ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
Τμήμα Ηλεκτρολόγων & Ηλεκτρονικών Μηχανικών

UNIVERSITY of WEST ATTICA
FACULTY OF ENGINEERING
Department of Electrical & Electronics Engineering

Πρόγραμμα Μεταπτυχιακών Σπουδών
Διαδικτυωμένα Ηλεκτρονικά Συστήματα

Master of Science in
Internetworked Electronic Systems

Μεταπτυχιακή Διπλωματική Εργασία

Μέτρηση ποιότητας αέρα, μέσω υποδομών αστικών μεταφορών

Μεταπτυχιακός Φοιτητής : Μονιός Νικόλαος, ΑΜ: 0011
Επιβλέπων : Χαράλαμπος Πατρικάκης, Αναπλ. Καθηγητής

ΑΙΓΑΛΕΩ, ΣΕΠΤΕΜΒΡΙΟΣ 2018

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
Τμήμα Ηλεκτρολόγων & Ηλεκτρονικών Μηχανικών

UNIVERSITY of WEST ATTICA
FACULTY OF ENGINEERING
Department of Electrical & Electronics Engineering

Πρόγραμμα Μεταπτυχιακών Σπουδών
Διαδικτυωμένα Ηλεκτρονικά Συστήματα

Master of Science in
Internetworked Electronic Systems

MSc Thesis

Air quality monitoring, using urban public-transportation infrastructure

Student: Monios Nikolaos, Reg. Nr.: 0011
MSc Thesis Supervisor: Charalampos Patrikakis, Assoc. Professor

ATHENS-EGALEO, SEPTEMBER 2018

ΠΕΡΙΛΗΨΗ

Καθώς πλησιάζουμε όλο και πιο κοντά στην εποχή των έξυπνων πόλεων, είναι σημαντικό να έχουμε υπηρεσίες που ωφελούν την πόλη και τον πολίτη, καταναλώνοντας όσο το δυνατόν λιγότερη ενέργεια και με μικρό χρόνο απόκρισης.

Μια τέτοια πολύ σημαντική υπηρεσία είναι η παρακολούθηση της ποιότητας του αέρα μέσα στην πόλη σε πραγματικό χρόνο, και η έγκαιρη ειδοποίηση των πολιτών, καθώς το ένα όγδοο των θανάτων παγκοσμίως προκαλείται από την ρύπανση του αέρα [5].

Μολονότι υπάρχουν έρευνες και εργασίες για μικρούς σταθμούς οι οποίοι είναι εφοδιασμένοι με αισθητήρες, η νέα τάση στην τεχνολογία είναι η εγκατάσταση αυτών των σταθμών σε οχήματα και η παρακολούθηση της ποιότητας του αέρα κατά τη διάρκεια της διαδρομής του, καθώς έτσι θα μπορέσουμε να πάρουμε μετρήσεις από σημεία όπου ένας ακίνητος σταθμός δεν θα μπορούσε να πάρει.

Η τεχνολογία για τους κινητούς σταθμούς δεν έχει ωριμάσει ακόμα, σε σχέση με τους ακίνητους, και υπάρχουν λίγες διαθέσιμες πηγές για μελέτη, παρόλα αυτά μπορούμε να περιμένουμε ότι στο μέλλον περισσότερα πρότζεκτ θα εμφανιστούν και θα προτείνουν λύσεις.

Η εργασία είναι δομημένη σε 6 κεφάλαια. Στο πρώτο κεφάλαιο γίνεται μια εισαγωγή του αναγνώστη στην τεχνολογία του Διαδικτύου των Πραγμάτων (Internet of Things – IoT), τα sensor-nodes και τα wireless sensor networks. Επίσης παρουσιάζονται

οι κύριες τεχνολογίες σχετικά με το hardware, το λογισμικό αλλά και τις τηλεπικοινωνίες, που βοήθησαν στην άνθιση τους.

Στο δεύτερο κεφάλαιο παρουσιάζεται η σημασία της παρακολούθησης της ποιότητας του αέρα στην πόλη, καθώς και IoT πρότζεκτ για σταθερούς και κινητούς σταθμούς και βελτιώσεις του νέφους.

Το τρίτο κεφάλαιο προτείνει έναν σταθμό για τη μέτρηση της ποιότητας του αέρα, ο οποίος σαν ειδικό χαρακτηριστικό έχει το ότι βρίσκεται τοποθετημένος σε αστικό όχημα μαζικής μεταφοράς. Εδώ δίνονται τα τεχνικά χαρακτηριστικά του σταθμού, το hardware που επιλέχθηκε, και διαγράμματα ροής του λογισμικού.

Στο τέταρτο κεφάλαιο παρουσιάζονται τα αποτελέσματα των πειραμάτων που έγιναν με τον προτεινόμενο σταθμό. Οι περιοχές που μελετήθηκαν ήταν η απόδοση της συσκευής energy-harvesting, η ακεραιότητα του GPS, η μετάδοση μεγάλων μηνυμάτων, και την επίδραση του κώδικα στον σταθμό.

Το πέμπτο κεφάλαιο αναλύει τα αποτελέσματα των πειραμάτων και επιχειρεί να τα συγκρίνει με αποτελέσματα άλλων εργασιών. Επίσης παρουσιάζονται περιοχές που χρειάστηκαν να αναθεωρηθούν μετά τα πειράματα και προβλήματα που συναντήθηκαν στη σχεδίαση του σταθμού.

Το τελευταίο κεφάλαιο παρουσιάζει τα συμπεράσματα της εργασίας, και δίνει μια γενική κατεύθυνση για μελλοντικές βελτιώσεις.

KEYWORDS: Air-Quality, Autonomy, Internet of Things, Low-Power, Public Transportation Infrastructure, Wireless Sensor-Networks

ABSTRACT

As we move closer to the smart-cities era, it is important that we have services that serve the city and its citizens, spending less energy and with minimal delay.

One very important service is the real-time monitoring of the air quality inside the city, and the timely warning towards the citizens, given that the one eighth of deaths annually is caused by air-pollution [5].

Although there are researches and projects for stationary nodes that can sense the quality of the air nearby, the latest trend is to embed these nodes to vehicles and monitor the air quality during their route, as it will be able to gather samples from areas that a stationary node would not reach.

This technology for remote nodes is not nourished yet, in contrast to the stationary nodes, and there are a few resources available to study, but we can expect that in the near future more projects will emerge and offer proposals.

This study is structured in 6 chapters. The first chapter introduces the reader to the Internet of Things (IoT), sensor-nodes, and wireless sensor networks notions. The key technologies behind hardware, software and telecommunications that led to their development will also be presented and discussed.

The second chapter presents the importance of sensing the quality of the air in a city, as well as IoT-based projects for both stationary and moving nodes, and cloud enhancements.

The third chapter proposes a node for measuring the quality of the air, which will have the special characteristic of being embedded on a public transportation vehicle. The specifications of the node, the hardware chosen, and some software concepts are also given here.

The fourth chapter presents the results taken from various tests, from the proposed node. The areas that were studied are the efficiency of the energy-harvesting module, the reliability of the GPS module, the transmission of long messages, and the effect of the software design on the node.

The fifth chapter analyses the previous results and attempts to compare them with the results of other projects. Furthermore, it presents areas that had to be adjusted after the tests and the troubles faced during the design.

The final chapter presents the conclusion of this study, and gives a general direction for enhancements on future projects.

KEYWORDS: Air-Quality, Autonomy, Internet of Things, Low-Power, Public Transportation Infrastructure, Wireless Sensor-Networks

TABLE OF ABBREVIATIONS

6LoWPAN	IPV6 Low Power Wireless Personal Area Network
ADC	Analogue-to-Digital Converter
AI	Artificial Intelligence
ANN	Artificial Neural Networks
BLE	Bluetooth Low Energy
EEPROM	Electrically Erasable Programmable Read-Only Memory
GPIO	General Purpose Input/Output
GPU	Graphical Processing Unit
IDE	Integrated Development Environment
IoT	Internet of Things
ITU	International Telecommunication Union
LoS	Line of Sight
MAC	Medium Access Control
MEMS	Micro-Electro-Mechanical Sensor
MPP	Maximum Power Point
MQTT	Message-Queueing Telemetry Transport
OS	Operating System
OTA	Over-the-Air
PCB	Printed Circuit Board
PoE	Power-over-Ethernet
PWM	Pulse Width Modulation
RTC	Real-Time Clock
SBC	Single Board Computer
SoC	System-on-Chip
SVM	Support Vector Machines
WHO	World Health Organization
WSN	Wireless Sensor Network

TABLE OF CONTENTS

CHAPTER 1: Introduction to the Internet of Things (IoT) and sensor-nodes 12
1.1 What is the Internet of Things (IoT)? 12
1.2 The technology behind the IoT 16
1.2.1 Open-source, low-cost hardware 16
1.2.1.1 Arduino – A board for rapid development of microcontroller applications 16
1.2.1.2 Raspberry Pi Single Board Computers (SCBs) 22
1.2.2 Open-source and cloud-based IDEs for embedded programming 23
1.3 What is a sensor-node? 26
1.4 The technology behind sensor-nodes 28
1.4.1 IEEE 802.15.4 based networks 28
1.4.2 LoRa technology 35
1.4.3 MQTT protocol 36
1.4.4 IoT gateways 37
CHAPTER 2: Projects on urban air-quality monitoring 39
2.1 Measuring the quality of the air 39
2.2 IoT projects for air-quality measurement using stationary sensor-nodes 41
2.2.1 Microcontroller-based projects 42
2.2.2 SBC-based projects 44
2.3 IoT projects for air-quality measurement using public-transportation infrastructure. 47
2.3.1 Microcontroller-based projects 48
2.3.2 SBC-based projects 50
2.4 Cloud-based improvements 50
CHAPTER 3: Proposed WSN for air-quality monitoring 52
3.1 WSN specifications 52
3.2 Hardware chosen for WSN Autonomy 53
3.3 Software for minimal energy consumption 57
CHAPTER 4: Results of application 66
4.1 Solar panel efficiency 66
4.2 Code efficiency 68
4.3 XBee communication 71
4.4 GPS reliability 77

