttyUSB0 - 1.../8/N/1/N - API 2
MAC: 0013A200412539CE

1 remote modules

Name: SensorStation2-EndD
Function: XBEE PRO 802.15.4
MAC: 0013A20041253990

Figure 15: XCTU - List of local and remote nodes

Using the Console mode in XCTU is also helpful to check the communication between the nodes. It allows to send single data frames or sequences that, if received by the other node, are then echoed.

Finally, the Xbee modules configured as End Devices are directly attached to their corresponding Waspmote boards. The other two modules stay on the Waspmote Gateways, which are attached to the Raspberry boards via USB.

4.3 Flashing the XM1000 motes

To create the XM1000 sensor network, the XM1000 motes are flashed with the provided sampling software [29]. This software collects the sensor information and transmits it via TinyOS 802.15.4 Radio Messages to the rest of the system.

First, TinyOS needs to be installed, since it provides the platform and environment needed to flash the motes. However TinyOS doesn't offer support for XM1000 motes, requiring further manual configurations to enable it.
The driver and instructions provided by AdvanticSys are obsolete with current versions of TinyOS [48]. Instead, we use an alternative solution by adding the updated *XM1000_Platform* folder and files available in a GitHub repository [49], merging them with the TinyOS directory.

In addition to this, due to changes in the new Python versions, the function *setBaudrate()* used in a few of the XM1000 support files produces errors. The solution is to modify all occurences, changing *"self.serialport.setBaudrate(baudrate)"* to *"self.serialport.baudrate = baudrate"*.
After these changes, the environment is ready to work with XM1000 motes.

The begin the installation, the mote is connected via USB to the computer with the TinyOS environment, and the *motelist* command is executed on a terminal to list the available motes, with a similar output to:

```
Reference Device Description
--------- --------------- ---------------------------------------------
FTXWQT2G /dev/ttyUSB0 FTDI MTM-XM1000MSP
```

Which indicates that the XM1000 mote is connected to port */dev/ttyUSB0*.

From the XM1000 software directory, the program can be flashed into the mote with the following command:

```
./install.sh <#USB port>
```

Once flashed, the mote begins the booting process automatically, and will soon start collecting sensor data and transmitting it via radio messages.
The gateway mote, connected to the Raspberry gateway, gathers all these data messages and forwards them to the Raspberry via serial forwarder.

4.4 Creating and setting up the database

We decided to store the sensor information from all our devices in a cloud relational database. During the development of the project, this allowed us to compare the results of every sensor in real time and to keep track of problems like erroneous readings or anomalous behaviour without having to check every device separately.

The database has one table, *data*, with the following fields:

```
nodeid, time, temperature, humidity, light, presence
```

The *nodeid* field (varchar) serves to identify the device from which the sensor values were collected. Every device, both Waspmote Sensor Stations and XM1000 motes, has a unique nodeid that also makes it easier to identify problems. The *time* field (datetime) keeps track of when the sensor data was collected. This is used to see the sensor values' evolution over time and across different days. The *temperature, humidity, light and presence* fields (float) store the sensor values from different devices. The types of sensors vary between devices, and in case a device doesn't have a particular type, the field is NULL. The values are processed by a script that runs continuously on the gateways, and inserts values in the database as they are received. This code can be found on our GitHub repository.

4.5 Setting up the Raspberry gateways

The first step to configure our Raspberry boards is to install the operative system, for which we chose the standard Raspbian OS [33] as it provides everything we need for the project.

SSH is enabled so that we can access the board and configure it in headless mode - without using a monitor or keyboard - and from another computer. The board is then connected to a power outlet and to the Internet via Ethernet cable.

Next, to setup a Raspberry for the Waspmote sensor network, the Xbee module (attached to a Waspmote Gateway) is connected to the Raspberry via USB port and will automatically forward the radio messages it receives through it.
To insert the values, the Raspberry uses a Python script that runs indefinitely and listens on the serialPort for forwarded messages. Once received, they are parsed to extract the *nodeId, time* and sensor readings values from the *data* part of the API frame. The data frames include extra header bytes that need to be decoded to a readable format before processing them.

```
def process_frame(frame):
    data = frame['rf_data']
    data = data.decode("ISO-8859-1")
```

Once this is done, the values are extracted and then inserted into the *data* table inside the database.

```
(...)
sql_query = "INSERT INTO data (nodeid, time, temperature, light, presence)
    VALUES (%s, %s, %s, %s, %s)"
    val = (NODEID, TIME, TEMP, LUMI, PIR)
    cursor.execute(sql_query, val)

    mydb.commit()
(...)
```

This script requires the *xbee* and *pySerial* python libraries to read the Xbee frames, and the *mysql* python library to connect to the database and insert values. These libraries only need to installed once on the Raspberry, during the initial setup.
The script is set to execute automatically on startup and run on the background, by adding:

```
python3 /home/pi/listenSerialXbee.py &
```

to the /etc/rc.local file. This automatic execution means that the Raspberry can be disconnected from power, moved and restarted without needing to repeat any of the setup.

To setup a Raspberry for the XM1000 mote network, we re-used the Physical Listener that is part of the Environmental Interface [50] designed to operate between the sensor network and ServIoTicy. While we couldn't use the platform or most of the Environmental Interface's functionalities, the Listener is still practical as it parses the data with a readable structure that facilitates inserting the values into the database. One of the only changes necessary was to convert the *timestamp* field, originally in miliseconds, to *datetime* which is the format the database uses.

4.6 Placing nodes and gateways

The last step in the deployment of the sensor network is to install the devices in the building. The distribution is decided taking into account coverage and range, so every floor has at least a sensor mote that transmits to its closest gateway.

Due to being battery-powered the motes can be installed anywhere, which allows to place them in optimal locations with few obstacles that could interfere with sensor readings. Light sensors are placed in areas with good access to the natural light coming from windows, and the presence sensor is placed in a corner facing the desk and the room's entrance door.

The Raspberry gateways need access to a power outlet and an Ethernet jack, which limits the amount of locations that can host them.

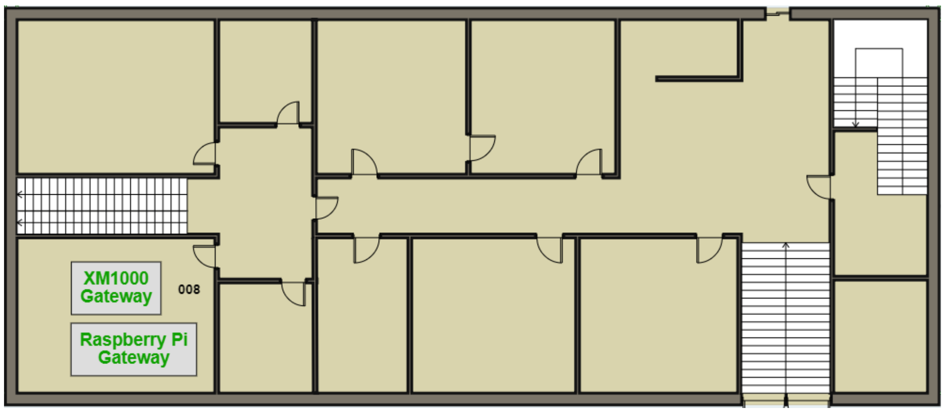


Figure 16: Floor 0 hardware placement

In floor 0, room 008 contains a Raspberry and the XM1000 mote that acts as a gateway. Given that the XM1000 motes have a shorter range, the transmitter motes in the other floors have been placed in the same side of the building and as close as possible to the location of room 008.

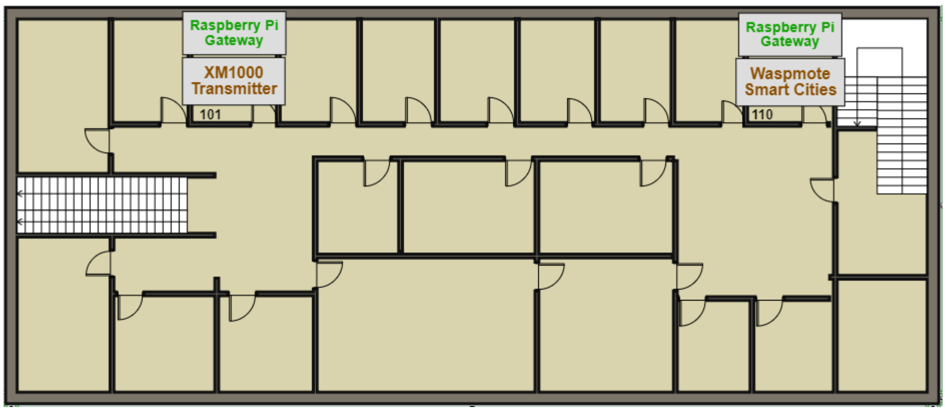


Figure 17: Floor 1 hardware placement

In floor 1 there are two office rooms, each with a Raspberry gateway and a mote. Room 101 contains an XM1000 transmitter mote, while room 110 contains the Waspmote Smart Cities board.

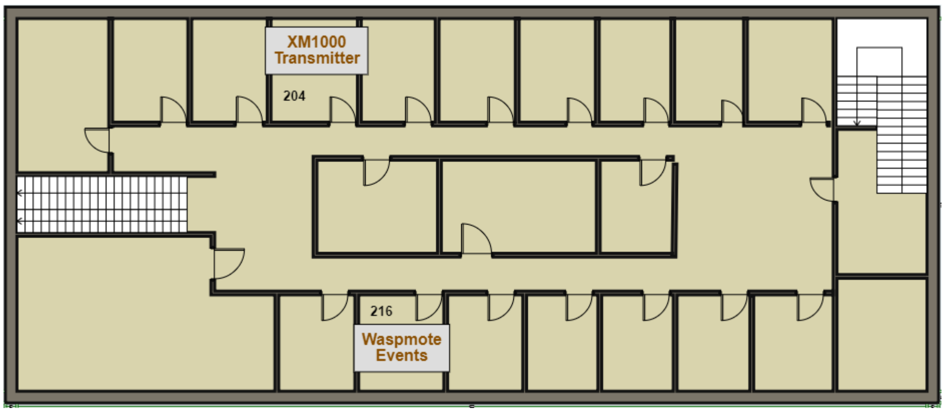


Figure 18: Floor 2 hardware placement

Finally, the remaining 2 motes are placed in office rooms in floor 2. Room 204 contains an XM1000 transmitter mote and room 216 has the Waspmote Events board.

5 Results

This section presents the results obtained from the sensor readings across every mote in the network after gathering data during 2 weeks in the D6 building.

The results are compared between them, separating them based on which type of mote provided the data, to find possible discrepancies and contrast their accuracy. Additional comparisons are made linking the evolution of the sensor results and their respective outdoors environmental conditions.

5.1 Data obtained

This section shows the data collected from every sensor by type, and separates and compares the results from the different sensor models in the network. In addition, some of the abnormal data and errors encountered while gathering the results are shown.

5.1.1 Light

Figure 19 shows the average luminosity values read by the XM1000 motes and their evolution through the day:

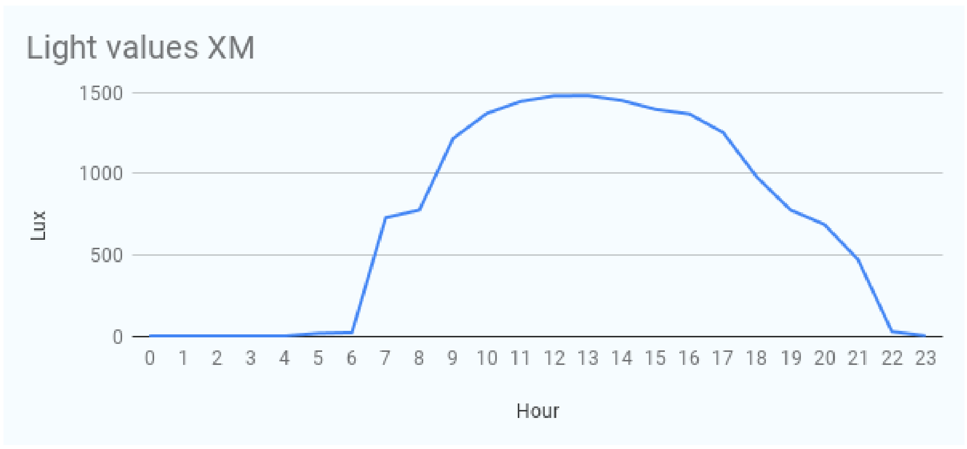


Figure 19: Luminosity values - XM1000

Figure 20 shows the average luminosity values read by the Waspmote sensor boards, equipped with the LDR sensor:

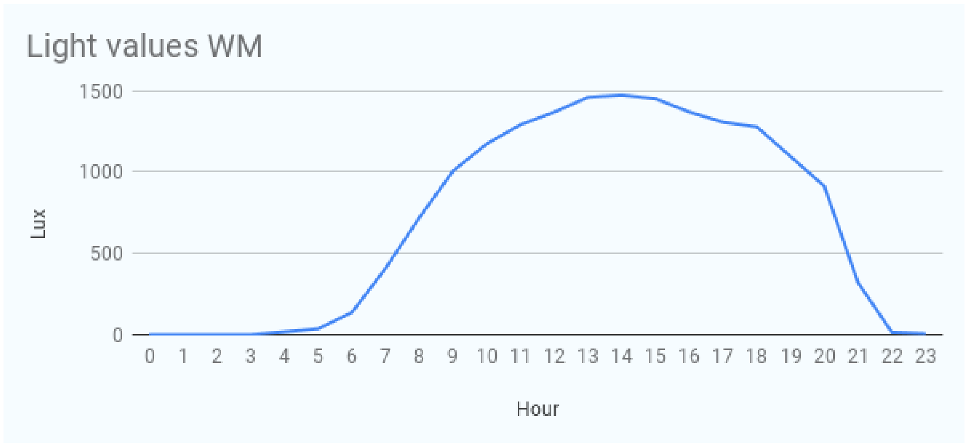


Figure 20: Luminosity values - Waspmote

There is no clear difference between the values read by each type of sensor, and they have the same peak light intensity and amount of daylight hours. Contrasting the sensor values with outside luminosity levels is difficult, given that there is no accessible way of obtaining accurate, objective levels of light.

One of the factors that can be compared is the period in which the light starts increasing in the morning and decreasing in the afternoon, as they are tied to sunrise and sunset times. The average sunrise in Barcelona for the month of June is at 6:17 and the sun sets around 21:26 [51], which indeed coincides with the graph's values.