CHAPTER 5: Analysis of the results **79**
CHAPTER 6: Conclusion - Proposals **83**
References **86**
Appendix A: Source Code **89**
coefficients.h 89
air_quality.pde 92

TABLE OF FIGURES

Figure 1: A depiction of the IoT technology [source https://www.cloudwards.net/what-is-the-internet-of-things/] 14
Figure 2: IoT topology. The dashed lines indicate that communication is capable but not a necessity. Two things can either interact locally using non-internet based communication protocols (for example LoRa, Zigbee, BLE etc), or remotely using the internet (TCP/IP , UDP etc). 15
Figure 3: Google trends for “internet of things” [8] 16
Figure 4: The Arduino UNO development board. Its pin layout has now become a standard and it is used by many other companies (source: https://store.arduino.cc/usa/arduino-uno-rev3) 18
Figure 5: Some development boards that are available in the market 19
Figure 6: The shield for the Waspmote that is used for air-quality and gases monitoring, alongside the sensors (Waspmote gases sensor board, source https://www.cooking-hacks.com/Waspmote-gasses-sensor-board) 19
Figure 7: The credit-card-sized SBC Raspberry Pi 3 model B+, featuring a quad-core 64-bit processor at 1.4 GHz and 1 GB RAM (source: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/) 23
Figure 8: A simple example using the Arduino IDE (source https://www.pololu.com/docs/0J61/6.2) 24
Figure 9: Illustration of the most important parts on a sensor node architecture. 26
Figure 10: The B-L475E-IOT01A sensor-node by ST Microelectronics (source [9]) 28
Figure 11: OSI communications model (source: https://www.electronicdesign.com/what-s-difference-between/what-s-difference-between-ieee-802154-and-zigbee-wireless) 30
Figure 12: A star network (a) and a P2P network (b) (source: [12]) 32
Figure 13: A mesh network. Node “A” can communicate with node “F” using two paths, e.g. ABDF, ACDF (source: [12]) 32
Figure 14: An XBee module alongside its antenna, and its pin layout (source: http://www.electronicwings.com/sensors-modules/xbee-module) 34
Figure 15: A chirp signal. A chirp spread modulation is a wideband frequency modulated sinusoidal signal that increases or decreases over time (source: [19]) 35
Figure 16: Basic MQTT topology. The client to the left publishes data to a topic to the broker. The clients that are subscribed to the topic will receive the data from the broker (source: https://spindance.com/smart-consumption-smart-products-understanding-mqtt/) 37
Figure 17: Two commonly used IoT gateways. Libelium’s Meshlium to the left, the Things gateway to the right 38
Figure 18: An air-quality monitoring station in Hong Kong (source: http://www.atimes.com/article/serious-air-pollution-tuen-mun-tung-chung/) 40
Figure 19: A Waspmote sensor-node by Libelium, can be easily installed in a light pole (source: http://www.libelium.com/products/plug-sense/technical-overview/) 42
Figure 20: The GroovePi+ HAT embedded on top of a Raspberry Pi SBC. The HAT provides an interface for connecting Groove sensors to the Pi (source: http://wiki.seeedstudio.com/GrovePi_Plus/) 45
Figure 21: The Beaglebone Black SBC (source: https://beagleboard.org/black) 47
Figure 22: Topology of fog computing (source: http://www.securens.in/securens_blog/demystifying-fog-computing-and-its-role-in-e-surveillance-part-1/) 51
Figure 23: The Waspmote (left), the XBee module (right) and its antenna (top) (source: https://www.cooking-hacks.com/Waspmote-868-sma-4-5-dbi) 55

Figure 24: Top-side view of the node. The dust filter enables the sensors to have exposure to the environment. 56
Figure 25: Top view of the node. 56
Figure 26: Inside view of the box. A soft layer under the Waspmote helps facing the vibrations that will be caused by the vehicle. 57
Figure 27: flow chart of the code that ensures low-power use of the node........ 58
Figure 28: Flow chart of the node's main process 60
Figure 29: Charging / discharging status of the battery, using the solar panel 67
Figure 30: Percentage of the battery's available after 2 tests, during the day-time mode 70
Figure 31: Percentage of the battery's available power after 3 hours, during the night-time mode 70
Figure 32: Meshlium visualizer tool. The data that have been captured will be illustrated as time-series charts (Source: [40]) 75
Figure 33: The cloud application of Meshlium. Data transmitted to the gateway will be parsed and illustrated here........ 76

LIST OF TABLES

Table 1: Comparison of commonly used development platforms for the IoT 21

CHAPTER 1:
Introduction to the Internet of Things (IoT) and sensor-nodes

1.1 What is the Internet of Things (IoT)?

The **Internet of Things** (IoT) is a technological trend that was born around 2009. One of the first definitions of the IoT was that is about creating a network of objects that can interact with humans or with other objects, using internet-based services [1]. As of 2012, a more formal definition has been proposed by the International Telecommunication Union (ITU) in Recommendation ITU-T Y.2060 and it is the following: "a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies."[2]

According to the UN-HABITAT 2011 report, 2007 was the first year in recorded history that the majority of the human population (more than 50%) was living in an urban environment rather than a rural one, and the trend indicates that this is irreversible. It is therefore required a strong transformation of a city's infrastructure, in order to be able to support the growing population; roads, public transportation, electricity grid, litter collection etc. must be able to sense, predict and react on their own [5]. The city will transform to a smart-city, and the technology that will help us achieve this transformation is the IoT.

Although we have portable and light-weight devices connected to the internet for over a decade, which are able to communicate with other devices (e.g. smartphones, tablets, smart TVs, smart watches etc.) using the Internet, the IoT aims to connect every imaginable device (or, "thing") to the internet and enhance it with some kind of intelligence using ultra-low-power microcontrollers and a cloud-based service – in contrast with the smartphones, tablets

etc. that the vast majority of the processes is done in the device; for example, the Apple iPhone 8 and iPhone X are using a 6-core, 2.39 GHz, 64-bit ARM System-on-Chip (SoC) to sample measurements from sensors, control the device, process images etc. [6]

Some of the areas that researchers, engineers and developers are creating applications for the IoT are [7]:

- smart homes & building automation
- smart agriculture
- energy management / water consumption – detection of leakage
- environmental monitoring
- infrastructure monitoring
- manufacturing & industry – Industrial Internet of Things (IIoT)
- wearables for healthcare and elder care
- smart transportation
- smart cities

The effort is focused on making each object "smart" and autonomous. Devices are getting attached with sensors, microcontrollers, and communication devices and often come with energy-harvesting modules, thus becoming smarter and help in optimizing existing systems. The trend in the IoT technology is the use of **Micro-Electro-Mechanical Sensors (MEMS)**, in order to reduce the size, cost and energy consumption of the application, as well as adding long-term reliability and stability [8]. Furthermore, another trend is the use of **Artificial Intelligence (AI)** either in cloud-based services or embedded within the microcontroller.

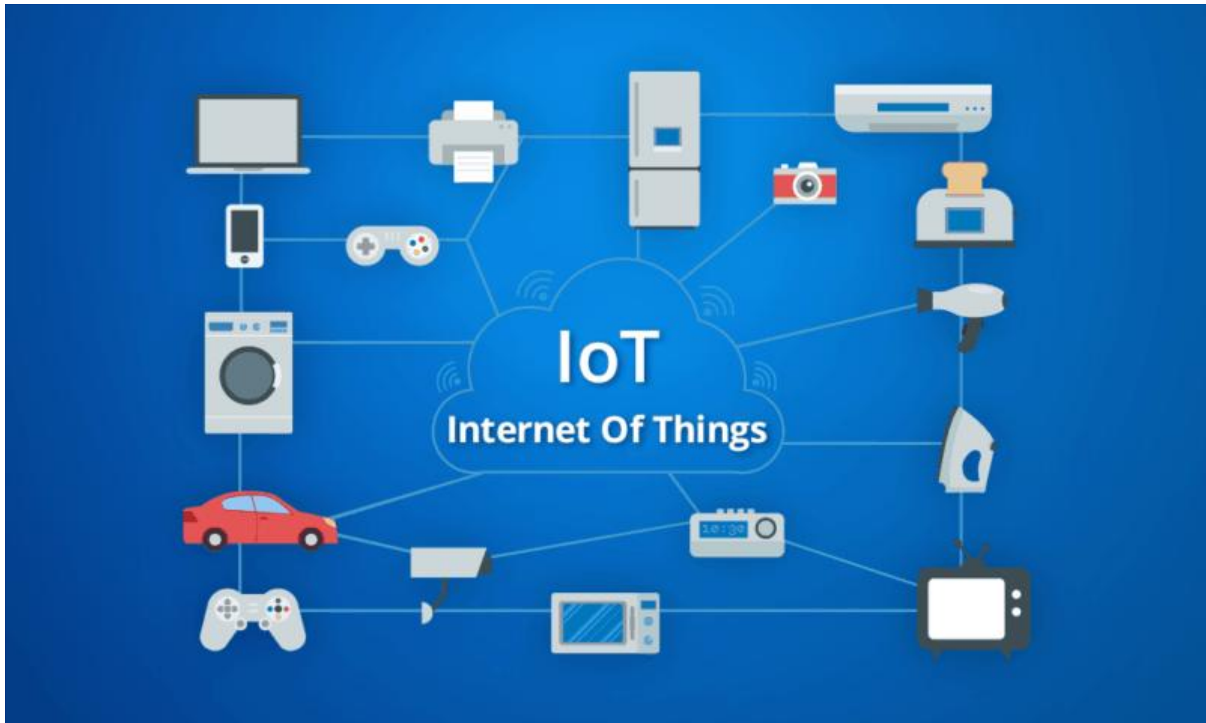

Figure 1: A depiction of the IoT technology [source https://www.cloudwards.net/what-is-the-internet-of-things/]

Hence, the IoT requires devices that enable information gathering and data processing, and modules that enable communication (wired or wireless) with other things or internet services. In some applications, a direct communication with the internet is not possible, either for security reasons or for energy consumption reasons. In that case, a gateway between the thing and the internet is used, to interpret messages from and to the thing. The communication between the thing and the gateway is achieved with wireless technologies like **Zigbee**, **Bluetooth Low-Energy** (BLE), **LoRa**, **Sigfox** etc. A general topology for the IoT can be seen in the following picture.