5.1.2 Temperature

Figure 21 shows the evolution over time of temperature values read by the XM1000 motes:

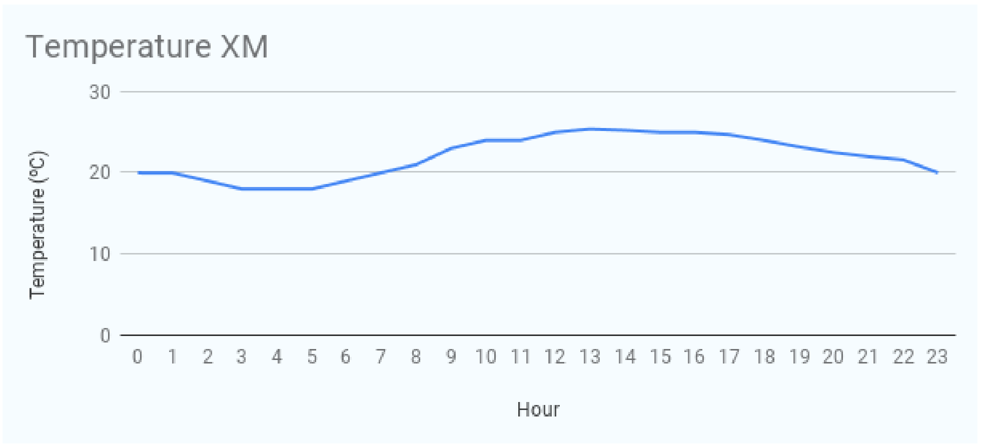


Figure 21: Temperature values - XM1000

Figure 22 shows the evolution over time of temperature values read by the Waspmote sensor boards, using sensor MCP9700A:

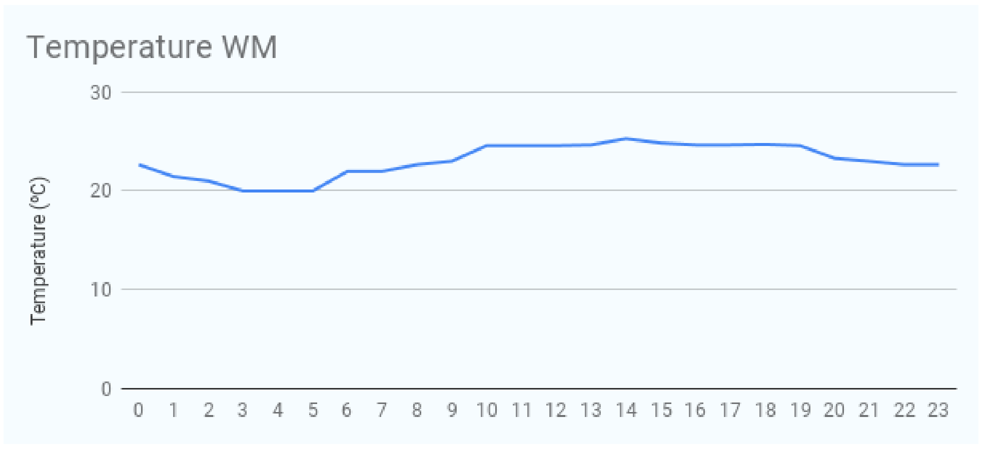


Figure 22: Temperature values - Waspmote

The values are decidedly similar between the two types of motes. Contrasting them with the recorded temperature values for the same month and location shows a link with the real environmental temperature [52], both in values and progression over time:

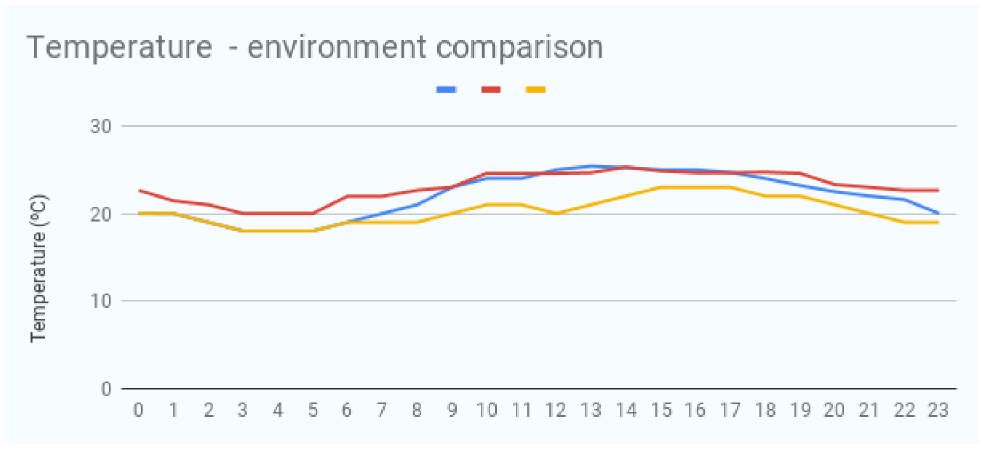


Figure 23: Temperature values contrasted with environmental temperature

However, the Waspmote's readings are more erratic overall, with sudden drops and abnormally high values that had to be removed from the extracted data to make the results accurate. While this doesn't happen regularly enough to significantly affect the data's usability, the irregular values should be filtered out to provide a more correct result. Some of these occurrences can be seen in the following figures:

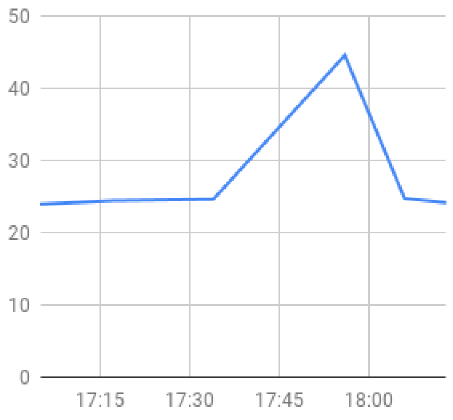


Figure 24: Abnormal values

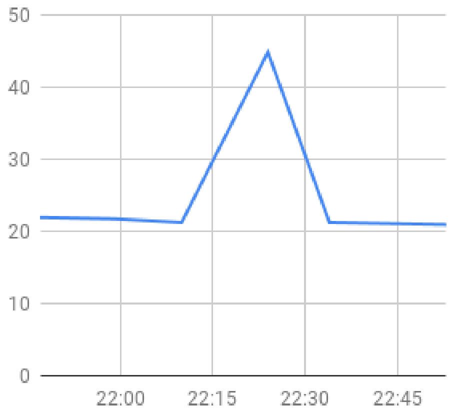


Figure 25: Abnormal values

This could be caused by a variety of issues like a poor socket connection, moisture getting trapped in the sensor, damage, or the sensor being accidentally moved or touched. Given that the problem occurred in both MCP9700A sensors, it can be attributed to sensitivity problems with this sensor model in particular.

5.1.3 Presence

Figure 26 represents the values detected by the presence sensor through the day, where *"1" = presence* and *"0" = no presence*. Each color represents a different day over a total of 5.

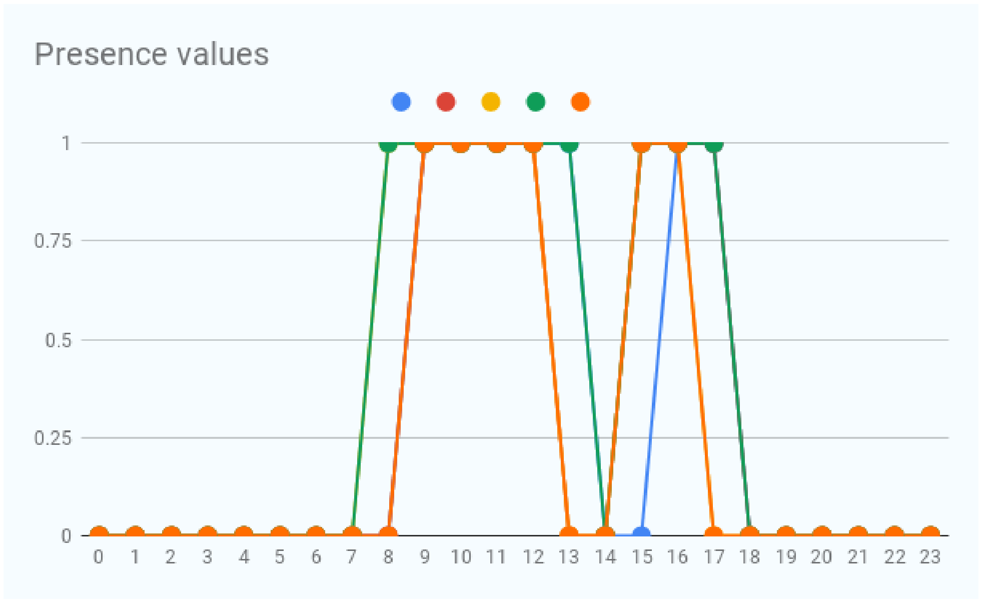


Figure 26: Office room presence values

Many values overlap due to the same professors usually having similar schedules and habits. There are a few observations to be made from this:
There is a distinctive lack of presence every day between approximately 12 PM and 15 PM, which can be attributed to lunch breaks.
The first detection of presence is usually between 7 AM and 9 AM.
The last detection of presence is a more variable number, but it can still be perceived as being between 3 PM and 6 PM.

5.2 Comparisons

This section aims to compare the results of the experiment to the models used for the simulations, and find the possible divergences between them. For these comparisons, the important factor to notes isn't a strict value correlation, but similar progression over the same period of time.

5.2.1 Light

Figure 27 shows the evolution of luminosity inside the building according to the theoretical models:

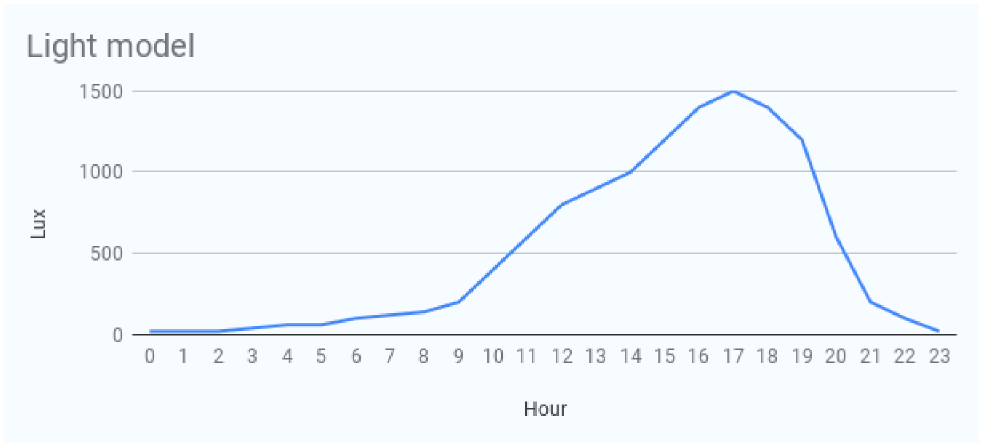


Figure 27: Luminosity model

If compared to the results from this project's implementation, obtained by averaging the readings of the 5 light sensors used, there are some clear differences.

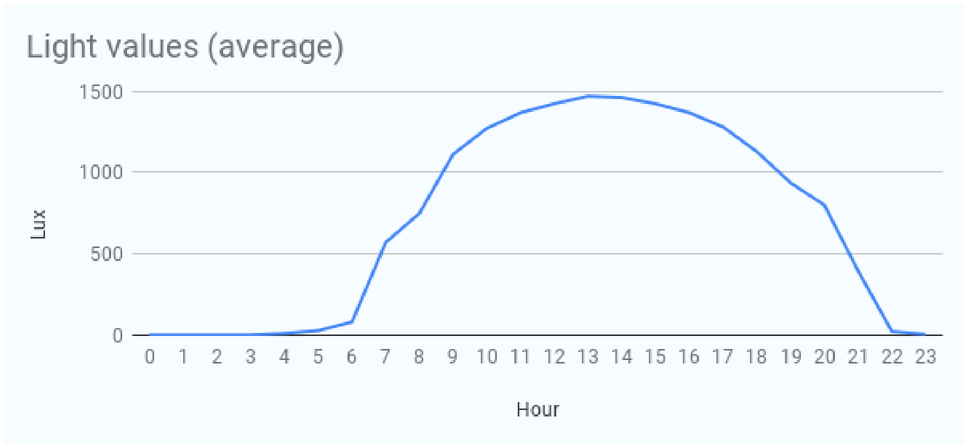


Figure 28: Luminosity values - all sensors

The light level is similar during the night (10 PM to 5 PM) but develops differently during the day, both regarding total daylight hours and intensity. In the case of real sensor data, light levels begin to increase at 6 AM and peak at midday, then start slowly decreasing until just past 9 PM. In the case of the models, light levels start rising between 8 AM and 9 AM, peak around 4 PM, and then quickly drop.

This however isn't a clear indication that either the models or sensor readings are incorrect. While erroneous sensor readings are possible, it's unlikely that this is the reason behind the discrepancy, given that every light sensor implemented provided very similar results regardless of their model.

The first factor that can affect the luminosity readings is the location of the sensors. They are placed indoors and receive both natural and artificial light, from windows and from the room's lights. This can explain, for example, the unusually high levels in the early morning and afternoon which could be caused by the lights being turned on by the occupants once daylight isn't enough.

The second factor is the time of the year. This project was developed during the spring semester and most of the sensor readings were collected during June, which has longer daylight hours compared to most of the year. While we don't know the exact timing used for the models, it is reasonable to estimate that they were obtained in a time of the year with less daylight.

5.2.2 Temperature

Figure 31 shows the daily evolution of temperature according to the models:

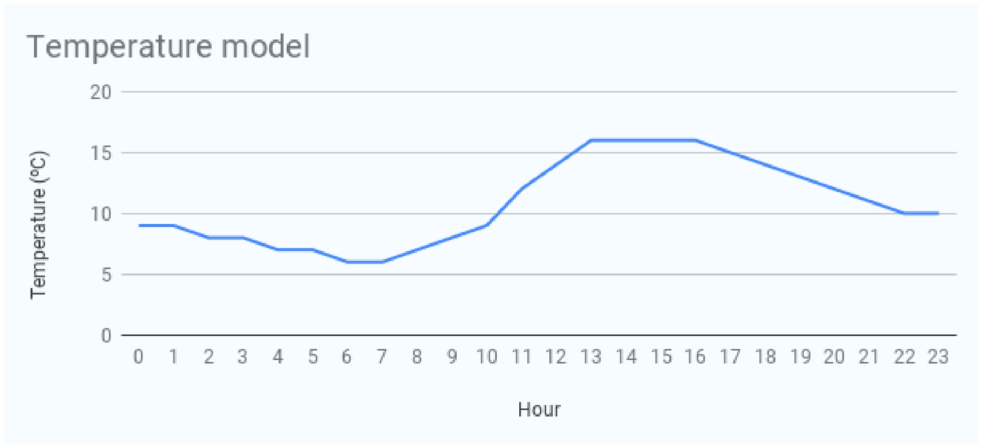


Figure 29: Temperature model

Compared to the following graph, constructed from averaging the data that was collected by the 5 temperature sensors over a period of two weeks, it is clear that the temperature values sensed during this project are higher overall, approximately by a factor of 2.

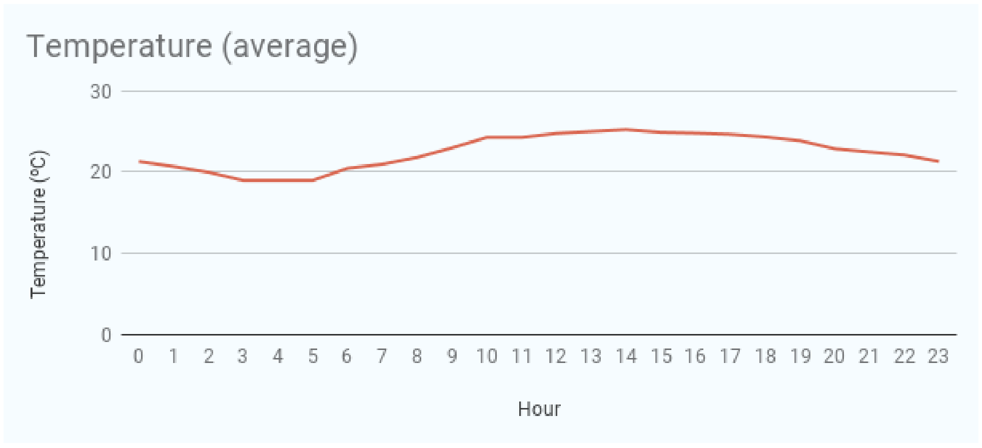


Figure 30: Temperature values - all sensors

In this case, it is justifiable to attribute this contrast to the different time of the year and weather conditions in which the model and the sensor values were obtained. However, the goal of the comparisons isn't to find identical or very similar values, as temperature fluctuates widely throughout the year and the system should be inherently prepared to deal with that. Instead, a similar evolution over the same period of time is a better indicator of the models' accuracy.