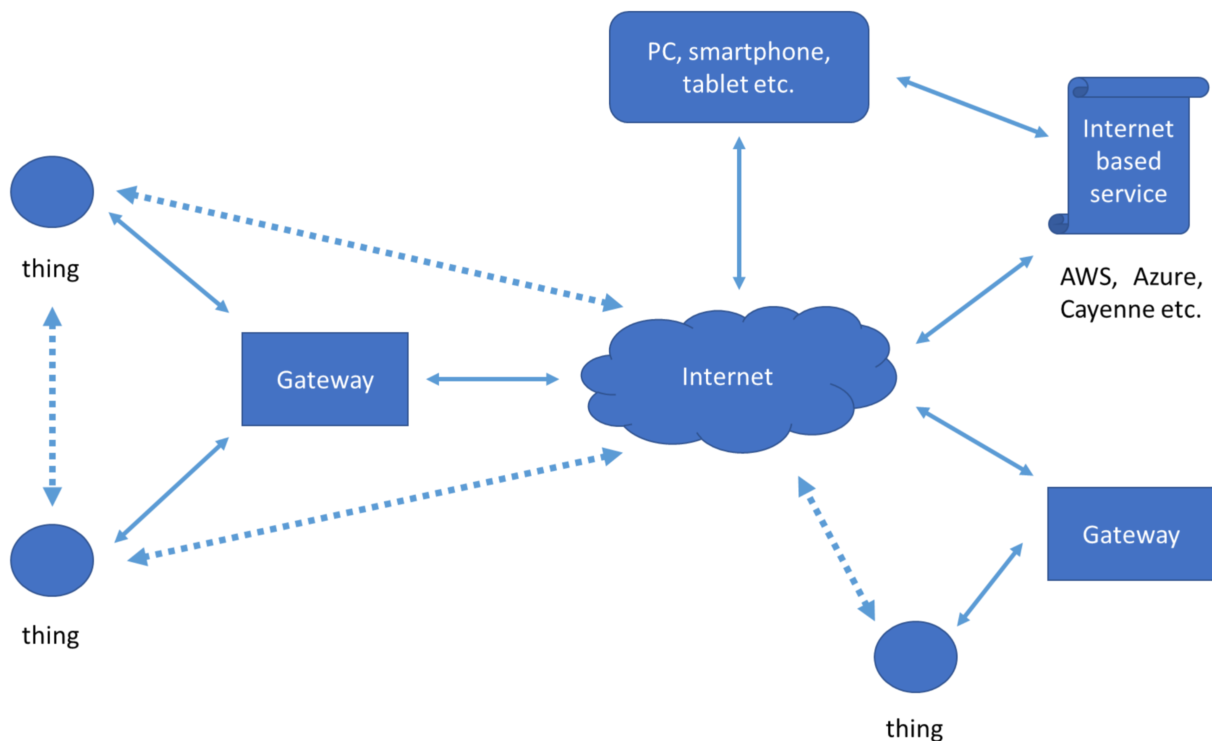

Figure 2: IoT topology. The dashed lines indicate that communication is capable but not a necessity. Two things can either interact locally using non-internet based communication protocols (for example LoRa, Zigbee, BLE etc), or remotely using the internet (TCP/IP , UDP etc).

The impact that the IoT technology had in both the academia and the industry is tremendous. In the following graph the google-trends of a decade for the IoT is illustrated.

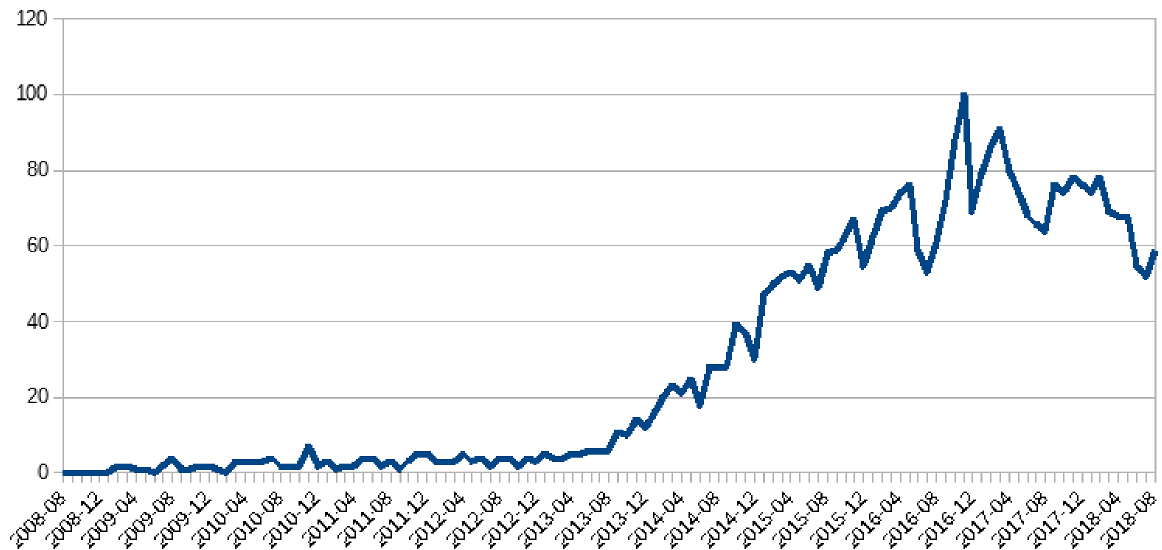

Figure 3: Google trends for “internet of things” [8]

1.2 The technology behind the IoT

As seen in figure 4, IoT took some time to get established. Certain technological advances in hardware and telecommunications domains had to be met, some others, older, had to be reconsidered, before being useful for the IoT. Here, these advancements are presented.

1.2.1 Open-source, low-cost hardware

The control units play perhaps the most important role in an IoT application, as it will be directly attached to the thing. These control units can either be microcontroller-based development boards or Single Board Computers. The advancements on these two categories are presented here.

1.2.1.1 Arduino – A board for rapid development of microcontroller applications

One of the biggest factors that enabled the idea of IoT to nourish and grow, was the revolution that the open-source-hardware embedded-platform **Arduino** brought to the

academia, to professionals and to hobbyists; it was now faster and cheaper to program an application using a microcontroller (ATmega32P) [3].

This board was the Arduino UNO, and it offered 14 digital I/O pins (6 of them provide PWM output as well), 6 Analog input pins (10-bits resolution), 32 Kbytes Flash memory, 2 Kbytes SRAM memory, 1 Kbytes EEPROM and was running at 16 MHz. UNO offered UART, I2C and SPI communication options with other peripherals and devices. The Arduino Mega followed next, which offered more I/O pins, more ADCs, more flash and RAM, amongst others.

The Arduino helped in rapid prototyping, going from the initial idea to a first working prototype while bypassing a lot of time-consuming lab-based tasks, like the PCB design and manufacturing, the need of an external –often expensive- debugger, the need to write every peripheral driver and libraries from scratch.

Alongside these boards, the Arduino foundation and other companies also designed "**shields**" that can be attached to the boards and expand their capabilities. Two of these shields enable internet communication using either Wi-Fi or Ethernet, while another shield adds Zigbee or LoRa capabilities. There are also shields for motor control, GPS data acquisition, navigation and locomotion, etc.

The shields use the same pin layout of the Arduino board. This layout has now become a standard and is used by many other development boards and shields.

Figure 4: The Arduino UNO development board. Its pin layout has now become a standard and it is used by many other companies (source: https://store.arduino.cc/usa/arduino-uno-rev3)

Having seen the success of Arduino, many electronics companies created their own open-source-hardware platforms, giving more tools and solutions to students, engineers, and hobbyists. For example, **Texas Instruments** has created the Launchpads series which mostly target applications that require ultra-low-power consumption, **ST Microelectonics** has created the Nucleo boards, which use the ARM architecture and offer more pin options, more analogue-to-digital converters (ADC) with higher resolution than the original Arduino UNO, more communication interfaces etc.

Arduino has expanded since then, releasing more development boards with better specifications and more connectivity options; there are now Arduino boards with integrated WiFi, BLE or LoRa chips. Other companies are using the low-cost SoC microchip **ESP8266** which has integrated WiFi and Bluetooth capabilities.

There are also companies that created their Arduino-like platform, and target specific IoT areas. One of these, **Libelium**, has created the platform **“Waspmote”**, which uses a

different microcontroller and pin layout than the ones that Arduino uses, hence the shields of Arduino cannot be used with it. It comes with its own Arduino-like IDE though, which adds compatibility with the official Arduino libraries. The Waspmote offers its own shields which target specific IoT areas; there are shields for water-quality monitoring, air-quality and gases monitoring, aquaponics and agricultural use-cases, smart cities, and others.