Juxtaposing the two figures shows that while the values are very different, the evolution over time does follow the same pattern, with the highest values appearing between 12 PM and 17 PM.

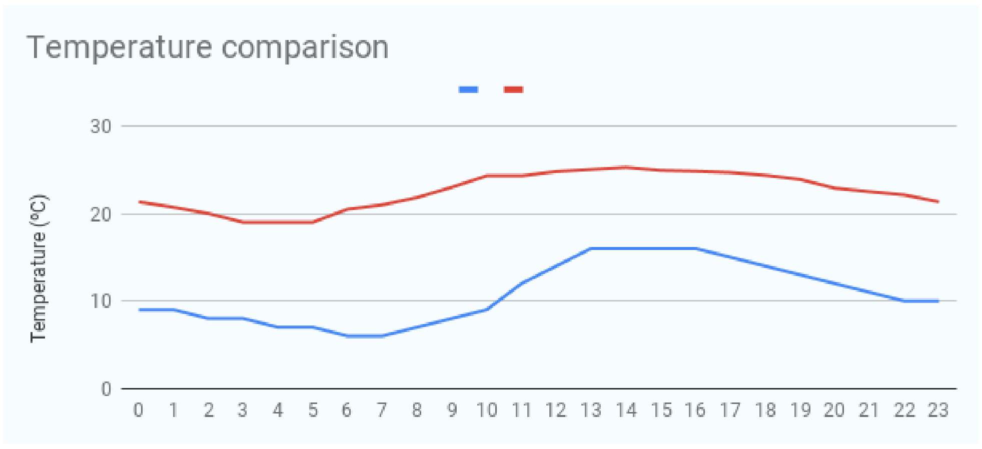


Figure 31: Temperature values - model vs. sensors

This lets us conclude that the temperature models were fairly accurate to the real environmental temperature inside the building.

5.2.3 Presence

The simulations used probability models to predict when the occupants would enter the building, take breaks or lunch, and exit the building. These occupants were separated depending on their profiles, belonging to either the *Professor*, *User* or *PAS* groups. In this project we are limited to one presence sensor, so our scope is reduced to one room and one type of user. The sensor is located in office 216 (see Figure 18), so the results are compared to the *Professor* group's probability models.

The following charts are taken directly from the published results [1] and indicate what probabilities there are of each action being performed over time:

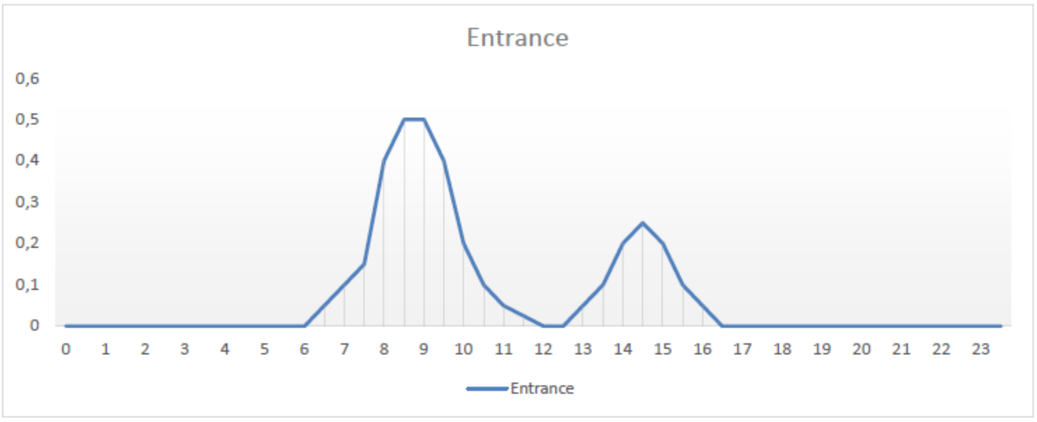


Figure 32: Professor entrance probability

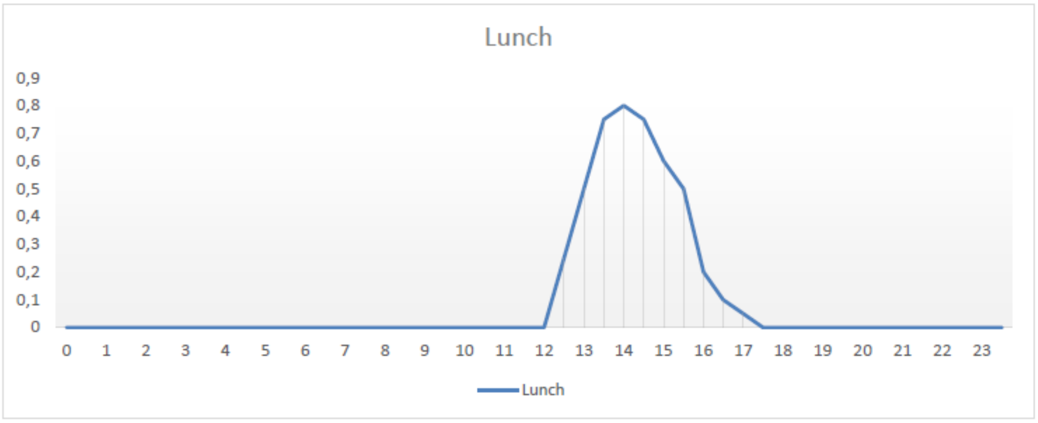


Figure 33: Professor lunch probability

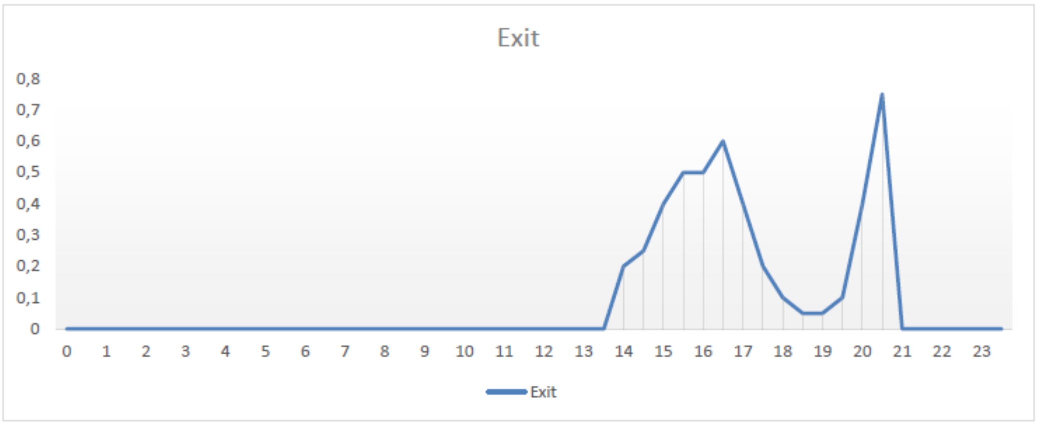


Figure 34: Professor exit probability

These graphs are compared against the average presence values obtained in this project, shown in Figure 35.