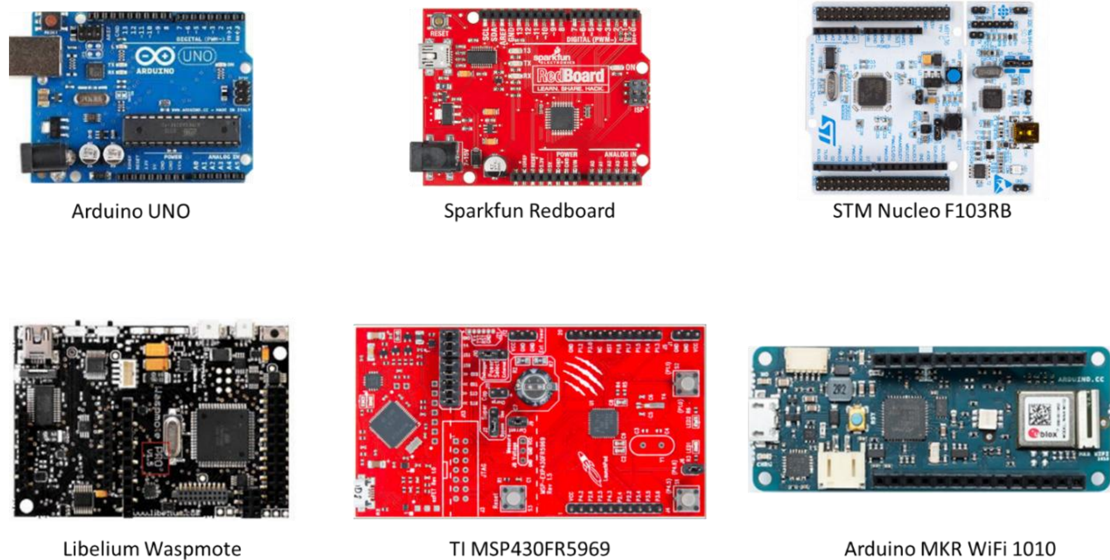

Figure 5: Some development boards that are available in the market

Figure 6: The shield for the Waspmote that is used for air-quality and gases monitoring, alongside the sensors (Waspmote gases sensor board, source https://www.cooking-hacks.com/Waspmote-gasses-sensor-board)

Next, a table of comparison of various commonly used IoT boards is presented. Since many companies are now releasing their own boards, a small attempt to compare the pros and cons of every board is illustrated, in order to help the reader decide which one fits his/hers own needs. A few boards are targeting fast internet connectivity (Arduino MKR WiFi 1010, Particle Photon, Sparkfun ESP8266 Thing), while neglecting the energy consumption of the board, and others focus on general utility without offering any wireless communication modules; if necessary, a shield that adds this functionality can be used.

Table 1: Comparison of commonly used development platforms for the IoT

Board	MCU	GPIO	Flash	RAM	ADC	Wireless connectivity	Energy consumption
Arduino UNO	8-bit ATmega328P @16 MHz	14	32 KB	2 KB	6x 10-bit	N/A	0.2 mA / MHz in active mode
Arduino MKR WiFi 1010	32-bit ARM M0+ @48 MHz	20-27	256 KB	32 KB	7x 8/10/12-bit	WiFi, BLE	Min 3.37 mA @ 48 Mhz, WiFi off
TI MSP430FR5969	16-bit RISC @16 MHz	40	64 KB unified FRAM		16x 16-bit	N/A	100 μA / MHz in active mode
STM Nucleo F103RB	32-bit ARM M3 @72 MHz	51	128 KB	20 KB	6x 12-bit	N/A	27 mA @ 72 MHz in active mode
Libelium Waspmote	8-bit ATmega1281 @14 MHz	25	128 KB	8 KB	7x 10-bit	N/A	500 μA / MHz in active mode
Cypress PSoC 4200	ARM M0 @48 MHz	98	256 KB	32 KB	Up to the user	BLE	12.8 mA @ 48 MHz
NXP FRDM-KL25Z	32-bit ARM M0+ @48 MHz	53	128 KB	16 KB	6x 16-bit	N/A	6.8 mA @ 48 MHz
Particle Photon	32-bit ARM M3 @120 MHz	18	1 MB	128 KB	6	WiFi	80 mA in active mode, WiFi off
Sparkfun ESP8266 Thing	32-bit MCU @80 MHz integrated in SoC	17	512 KB	36 KB	1	WiFi	80 mA in active mode, WiFi off

1.2.1.2 Raspberry Pi Single Board Computers (SCBs)

In 2012 a British charity called the "Raspberry Pi Foundation", that wished to promote computer-science in schools [26], designed a low-cost single-board computer which featured a 32-bit ARM processor at 700 MHz, a Graphics Processing Unit (GPU), 256 MB RAM, USB ports, SD card interface, HDMI interface, audio I/O, MIPI camera interface, and digital GPIO interface, all in a credit-card-sized board, for the price of 25 $ [27]. They named their SBC **Raspberry Pi**.

The Raspberry Pi uses a Debian-based Linux distribution Operating System (OS), called **"Raspbian"**, which is open-source, and boots from the SD card. Its pin layout has become a standard, much like the Arduino one, and there many shields, called **HATs**, that can be embedded on the SBC and expand its utilities.

The Pi was greatly accepted not only by scholars and the academia, but from hobbyists as well that wished to get introduced to Linux. The Pi quickly expanded to other uses, like robotics, home-automation, media servers etc. The secret-ingredient for this success is the support from the community that a user receives when facing a problem. There are countless tutorials and online resources to help the user tackle problems. Every sensor or peripheral that can be used by the Arduino, has also libraries for the Pi.

Since then, the foundation has released many SBCs as an upgrade, adding Ethernet and WiFi interfaces, switching to 64-bit ARM quad-core processors up to 1.4 GHz, 1 GB RAM, MicroSHDC slot and others. The price has risen by 10 $, but the overall price is again considered low for the specifications. By March 2018, the foundation had sold 19 million SBCs [25].

In the IoT, the Pi is used not only as an end-device or a node which takes samples from sensors or controls actuators, but many times as a server or as a gateway because of the

stability and security that a Linux-based OS can offer. Being a computer in a credit-card size, the user can use any programming language or framework is required for the application, given that it can be downloaded in a 64-bit ARM processor.

Since then, many other companies have followed the Pi example and have released their own Linux-based SBC, either following the Pi layout standard or creating expansion boards that add this layout. Some boards target robotic applications, for example the **Beagleboard**, others target media servers, for example **Orange Pi** and **Banana Pi**, and others simply pump the specifications of the processor and RAM up, like the **ODROID**. The Raspberry Pi continues to be the favourite SBC of the community though, due to the fact that the other SBCs lack resources or support [28].

Figure 7: The credit-card-sized SBC Raspberry Pi 3 model B+, featuring a quad-core 64-bit processor at 1.4 GHz and 1 GB RAM (source: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/)

1.2.2 Open-source and cloud-based IDEs for embedded programming

Arduino also introduced its own **IDE**, alongside the boards, which supports all its development boards with all the libraries that any developer might need (UART, SPI, I2C

etc.). It is no longer necessary to write a library from scratch when you switch boards, spending many hours reading the technical datasheet for the registers of the new microcontroller. All it needs to be done is that the developer includes this library in the application; the compiler will build the code with the proper registers. The developer can write in C and C++.

The initialization of the application and its components is done in a function called setup(), which is executed before the main function which is called loop(), as seen in the following picture.

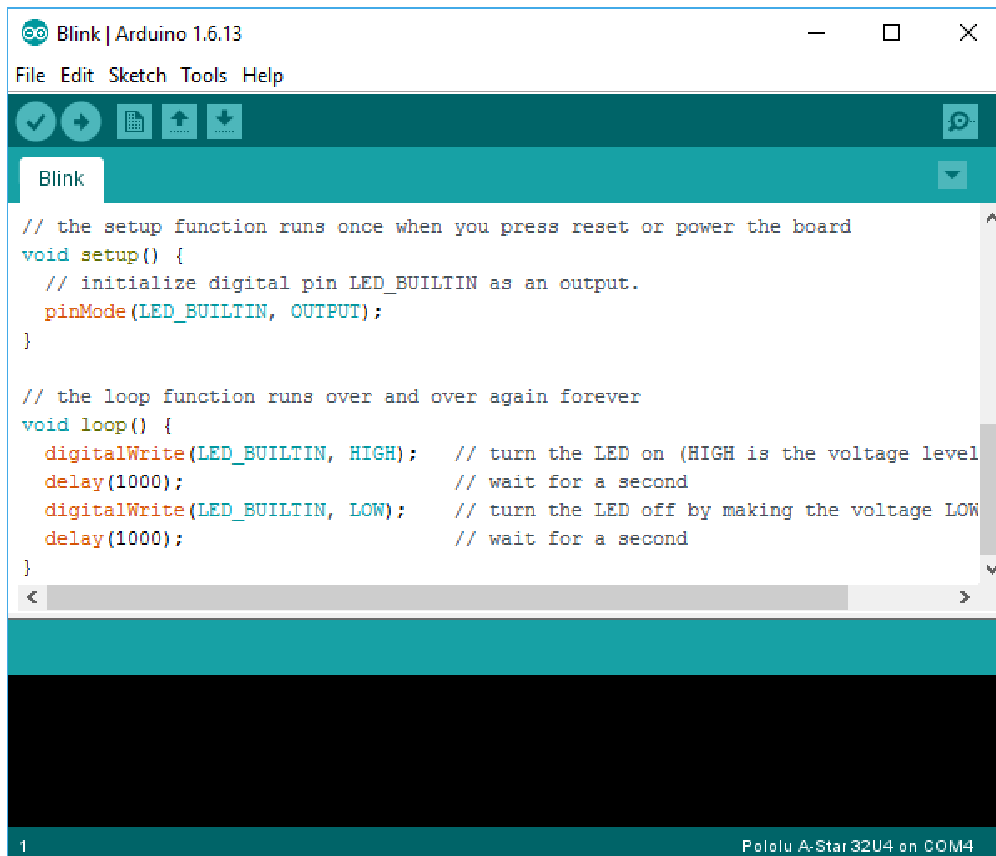

Figure 8: A simple example using the Arduino IDE (source https://www.pololu.com/docs/0J61/6.2)

In the example seen in the picture, the built-in LED of the development board is initialized as an output. Then, the program enters the endless loop function (void loop()), where it will blink the LED with a 1 Hz frequency for ever.

Nearly all the companies that created their own Arduino-like development boards, released addons for the Arduino IDE, so the developer can program them using the official Arduino IDE. Other companies, like Libelium, have created a clone of the Arduino IDE that supports only their products.

ARM has created a cloud-based IDE, called **Arm mbed**. The developer can write and compile code from any place in the world, as long as there is internet connection. Following the Arduino example, ARM has created its own libraries for the boards that it supports, to ensure seamless code portability when changing development boards. Alongside the code editor and the compiler, Arm mbed also includes a code revision tool, making it a really powerful cloud IDE.

Texas Instruments also released a cloud-based IDE for its development boards, but the developer has to write the code from scratch as there are no libraries for peripherals available, which increases the development working hours. An Arduino-like IDE for the Texas Instruments boards exists, called **Energia**, and adds compatibility with the official Arduino libraries for the Texas Instruments boards.

Both hardware-wise and software-wise, the transition to low-cost open-source development platforms and services, played an important role for the blossoming of the IoT. Never before a project that required a microcontroller was it that easy to begin from scratch.

1.3 What is a sensor-node?

The combination of one of the previously discussed platforms with one or more shields and a few sensors, create a **sensor-node**. The sensor-node is defined to be a device within a network which has processing capabilities, an interface with analogue and digital sensors, and a communication device to help it communicate with other sensor-nodes or a gateway [4]. The most important parts of a sensor node are the microcontroller, the transceiver, the external memory, the power supply unit, and the interface with one or more sensors.

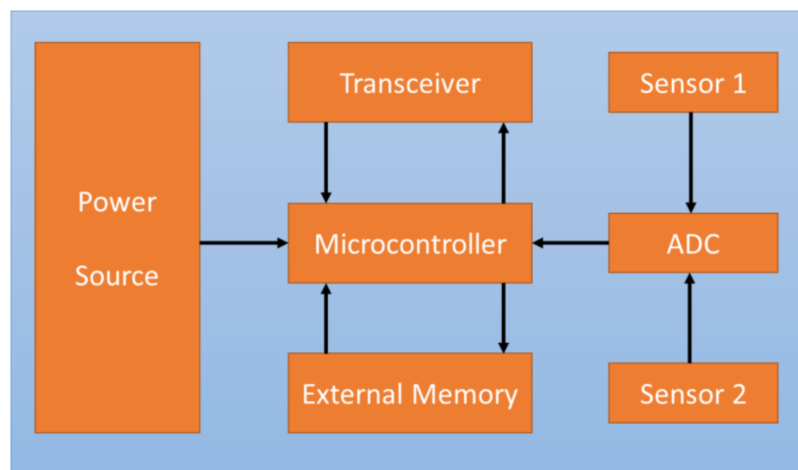

Figure 9: Illustration of the most important parts on a sensor node architecture.

The purpose of the sensor-node is to add sensing capabilities to remote things, in the IoT topology. There are occasions that these things do not have any power source near them, so the sensor-node must have ultra-low-power capabilities; an ultra-low-power microcontroller powered by 3.3 Volts, spending the majority of its duty-cycle in sleep mode, and making a few transmissions per day to the gateway. Regularly, some energy-harvesting devices are attached to the sensor-node to add more autonomy. For example, a small solar-panel (70x55 mm) can guarantee an endless function of the sensor-node, given it is installed in a sunny area and the board has a low duty-cycle with few daily transmissions.

Modern development boards host the vast majority of these components, and with the addition of the missing components, they qualify for the sensor-node description. Some of the latest boards, like the B-L475E-IOT01A by ST Microelectronics, actually host all the components in a single development board, thus making it ready for field installation once it is programmed.

The B-L475E-IOT01A is an all-in-one solution, offering in a single board [9]:

- ARM Cortex M4 microcontroller
- MB Flash
- 128 KB RAM
- Bluetooth Low Energy module
- LoRa module
- Near Field Communication (NFC) module
- WiFi module
- 7 analogue and digital on-board sensors
- Compatibility with Arduino shields
- Power regulator for external power-supply sources
- Arm mbed support

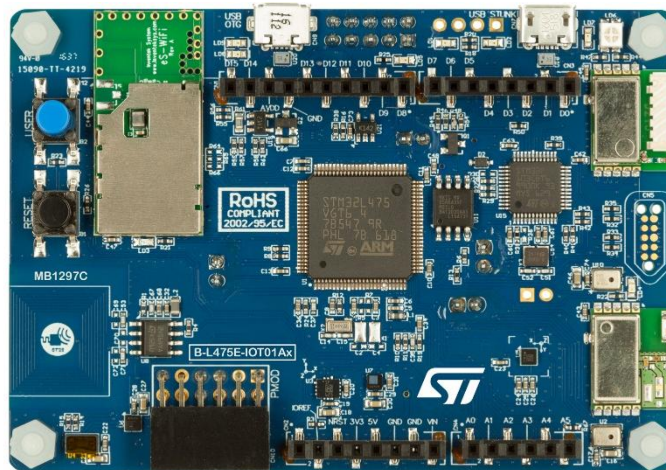

Figure 10: The B-L475E-IOT01A sensor-node by ST Microelectronics (source [9])

There are many companies, universities, and research groups that have created their own sensor-nodes for various applications. A comprehensive list of these sensor-nodes and their specifications can be accessed in the following link: https://en.wikipedia.org/wiki/List_of_wireless_sensor_nodes

1.4 The technology behind sensor-nodes

As discussed earlier, the advances in low-power, low-cost, small-sized hardware played a significant role in the IoT, and this applies to the sensor-nodes too. Although the IoT requires that a device has access to the internet, this is not always true for the sensor-node as most of the time the node is situated in a remote place and powered by a battery; constant internet access would drain the battery really fast.

1.4.1 IEEE 802.15.4 based networks

What also helped the sensor-node technology is the advances in low-power, narrow-band, short-range and long-range wireless telecommunication protocols. In 2007 an idea that that the internet protocol can and has to be applied to all devices, even the smallest, was published [10]. The author named this protocol for the small devices **6LoWPAN** (IPV6 Low-

Power Wireless Personal Area Networks), and it targets in enabling IPV6 packets to be carried on low-power networks.

6LoWPAN is a protocol that gives definitions for mechanisms that allow IPV6 communication over **IEEE 802.15.4** networks (low-rate wireless personal area network). The device that use 802.15.4 networks [11] are low-cost, low-power, and require short-range and low bit-rate communication.

The devices that use 802.15.4 protocol often have limited computational power, limited memory and limited energy availability, a sensor-node is known to have all these attributes.

The 802.15.4 standard gives the definitions of the physical and MAC layers of the OSI model of a network (layers 1 and 2). At each OSI layer, the address and other headers and error detection fields are added to the data when in transmission mode, while when in receiving mode, these fields are removed so the device gets just the data, as seen in the next picture.

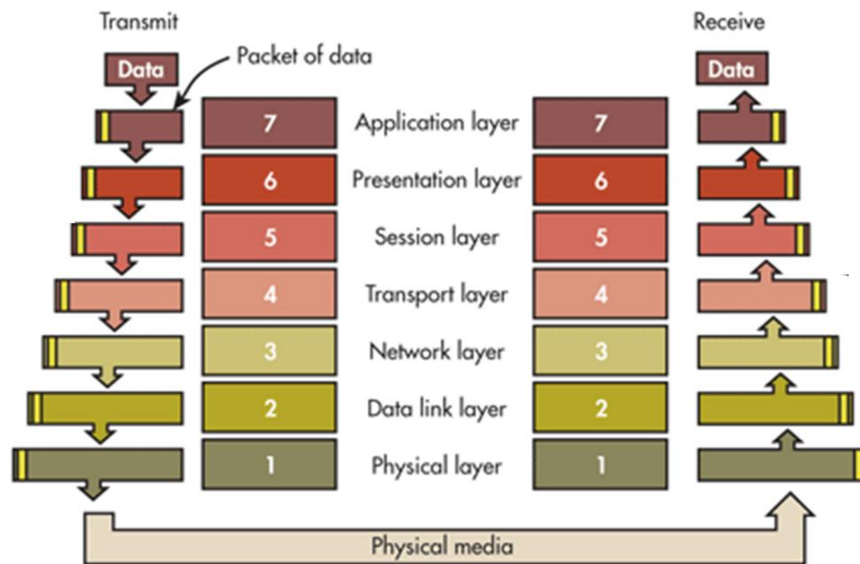

Figure 11: OSI communications model (source: https://www.electronicdesign.com/what-s-difference-between/what-s-difference-between-ieee-802154-and-zigbee-wireless)

In the physical layer there are definitions for the frequency, the power, the modulation and other wireless conditions of the link, while the MAC layer gives the definitions of the format of the data. The other layers have definitions of other data-handling measures and protocol enhancements, including the final application [12].

The 802.15.4 standard offers 3 bands for use. 868 MHz for Europe (BPSK modulation, 20 Kbits / sec), 915 MHz for North America (BPSK modulation, 40 Kbits / sec), and 2.4 GHz for worldwide use (Q-QPSK modulation, 250 Kbits / sec) [12].

In 802.15.4 networks, devices are divided into Personal Area Networks (PAN), and each PAN has a unique 16-bit PAN identifier, which can either be predefined or scanned during start-up from a coordinator. Devices have 2 addresses, a 64-bit long global one, and the 16-bit PAN one [13].

According to the 802.15.4 standard, the size of the packet must be 127 octets (1 octet consists of 8 bits), which means that with a maximum frame of 25 octets the remaining 102

octets are spared for the MAC layer, and if a link-layer security is desired, then the spared octets drop to 81. Given that 40 octets are used for IP header and 8 octets for the UDP header, the actual data size is 33 octets, attributes that are small for IPV6 communication. In contrast, IPV6 communication requires 1280 bytes sized packets. The solution to this is described in the 6LoWPAN protocol, and is the packet fragmentation which occurs below the network layer, and the compression of the IP address (when it can be derived from other headers), the compression of the link-local prefix (fe80::), and the compression of common headers like the TCP, UDP, and ICMP. Furthermore, 6LoWPAN offers a Mesh Address Header that enables routing of packets in a **mesh network** [13].