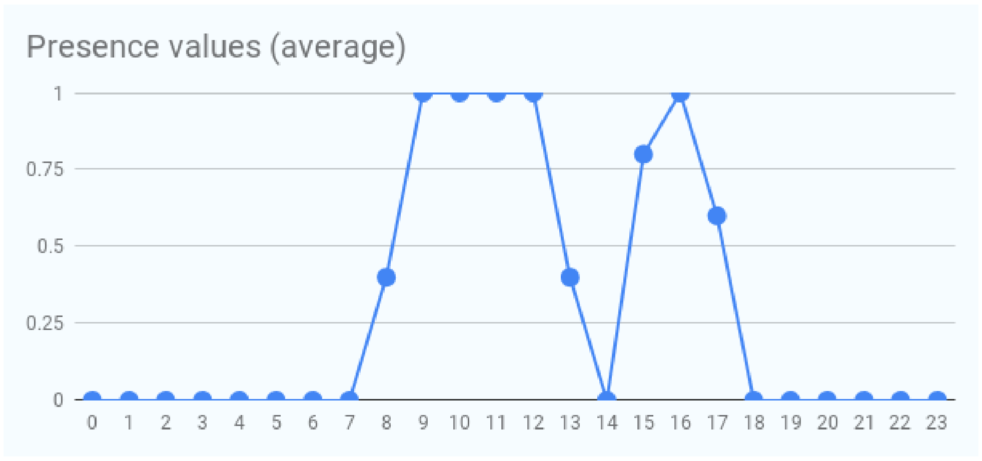


Figure 35: Office room presence values - 5 day average

As it can be seen, the first time the sensor detects presence during the day is usually between 8 AM and 10 AM, which aligns with the morning entrance probability in Figure 32. The afternoon entrance time, between 3 PM and 5 PM, also corresponds with the predictions.

The lunch break probability model predicts that lunch breaks are usually taken between 1 PM and 3 PM, and this indeed corresponds to a lack of presence detection in the sensor results, which have a negative peak at 2 PM.

Finally, the building exit probability indicates that most professors leave between 4 PM and 6 PM, with another peak of probability at 8 PM. The sensor readings corroborate the first prediction, having the last positive presence reading just after 5 PM, but don't show corresponding data for the 8 PM prediction.

6 Conclusions

The development and deployment of this project allows us to extract some useful conclusions, both regarding the current installation and the prospect of deploying a full automatic actuation platform as described in the original thesis. This section also presents suggestions and improvements in regards to the obstacles found during the project, which could be implemented in future work.

6.1 About sensor types

In the experiment, the XM1000 and Waspmote sensors ended up returning similar data in regards to temperature and luminosity levels. Contrasting these results to real data from the outdoors environment we can see that the accuracy was significant, and conclude that the deployed sensor network does provide authentic real time information about the current environmental values inside the D6 building, thus fulfilling the main goal of the project.

There were some notable differences between the behavior of the sensors in regards to other aspects, however. The Waspmote sensors experienced significantly more cases of erratic and unusual sensor readings, the causes of which couldn't be exactly diagnosed. In contrast, the XM1000 sensors were more consistent overall and while some wrong readings were found, they could mostly be attributed to the weak communication between the motes.

Continuing with the aspects of communication and range, the Waspmote sensor network proved to be significantly better. The range was never a problem, as every node could detect the other ones no matter where they were placed, and data was consistently being received from all sensors, indicating that the data packets had no trouble reaching their gateway. This comparison however should note that the Xbee radio modules used were attached to an antenna, which the XM1000 motes didn't have. Without it the range is similar to that of XM1000 motes and could have caused the same problems. On this topic we can conclude that it is recommended to have a wider communication range than necessary when deploying a sensor network indoors, given that even if technically the range is enough to cover the area, there are too many obstacles that can interfere with communication.

Regarding complexity, the Waspmote network was simpler to deploy. The available tools for Xbee provide an easy way of configuring the network and adapt it to our necessities. The Waspmote boards and the provided software are everything we need to manage mote resources and sensors and get exactly the behavior we want out of them. In comparison the XM1000 motes demanded a more cumbersome setup, caused in part by outdated software (both TinyOS and the XM1000 drivers) that required more research and manual adjustments in order to work in our computer. As a conclusion, and taking into account the price differences (see Section 7.1), we would recommend using mostly XM1000 sensors to obtain basic data about temperature and light, with some Waspmote boards for specific locations and sensors like presence, noise, or dust.

Having an heterogeneous system with different kinds of sensors required the development of different tools to process and store the data. We can conclude that a platform like ServIoTicy would greatly help in this regard, as the standardization of the messages would facilitate integrating new motes, both from existing types and new ones, with less hassle.

6.2 About comparisons with models

Overall the results obtained align with the ones from the models, meaning that the conclusions from the original thesis regarding energy efficiency in the building would presumably be similar in a real implementation of the platform.

The presence sensor provided pretty similar results for professors models, excepting the probability of random walks that our sensor didn't detect. Before being able to give a definitive conclusion we should perform the same study with the the other models, students and PAS. However the limited information we obtained does point to the simulated models being pretty similar to the real situation.

The temperature sensors also provided similar results, not taking into account the difference caused by weather conditions and different times of the year. It is pretty clear that automating the temperature of the building could provide significant energy savings based on the results from simulating its automation with the model values.

Although the light sensors gave us mostly similar results, we conclude that installing the light sensors indoors didn't provide very useful information overall. Implementing luminosity sensors inside the building could be useful once an automation system is implemented, but currently the lights can be turned on or off by anyone, causing interferences. From our results it is difficult to estimate when they should be automatically turned on or off, and trying to separate the values that come from artificial light from those that come from natural sources is too complex. This could be fixed by repeating this study during a period of time when the building in empty, for example during the month of August.

7 Budget and Planning

7.1 Budget

This section gives an estimation of the budget used to develop the project, taking into account the costs of hardware, software and human resources.

7.1.1 Human resources budget

The project is developed entirely by one person, who fills the roles of project manager, software developer, hardware engineer and tester.

Role	**Hours**	**€/hour**	**Salary**
Project manager	70	50,00€	3.500,00€
Software developer	190	40,00€	7.600,00€
Hardware engineer	130	40,00€	5.200,00€
Tester	30	30,00€	1.800,00€
Total	450	-	18.100,00€

Table 5: Human resources budget

7.1.2 Hardware budget

This budget includes all the hardware used in the implementation and the computer used to do some of the setup, configurations and programming. The amortization is calculated based on an usage time of 5 months.

Product	**Units**	**Unit price**	**Useful life**	**Amortization**
Computer	1	1000€	4 years	104,17€
Raspberry Pi 3	3	38€	5 years	3,16€
Waspmote PRO (v.1.2)	2	228€	4 years	23,75€
XBee PRO S1 802.15.4	4	36,90€	4 years	3,85€
Waspmote Events Sensor Board	1	55€	4 years	5,72€
Waspmote Gateway 802.15.4	2	57€	4 years	5,93€
AdvanticSys XM1000	3	95€	4 years	9,89€
Total	-	1.509,9€	-	156,47€

Table 6: Hardware budget

7.1.3 Software budget

This project is developed using free and open-source tools, so the software doesn't add any additional budget costs.

Product	Units	Unit price	Useful life	Amortization
GitHub	-	0€	-	0€
TinyOS	-	0€	-	0€
Waspmote IDE	-	0€	-	0€
Visual Studio Code	-	0€	-	0€
Etcher	-	0€	-	0€
Ubuntu 16.04	-	0€	-	0€
Raspbian	-	0€	-	0€
XCTU	-	0€	-	0€
Total	-	0€	-	0€

Table 7: Software budget

7.1.4 Indirect costs

This budget includes other costs that aren't directly related to the project itself, but still counted as part of the overall budget.

Product	Units	Price	Estimated cost
Internet	5 months	30€/month	150€
Electricity	1500 kWh	0,13€/kWh	195€
Transport	5 months	40€/month	160€
Total	-	-	505€

Table 8: Indirect costs

7.1.5 Total cost

The total estimated budget for the project is calculated by adding up the costs of each individual budget.