In an IEEE 802.15.4 network there are two main topologies. The star-network topology is the first, where all data communication between the nodes crosses a central coordinator node. The second topology is the basic peer-to-peer (P2P), where any node may interact with any other node. The P2P topology may be expanded to the mesh topology [12].

A great advantage of the mesh network is that every node can interact with any other node, even if it is not in range, by hoping the data via nearby nodes. Even if a node is disabled, in a mesh network there are usually alternate paths, which increases the reliability of the network [12].

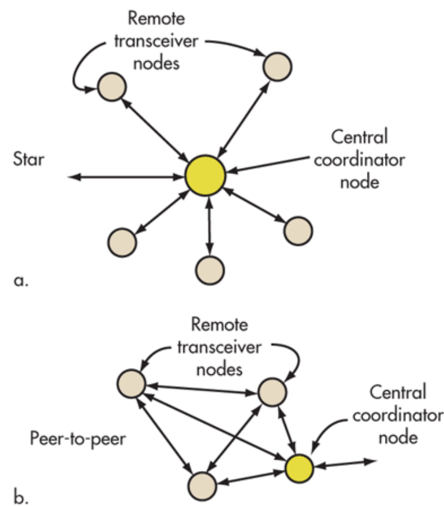

Figure 12: A star network (a) and a P2P network (b) (source: [12])

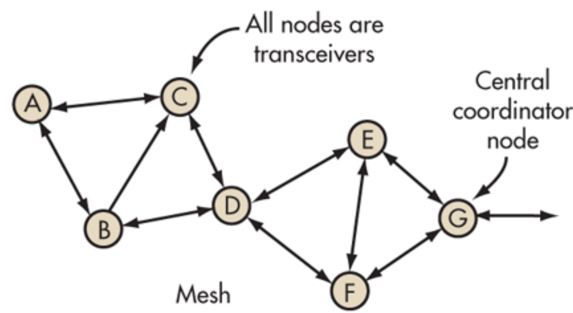

Figure 13: A mesh network. Node "A" can communicate with node "F" using two paths, e.g. ABDF, ACDF (source: [12])

There are other higher-level (OSI layer 3) protocols that make use of the 802.15.4 protocol, besides 6LoWPAN. **Zigbee** and **Bluetooth Low Energy (BLE)** are the most important. In fact, Zigbee and BLE can be used on top of 6LoWPAN, so we can have 6LoWPAN-over-BLE and 6LoWPAN-over-Zigbee communication.

Zigbee is an open-source, low-cost, low-power networking standard that targets battery-powered devices, aiming in low-latency bi-directional communication in a wireless mesh network. It is being developed, maintained and supported by an organization called "the Zigbee alliance".

The OSI layer 3 protocol of the Zigbee is responsible for the channel management, offers mechanisms for device-discovery / device-pairing and power saving, enables secure communication and multiple data packet transmissions, and a thin layer that enables a star topology network [14].

At the application layer (OSI layer 4) there are enhancement features about the authentication with valid nodes, encryption and security, the routing of the data, as well as forwarding capabilities for mesh networking [12].

The Zigbee network consists of two node types, the target node and the controller node. The controller node can be a member in multiple PANs, whereas the target node can be controlled by many controller nodes and supports inter-PAN communication. The maximum range that Zigbee may achieve is approximately 300 meters with line-of-sight (LoS) [16].

Digi International created Zigbee modules and shields for the Arduino (or Arduino-like) platform, it calls its modules **"XBee"**. Digi International created its own pin layout for the XBee module, which has become a standard too, like the Arduino pin layout. The library for the module is open-source and can be downloaded using the Arduino IDE. Once it is downloaded, the user is given a plethora of examples to study, and build applications upon them.

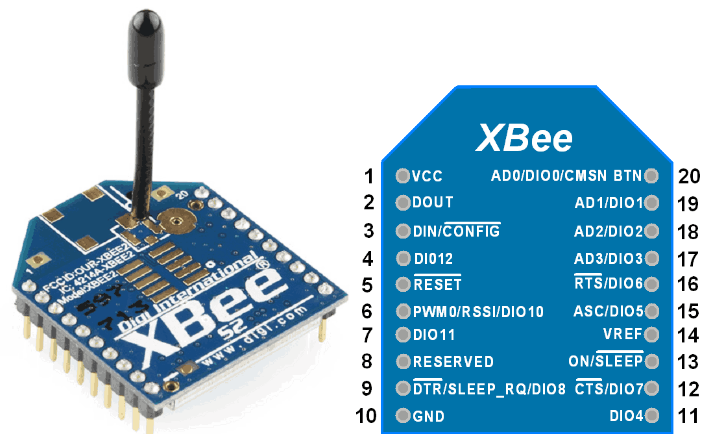

Figure 14: An XBee module alongside its antenna, and its pin layout (source: http://www.electronicwings.com/sensors-modules/xbee-module)

As seen in figure 13, the XBee module has power supply pins (VCC, GND), data in / data out pins (UART communication), a reset pin and a sleep pin, flow control pins, digital and analogue GPIO, and indicator lines [15].

Libelium's Waspmote has an extra pin layout ready to accept the XBee modules, as it is a platform that aims rapid prototyping for IoT and sensor-node applications.

The second commonly used wireless technology is the **BLE**. It is a wireless PAN technology created by the Bluetooth Special Interest Group (Bluetooth SIG), aiming to vastly reduce the energy consumption of the wireless communication while staying in the same range as normal Bluetooth.

BLE operates in the 2.4 GHz ISM band, like classic Bluetooth, but uses 40 2-MHz channels instead of 79 1-MHz channels. Within a channel, the data are transmitted at a 1 Mbit / sec rate using Gaussian Frequency Shift-Key (GFSK) modulation. The connection time is just a few milliseconds. Like Zigbee, BLE also supports mesh networks [17]. The maximum range that BLE may achieve is approximately 100 meters [16].

1.4.2 LoRa technology

LoRa (abbreviation of "Long Range") is a 2012 patented wireless technology developed by French company Cycleo, and bought by Semtech.

This technology uses licence-free sub-Gigahertz bands (169 MHz, 433 MHz and 868 MHz in Europe, 915 MHz in North America) for long range transmissions (approximately 15 Km in rural areas) while achieving low-power consumption. LoRa is in the first layer of the OSI model (physical layer).

The MAC layer is called LoRaWAN, and it is an open bi-directional Low-Power WAN (LPWAN) protocol for the IoT, offering high capacity, low power, and long range for the node that uses it. The protocol also allows reliable message delivery (via confirmations), end-to-end encryptions for security, over-the-air registration of nodes, and multicast capabilities [18].

The maximum data rate is 50 kbps in Europe and 21.9 kbps in North America, and the modulation that is used is a variant of Spread Spectrum Modulation (SSM), called Chirp Spread Modulation (CSM), where a chirp signal is used to encode the data [19].

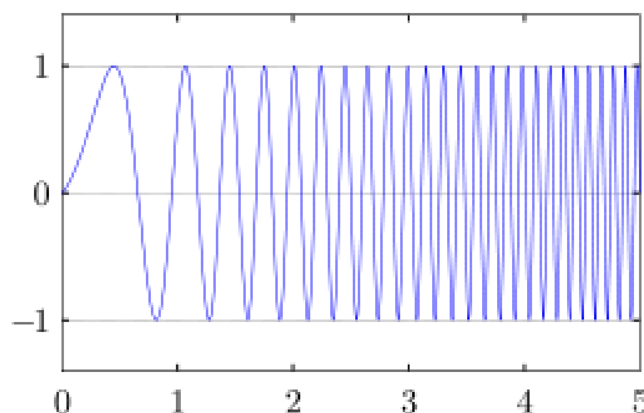

Figure 15: A chirp signal. A chirp spread modulation is a wideband frequency modulated sinusoidal signal that increases or decreases over time (source: [19])

Following the general trend, there are many LoRa shields and modules for all open-source platforms, to help creating a fast prototype.

1.4.3 MQTT protocol

MQTT (Message-Queueing Telemetry Transport) is an open-source, lightweight publish-subscribe (pub-sub) messaging protocol, which works on top of the TCP/IP protocol. It is designed and optimized for machine-to-machine and mobile applications that require a small footprint [21]. The MQTT can be used on top of 6LoWPAN to help devices communicate the data over IPV6 addresses. MQTT-S is a variant of MQTT that is used for communication over Bluetooth.

The MQTT topology requires a server, which is called "broker", and clients that communicate with the broker. A client can either be a subscriber to information or publish information to subscribed clients. Information is classified in topics. Whenever a publisher wishes to send data, a control message which also contains the data is sent to the broker. The broker is responsible for the distribution of the information to any subscriber that is interested in the topic [22].

If the broker receives data in a topic, and there are no subscribers for it, the information will be discarded, unless the publisher gives an indication that the topic has to be retained. This allows new publishers to receive the latest data, instead of waiting for the new data from the publisher. During the first connection to the client, the publisher may set up a default message that can be sent to the subscribers if the broker recognizes that the publisher had an unexpected network-disconnection from the broker [22].

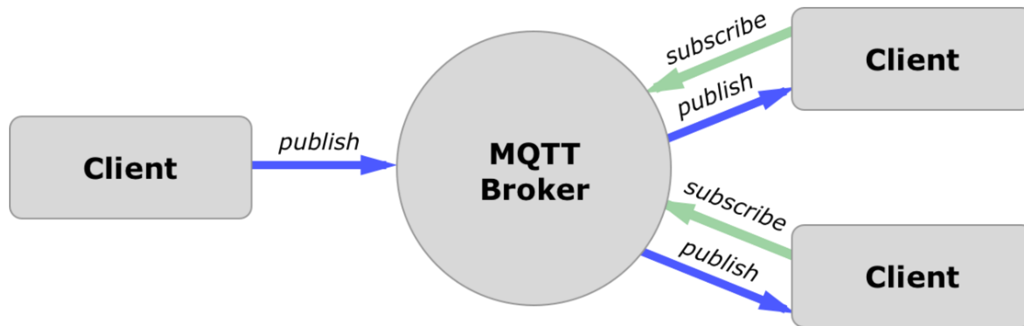

Figure 16: Basic MQTT topology. The client to the left publishes data to a topic to the broker. The clients that are subscribed to the topic will receive the data from the broker (source: https://spindance.com/smart-consumption-smart-products-understanding-mqtt/)

A control message may be as small as two bytes, and nearly 256 MB if necessary. There are fourteen message types that can be used by the client for connection and disconnection from the broker, for acknowledging the reception of data, for supervising the connection between the client and the broker [22].

Since MQTT sends connection credentials in a plain-text format, the authentication of security is not included in the protocol. Security must be provided in the TCP/IP layer, to protect the integrity of the information [22].

1.4.4 IoT gateways

Generally, a gateway is a device that serves the connection between the cloud (or the internet) and an end-device, using more than one telecommunication protocols.

Gateways are used to collect the data from IoT and sensor-node applications and push them to the cloud, as well as to receive commands and data from the cloud and reroute them to the devices.

The most used architectures for data exchange protocols are the bus-based and the broker-based. Bus-based protocols include **DDS**, **REST** and **XMPP**, while the broker-based include **AMQP**, **CoAP**, MQTT and **JMI**. The AMQP, MQTT, JMS and REST protocols are

classified as message-centric, and the DDS, XMPP and CoAP protocols are data-centric protocols [20].

Libelium has created its own gateway for IoT, called **Meshlium**. It supports XBee, WiFi, Ethernet, GPS, 4G/3G/GPRS/GSM, and LoRa (without any LoRa Alliance features) protocols. Another commonly used gateway is "**the Things gateway**" from the Things network, which supports LoRaWAN, XBee, BLE, WiFi, and Ethernet protocols.

Figure 17: Two commonly used IoT gateways. Libelium's Meshlium to the left, the Things gateway to the right

CHAPTER 2:
Projects on urban air-quality monitoring

2.1 Measuring the quality of the air

This chapter presents some projects about air-quality monitoring in an urban area, mostly using public-transportation infrastructure, alongside proposals from researchers about commonly-faced problems in such applications.

This research focused on the technology that each project used, both on hardware and on software, and its results. The outcome of this research will be used as a basis for creating an IoT application that will measure the quality of the air, using sensor-nodes, on top of public-transportation buses in the city of Athens, Greece.

Measuring and monitoring the quality of the air has become a necessity recently, especially in urban areas with high population-density, where the levels of air pollution are constantly increasing. Air pollution is a severe threat for the human health, as it has been linked with the appearance of various diseases like asthma, lung cancer etc. and particularly for the sensitive groups (children, elderly, pregnant women), according to the World Health Organization (WHO) [23].

Moreover, air pollution is responsible for 7 million deaths of people, annually (12.5% of total deaths), and this effect is more hazardous in developing countries where the processes of industrialization and high-density-urbanization are in progress, alongside the rapid increase of vehicles that use internal combustion engines in those countries [24].

There are many countries around the world that have already established air-quality stations in cities, to monitor the air pollution and inform their citizens about the concentration of various air pollutants, like carbon dioxide (CO_2), sulfur dioxide (SO_2), nitrogen dioxide

(NO_2) etc. Although these stations are usually equipped with high-end equipment, the distance between the stations is usually large, hence the information they provide fail to adequately depict the quality of the air in real time. Furthermore, the cost for the installation and maintenance of the equipment is usually great [23].

Figure 18: An air-quality monitoring station in Hong Kong (source: http://www.atimes.com/article/serious-air-pollution-tuen-mun-tung-chung/)

An alternative solution to monitor the air-quality can be provided by the **Wireless Sensor Networks** (**WSNs**). A WSN is a dense wireless network of sensor-nodes which can collect and distribute data, and its applications include environmental monitoring, disaster management, surveillance, and medical diagnosis amongst others [23].

WSNs have a great advantage, being low-cost and lightweight. They can be attached on top of vehicles and gather data from its sensors throughout the route, hence monitoring the quality of the air in areas that a station cannot reach, providing a clearer picture of the pollution. Yet, there are challenges that have to be addressed to be able to talk for a reliable solution. For example, due to the low-cost nature of the design the sensors are often unreliable or drift frequently. Other issues involve power management, communication, and

data integrity [23]. Some of the projects that will be discussed involve proposals for these issues.

2.2 IoT projects for air-quality measurement using stationary sensor-nodes

IoT and WSN-based projects are emerging increasingly as alternative solutions for measuring and monitoring the quality of the air [24]. As discussed earlier, sensor-nodes have great advantages, like their low-cost, small size, small energy consumption, and are easy to be reprogrammed.

Researchers propose that sensor-nodes will play an important role in the future smart cities. A swarm of nodes can be installed in cities and provide real-time monitoring of the air-pollution, even in remote areas, using the mesh architecture. The data can then be either viewed by any citizen, real-time, using a smartphone or a PC, or be processed by an organization that will make forecasts about the progress of the air-pollution.

Given the recent advances in low-power electronics and sensor technologies, the sensor-nodes can function with a few rechargeable batteries and an energy-harvesting module (e.g. solar panel). They can be installed in light poles, leaving space for other urban infrastructure.

Figure 19: A Waspmote sensor-node by Libelium, can be easily installed in a light pole (source: http://www.libelium.com/products/plug-sense/technical-overview/)

2.2.1 Microcontroller-based projects

Pieri and Michaelides [23] created a solution that features 10 **Waspmotes** from Libelium as sensor-nodes, all installed in an area of a city in Cyprus that approximately covers 1 km^2. Each sensor-node had sensors for measuring the temperature, humidity, noise, luminosity, CO_2, CO, NO_2, O_3 and PM10.

The sensor-nodes transmit their data using XBee every 30 minutes towards a **Meshlium** gateway (there are 2 Meshliums installed within the 1 km^2 area), which has an internal data capacity of 8 GB, and from there they are sent to a cloud database using **TCP/IP** communication and **MySQL** for further data processing.

The sensor-nodes achieve autonomous operation by using rechargeable batteries as power source and a solar panel to charge the batteries. The Meshlium can be powered by an Ethernet cable (**Power-over-Ethernet**, **PoE**).

The authors provide a few issues that troubled them, with the most important being the communication between the nodes and the gateway. In an urban environment there can be

communication problems mostly because many buildings are made from concrete, or losing the **Line-of-Sight** (**LoS**) from trees growing between the path. The maximum distance that they achieved was 60 meters.

Another problem was the calibration of the gas and dust sensors, as this procedure requires that the reference values have to be taken from the government and this is not always an accurate process because this implies that the readings of all sensors were the same during the calibration period, even though they were situated in different parts of the city.

Finally the authors faced the main problem of a low-cost sensor, which is reliability. Three sensors were malfunctioning and their results could not be trusted and had to be replaced.

David Chavez et al [24] in their work, created a sensor-node based on the **Arduino DUE** and both analogue and digital sensors for CO_2, NO_2, SO_2, H_2S, O_3, PM2.5, PM10, O_2, Temperature and Humidity.

In order to optimize the dynamic range of the DUE for some sensors, an interface with a few operational amplifiers was used as an extra circuit. Also, a real time clock module was used to add accurate timestamps to the samples. The data were saved in a microSD card. The board was powered by rechargeable batteries and a solar panel and they calculated that the total power consumption of the sensor-node is 1 Watt in idle state, due to the fact that the gases sensors have to be permanently on, in order to provide accurate samples.

They used XBee wireless communication to send the data to the gateways, and from there the data were uploaded to the cloud using either Ethernet or GSM. The data were then downloaded using **MySQL** and presented to the user through a **Graphical User Interface** (**GUI**).

Benammar et al [33] created a solution that features the PRO edition of the Waspmote from Libelium. The PRO edition offers calibrated sensors with greater precision. The board can also accept 4 more sensors, than the original one (16 and 12 respectively).

For wireless communication, the **XBee PRO** module was used, which has less power consumption than the plain XBee module and has a range of approximately 750 meters. The gateway was a **Raspberry Pi 2 model B** which had an XBee PRO module attached to it. The internet connectivity is achieved via an Ethernet cable.

Their solution puts the Waspmote in sleep mode for 13 minutes. When it wakes up, it turns on the sensors and wait for 3 minutes for the sensors to warm up. Then it takes 5 samples from each sensor, it calculates its average, and applies a calibration algorithm. The data is then saved to an SD card and sent via XBee to the gateway. The sensors and the XBee module are the turned off, before Waspmote going back to sleep.

From the gateway, the data are uploaded to the open-source Emoncms web application using the **MQTT** protocol, via APIs, for visualization.

2.2.2 SBC-based projects

Balasubramaniyan and Manivannan [30] in their work, presented a solution based on the Raspberry Pi SBC. They used the **GrovePi+** HAT, which enables the use of low-cost Grove sensors by **seeed studio**.

The GrovePi+ HAT features an interface for 7 digital ports, 3 analogue ports, 2 I2C ports, 2 UART ports, and can be directly attached to the Pi SBC.

The Grove sensors that were used for their project were the following:

- DHT11, for temperature and humidity

- MQ-5, for H_2, LPG, CH_4, CO and alcohol detection
- MQ-7, for CO detection
- MQ-135, for CO_2

The libraries for the sensors are open-source and written in **Python**. The authors created a script also in Python, on the Raspberry Pi, to control the sampling of the sensors. The control script automatically starts after the Pi has finished booting. Another script, also in Python, checks the network connectivity and it re-establishes it if lost. The WiFi chip of the Pi was used to achieve communication with the internet and the cloud services.

The cloud-services provider that they chose is **ThingSpeak**, an open-source IoT cloud platform, which apart from remote data monitoring also offers data analytics and integrated real-time support from **MATLAB**, via APIs.

Figure 20: The GroovePi+ HAT embedded on top of a Raspberry Pi SBC. The HAT provides an interface for connecting Groove sensors to the Pi (source: http://wiki.seeedstudio.com/GrovePi_Plus/)

The obvious disadvantage of this proposal is the power consumption of the Pi, alongside the use of the WiFi chip. This sensor-node would last for a few days, if not hours, if the control script is not driving the Pi to sleep mode regularly, and turning off the WiFi when it is not needed.