Area	Cost
Human resources	18.100,00€
Hardware resources	1.509,9€
Software resources	0€
Indirect costs	505€
Total	20.114,9€

Table 9: Total cost

7.2 Planning

This project has an approximate duration of 5 months, starting at the beginning of February and ending on July 1st with the final oral defense. In this section the overall planning and tasks completed to develop the project are described. Most tasks related to the actual implementation are only given a brief summary, since they have been thoroughly described in Section 4.

As this project is based on an existing thesis, the development began with a study phase during which the original material was reviewed and studied. This included looking for information on related topics like the Internet of Things, Wireless Sensor Networks and Smart Buildings; as well as reading through ServIoTicy's documentation.

The following phase was setting up the necessary environment and installing the required tools for developing the project. During this phase we tried to access the ServIoTicy server and found out about its problems, requiring us to shift our focus to that issue.

The following weeks were spent trying to install ServIoTicy locally. We tried to install it in a Vagrant box as well as directly from the available GitHub repositories. This task wasn't producing results and was delaying the rest of the project, so we decided to start working on the WSN deployment while we tried to find a solution on the side.

The development phase started with studying the available hardware and designing the overall architecture of the system. First the Waspmote technical guides were studied to understand how the different components are attached to each other and how they work. The boards were programmed and tested until their readings were as accurate and consistent as possible.

Next, the Xbee modules and communication capabilities were studied, after which they were configured to implement the Waspmote sensor network. After the Waspmote network started working, we studied the XM1000 motes and configured them to create their own network. During this time we concluded that finding a solution for ServIoTicy was not likely, and changed the focus and scope of the project.

Once the networks were ready to start transmitting, we designed and created the database and configured the Raspberry Pis as gateways, developing the necessary software to process the different radio messages and insert them into the database. After that the WSN was ready to be deployed.

Finally we installed the devices in the building and began testing, adjusting a few of the sensors to fix problems regarding energy consumption and erroneous readings. Once the testing finished, the sensor network was left to gather as much data as possible while only monitoring the results to keep track of problems and perform minor tweaks.

7.2.1 Gantt chart

The Gannt chart shows the described tasks, the time they took to complete and how they developed over time.

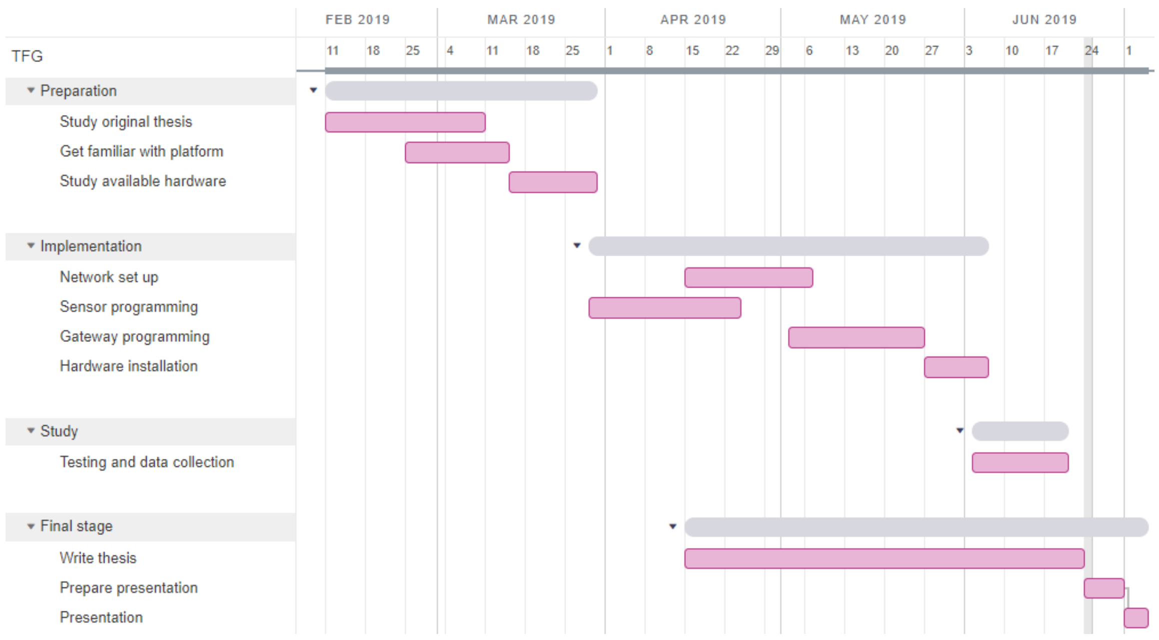


Figure 36: Gantt chart

8 Sustainability

This section analyzes the environmental, economical and social impact that this project can have.

8.1 Environmental

During this project, we constantly had computers, gateways, routers, sensors and at least a server running. We had to ensure that despite this, the results made this installation worthwhile, as energy efficiency was one of the main priorities regarding the design. We used very low consumption devices for the sensor network, and maximized energy savings by using low consumption modes whenever possible - every program was developed with energy waste in mind. The settings of the sensor boards were tweaked to find the minimal amount of energy consumption that would still give us reasonably precise and periodical sensor readings.

However we cannot neglect the amount of energy and materials spent on producing these devices, which is an indirect but unavoidable cost of using new technology. Some of these costs could be alleviated by buying second-hand devices, but this could also make their useful life shorter, which would end up lowering the efficiency of the project itself since the hardware would have to be replaced earlier than expected. For this reason, using brand new devices seems like the optimal choice, despite its environmental cost.

8.2 Economical

Section 7.1 details the estimated costs of the project, taking into account the cost of resources (hardware, software, and human), and also any costs not directly related. As it can be seen, the project's main cost is human resources with a budget of 18.100€.

While the hardware costs are listed as 1.509,9€, the devices were already available and were borrowed from the university, so no additional hardware has been bought specifically for the project. In addition, we have only used free software, which has also helped cut down costs. Thanks to this, the project hasn't gone over budget even after dealing with unexpected problems that made us increase the development hours used.

8.3 Social

This project can provide information about the deployment of sensor networks in similar buildings, which problems can arise and what considerations should be taken regarding sensor placement, autonomy and cost. It also contributes more data about the viability of the platform described in the original thesis and supports the idea, which could be a motivation for someone to decide to fully implement a similar platform.

On a personal level, it was rewarding to be able to develop a project based on a topic that is of great personal interest. Many skills learned in class were put to the test, and it helped to learn some new ones too, as well as discover new developments in IoT and how they could be applied to this and future personal projects.

9 Technical Competences

9.1 CTI1.4

***To select, design, deploy, integrate, evaluate, build, manage, exploit and maintain the hardware, software and network technologies, according to the adequate cost and quality parameters*.**

This was done during the entire project, which encompassed selecting the appropriate hardware from the products we had available, designing the architecture of the network, configuring its components, deploying the hardware, and testing the installation among other of the tasks described in Section 7.2. Once installed, the hardware and network were maintained and made to comply with the parameters of quality of the project, which included keeping a low energy consumption and obtaining the best accuracy from the sensors.

9.2 CTI2.1

To manage, plan and coordinate the management of the computers infrastructure: hardware, software, networks and communications.

As the only developer, we were tasked with the management of the entire infrastructure developed for this project. We had to coordinate every element and the communications between each other, as well as managing the connection to and from the gateways, and planning and setting up the development environment in our computer.

9.3 CTI3.3

To design, establish and configure networks and services.

To build the sensor network first we had to design it, then proceed to configure it according to specifications. After that it was established by placing all the devices around the building, which involved connecting and incorporating the motes, turning them on, connecting the gateways to the internet and power outlets and finally connecting to the database service to store the received values.

List of Figures

List of Tables

Glossary

I2C Inter-Integrated Circuit Bus.

IoT Internet of Things.

ISM Industrial, Scientific and Medical.

PANID Previous Access Network Identifier.

PIR Passive Infrared Sensor.

RIFD Radio Frequency Identification.

RTC Real Time Clock.

SPI Serial Peripheral Interface.

UART Universal Asynchronous Receiver-Transmitter.

WSN Wireless Sensor Network.

References

[1] David Sembroiz Ausejo. "Design and simulation of an interoperable IoT platform for automatic actuation in buildings". MA thesis. Universitat Politècnica de Catalunya, 2016.

[2] Internet of Things Global Standards Initiative. *Internet of Things Definition*. URL: `http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx`.

[3] Y Series. "Global Information Infrastructure, Internet Protocol Aspects and Next-Generation Networks". In: *ITU-T Recommendation Y* (2001).

[4] IoT Agenda. *What is internet of things (IoT)?* URL: `https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT`.

[5] M Keenan. *The Future of Data With the Rise of the IoT - RFID Journal*. URL: `https://www.rfidjournal.com/articles/view?17954`.

[6] Ericsson. *Internet of Things forecast – Ericsson Mobility Report*. URL: `https://www.ericsson.com/en/mobility-report/internet-of-things-forecast`.

[7] Cisco. *Seize New Opportunities with the Cisco IoT System*. URL: `http://www.cisco.com/web/solutions/trends/iot/portfolio.html`.

[8] We Are Social. *Global Digital Report 2019*. URL: `https://wearesocial.com/global-digital-report-2019`.

[9] Energy - European Commission. *Buildings - Energy - European Commission*. URL: `https://ec.europa.eu/energy/en/topics/energy-efficiency/energy-performance-of-buildings`.

[10] A Allouhi et al. "Energy consumption and efficiency in buildings: current status and future trends". In: *Journal of Cleaner production* 109 (2015), pp. 118–130.

[11] Michael Levin and Josuhua Lowitz. "Amazon Echo: What We Know Now (Updated)". In: *Chicago: Consumer Intelligence Research Partners* (2016).

[12] Shannon Liao. *Smart homes got fancier this year at CES*. Tech. rep. 2019. URL: `https://www.theverge.com/2019/1/12/18169125/smart-home-fancy-doorbells-locks-ces-2019`.

[13] *ServIoTicy source code*. URL: `https://github.com/serviotcy`.

[14] Eugeny I Batov. "The distinctive features of "smart" buildings". In: *Procedia Engineering* 111 (2015), pp. 103–107.

[15] Saber Talari et al. "A review of smart cities based on the internet of things concept". In: *Energies* 10.4 (2017), p. 421.

[16] Deborah Snoonian. "Smart buildings". In: *IEEE spectrum* 40.8 (2003), pp. 18–23.

[17] S. Khandavilli. *Case Study: Intel Creates Smart Building Using IoT*. URL: `https://www.intel.es/content/www/es/es/smart-buildings/smart-building-using-iot-case-study.html`.

[18] Intel. *Intel® Core™ i9-9980XE Extreme Edition Processor*. URL: https://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-9980xe.html.

[19] Thomas Weng and Yuvraj Agarwal. "From buildings to smart buildings—sensing and actuation to improve energy efficiency". In: *IEEE Design & Test of Computers* 29.4 (2012), pp. 36–44.

[20] Malik Tubaishat and Sanjay Madria. "Sensor networks: an overview". In: *IEEE potentials* 22.2 (2003), pp. 20–23.

[21] Anandbabu B, Siddaraju R, and Guru R. "Energy Efficiency Mechanisms in Wireless Sensor Networks: A Survey". In: *International Journal of Computer Applications* 139 (Apr. 2016), pp. 27–33. DOI: 10.5120/ijca2016908954.

[22] Xiaolin Jia et al. "RFID technology and its applications in Internet of Things (IoT)". In: *2012 2nd international conference on consumer electronics, communications and networks (CECNet)*. IEEE. 2012, pp. 1282–1285.

[23] Robin Heydon. *Bluetooth low energy: the developer's handbook*. Vol. 1. Prentice Hall Upper Saddle River, NJ, 2013.

[24] Paolo Baronti et al. "Wireless sensor networks: A survey on the state of the art and the 802.15. 4 and ZigBee standards". In: *Computer communications* 30.7 (2007), pp. 1655–1695.

[25] Dae-Man Han and Jae-Hyun Lim. "Smart home energy management system using IEEE 802.15. 4 and zigbee". In: *IEEE Transactions on Consumer Electronics* 56.3 (2010), pp. 1403–1410.

[26] Patrick Kinney et al. "Zigbee technology: Wireless control that simply works". In: *Communications design conference*. Vol. 2. 2003, pp. 1–7.

[27] *Digi International Inc.* URL: https://www.digi.com/about-digi.

[28] *TinyOS*. URL: https://github.com/tinyos/tinyos-mai.

[29] *TinyOS XM1000 Radio Sensing Software*. URL: https://github.com/DavidSembroiz/XM1000.

[30] *Etcher*. URL: https://www.balena.io/etcher/.

[31] *XCTU platform*. URL: https://www.digi.com/products/iot-platform/xctu.

[32] *Waspmote SDK and Applications*. URL: http://www.libelium.com/development/waspmote/sdk_applications/.

[33] *Raspbian*. URL: https://www.raspberrypi.org/downloads/raspbian/.

[34] *Visual Studio Code*. URL: https://code.visualstudio.com/.

[35] *Waspmote website*. URL: http://www.libelium.com/products/waspmote/.

[36] *Libelium website*. URL: http://www.libelium.com.

[37] *Waspmote technical guide*. URL: http://www.libelium.com/downloads/documentation/v12/waspmote_technical_guide.pdf.

[38] Libelium. *Smart Cities technical guide*. URL: http://www.libelium.com/downloads/documentation/v12/smart_cities_sensor_board.pdf.

[39] *Events 2.0 technical guide*. URL: http://www.libelium.com/downloads/documentation/events-sensor-board_2.0.pdf.

[40] Microchip. *MCP9700A documentation*. URL: http://ww1.microchip.com/downloads/en/devicedoc/20001942g.pdf.

[41] *808H5V5 documentation*. URL: https://www.tme.eu/Document/a0d04989cb8f2215a17d6476eac5f482/SENS-808H5V5.pdf.

[42] Gabriel Montenegro et al. *Transmission of IPv6 packets over IEEE 802.15. 4 networks*. Tech. rep. 2007.

[43] *XBee/XBee-PRO S1 802.15.4 User Guide*. URL: https://www.digi.com/resources/documentation/digidocs/PDFs/90000982.pdf.

[44] *Raspberry Pi Foundation*. URL: https://www.raspberrypi.org/about/.

[45] AdvanticSys. *TI MSP430F2618 Microcontroller Manual*. URL: http://www.advanticsys.com/shop/%20documents/1322654976_MSP430F2618.pdf.

[46] *Repository with code developed for the project*. URL: https://github.com/Cmaral/IoT-platform/.

[47] *Waspmote power Programming Guide*. URL: https://www.libelium.com/v11-files/documentation/waspmote/waspmote-power-programming_guide.pdf.

[48] *XM1000 TinyOS Platform Installation*. URL: https://www.advanticsys.com/wiki/index.php?title=XM1000_TinyOS%5C%C2%5C%AE_Platform_Installation.

[49] *XM1000$platform$*. URL: https://github.com/benlammel/Vagrant_TinyOS-2.1.2_msp430-47_XM1000/tree/master/tinyos-2.1.2/XM1000_Platform.

[50] *EnvironmentalInterface GitHub repository*. URL: https://github.com/DavidSembroiz/EnvironmentalInterface.

[51] *Sunrise and sunset times in Barcelona*. URL: https://www.timeanddate.com/sun/spain/barcelona.

[52] *2019 Weather in Barcelona — Graph*. URL: https://www.timeanddate.com/weather/spain/barcelona/historic?month=6&year=2019.