For an application that will be installed on top of public-transportation infrastructure, it might be possible to tackle this obstacle if the Pi gets some power from the vehicle, and use a 3G / 4G HAT to enable internet access.

Desai and Alex [31] used the **Beaglebone Black** low-cost SBC, which features a 1 GHz ARM Cortex-A8 processor, 512 MB RAM, USB port, HDMI port, 92 GPIO, and can boot Linux in less than 10 seconds.

The Beaglebone has also inbuilt Analogue-to-Digital Converters (ADCs), so it ca be interfaced both with analogue and digital sensors. Their solution included a CO sensor, a CO_2 sensor, and a GPS module that can be used to add a timestamp on the samples, as well as seeing the coordinates of the samples on the map.

After sampling the sensors, the sample is calibrated and stored locally in a database on the Beaglebone. The data are uploaded to **Microsoft's Azure Cloud** using **SQL** for Python, whenever the Beaglebone boots or every 12 hours. With Azure, a **Machine Learning** tool can be used to train the data and cast predictions about the progress of air-pollution.

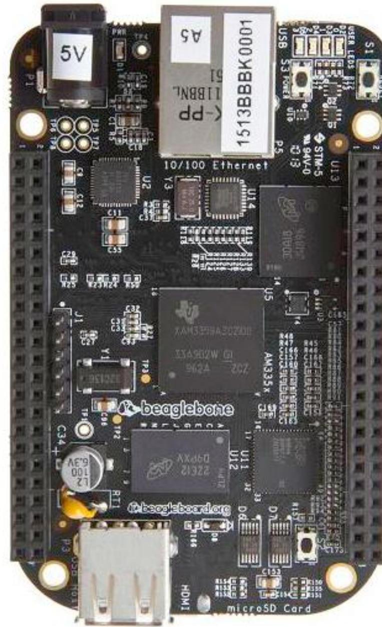

Figure 21: The Beaglebone Black SBC (source: https://beagleboard.org/black)

2.3 IoT projects for air-quality measurement using public-transportation infrastructure.

The idea of a smart vehicle, one that can gather data for its behaviour and upload them to a cloud, is not new and there are companies that offer this option in many of their models. Recently, it has been suggested that vehicles can be also used as being "sensors" themselves [32].

Since public-transportation buses move around the city on a constant basis, these vehicles can be used to gather samples about the environment, the quality of the road or the status of the traffic flow.

There several advantages for mounting the sensor-nodes on public-transportation vehicles, instead of simply attaching them to light poles. Firstly, fewer sensor-nodes can be used to measure a great area, since they will gather data from different points of a route and not from a static region. Furthermore, the equipment is not left unattended, but rather it has the protection of the bus. Also, more sensors and buses can always be added for the coverage

to be increased. Finally, by carefully picking bus lines we can achieve a real-time detailed map of the pollution of a city [32].

2.3.1 Microcontroller-based projects

Kang et al [32] created a sensor-node based on the **CC2530** by Texas Instruments. They attached sensors, a WiFi module and a GPS module, and implemented a **Zigbee PRO** protocol on the CC2530. They also added a camera that is used to capture pictures or video.

The measurements from the sensors are transmitted to the gateway using the Zigbee protocol, whereas the pictures and videos via WiFi when the bus finds a WiFi spot available. Sensor data can be transmitted from bus to bus as well, due to the mesh support of a Zigbee network. Every sensor-node is on sleep mode and it wakes up periodically every 5 minutes to perform its sampling.

The data from the gateway are pushed to a cloud-based database that the authors created in **Java**. Each sample has a timestamp as well as the latitude and longitude coordinates of the bus the moment that the sample was taken. The user can see on the cloud application a map with the route the bus has followed, as points, with each point depicting a measurement. With a simple click on a point the user receives the latest information about the quality of the air, as well as a table with some of the previous samples.

Wong et al [34] in their project, installed a sensor node on public buses in Singapore. The sensor-node featured a **Renesas H8S** microcontroller, a GPRS module, a WiFi module, a GPS module, and the air-quality sensors (CO, Methane, Iso-Butane, Ethanol, Hydrogen, Diesel & Gasoline, and Particle Pollutants).

Each sample is given a timestamp and coordinates, to be easier to see on the map the time and place that each sample was taken. The data can be transmitted either via **GPRS / 3G**

or WiFi to their database, where are processed and plotted on **Google Earth**. The user can see on which part of the route the air-pollution was high and where it was low. As expected, they found out that in urban environments the quality of the air is not as good as in less sparsely populated areas.

Gao et al [35] in their Mosaic project, which is based on Arduino UNO, discovered that the airflow disturbances that are created by the movement of buses, greatly reduce the reliability of the readings of the sensors.

In their work, the authors utilized samples from accelerometers and a GPS module, to derive the speed of the bus, alongside the samples of the air-quality sensors, and created an algorithm which filters the data from the sensors and makes them more reliable.

The data were uploaded to the Aliyun cloud using a GPRS / 3G, utilizing the GPRS shield that was attached to the UNO. The calibration of the raw data is done on a local PC, after being downloaded from the cloud using MySQL APIs, by using state-of-the-art techniques like **Artificial Neural Networks** (**ANN**) and **Multi-Support Vector Machines** (**SVM**). The appliance of the filtering algorithm is also done on the local machine, before sending the data to a GUI [36].

Reshi et al [37] in their 2013 project called VehNode, created a sensor-node based on the **Atmega 328P-PU** microcontroller, on a GPRS module and on a GPS module. The sensors that were used were the following:

- MQ-135, to measure NOx, Benzene and CO2
- MQ-7, to measure CO

The samples are given a timestamp, taken from the GPS module, and transmitted to the internet using the GPRS module. The user can then receive back his device an alert if the vehicle produces a lot of pollution.

Although their design was not meant to be used solely by public-transportation infrastructure, it was worth mentioning their approach as it can be used by any vehicle.

2.3.2 SBC-based projects

No SBC-based sensor-nodes for embedment in public transportation infrastructure projects were found. This is not a surprise, given that SBCs spend more energy than a microcontroller.

If the sensor-node had the possibility to receive power from the bus, then it would have been viable to have SBC-based solutions, which will add greater data processing and stability to the solution.

2.4 Cloud-based improvements

Wei Wang et al. [29] proposed the distribution of the sensor data computing to a fog of smaller devices, like gateways, smart phones, or edge clouds ("**fog**"), to decrease to size of the data that are used by algorithms, and to provide high-quality data for the data centres. Being usually close to the data consumers, these devices may provide an improved quality of service and reduce the latency.

Their research emphasize the ever-increasing amount of data that IoT devices produce, and that in the future even the state-of-the-art data centres will face a serious problem with their big data processing techniques. The intelligent distribution of the

computation to the fog has the potential to mitigate this problem and establish the foundation for data processing that is application-independent.

In order to distribute the computation of the data produced by WSNs, the authors used **semantic technologies** to associate each node with nearby gateways. The gateway will then be responsible for distributing the data to other devices and services.

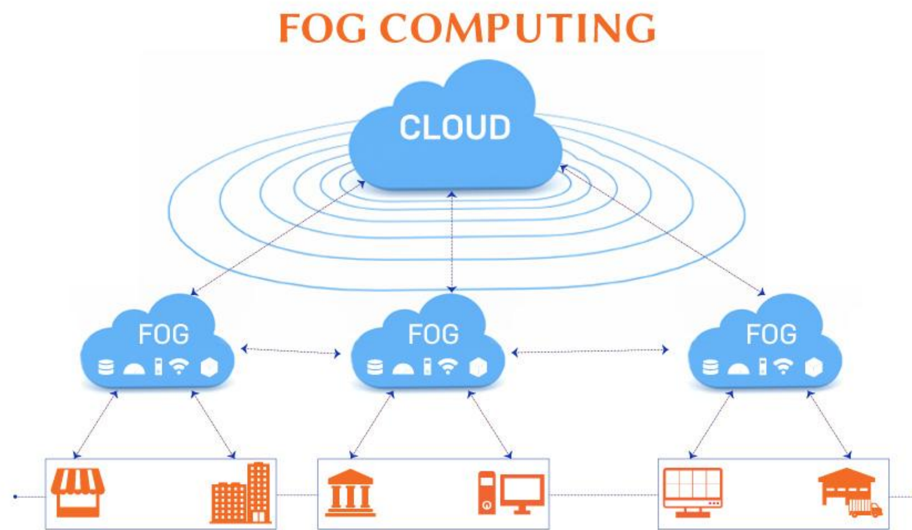

Figure 22: Topology of fog computing (source: http://www.securens.in/securens_blog/demystifying-fog-computing-and-its-role-in-e-surveillance-part-1/)

CHAPTER 3:
Proposed WSN for air-quality monitoring

3.1 WSN specifications

This chapter presents the specifications of the WSN that will be installed on public buses, as well as the hardware, and subsequently the software, which were chosen for the task. The previous chapters provided the foundation that the proposed WSN was built upon, giving an insight in technologies and trends that researchers are using globally.

The most important aspect that has to be considered, is the energy availability. It is unknown if the public transportation company will allow modifications on its buses to allow the WSN to receive power from the bus, hence some source of **energy-harvesting** must be used to provide power to the node.

Furthermore, a low-power microcontroller has to be used in order for the application to spend as less energy as possible when it is awake, making sure that the node's needs will be covered by the energy-harvesting module. It is thus important that the WSN spend as less time awake as possible, only waking up when it is time to gather samples for the quality of the air.

This leads to another aspect that has to be considered, the technology that will be used for the wireless transmission of the data. The WSN hast to use a low-power wireless technology and make sure that it does not transmit when it is unnecessary.

Moreover, the reliability and integrity of the data have to be considered. The WSN must gather accurate samples and make sure that they will not get lost, if for example the node runs out of energy or restarts.

Finally a minor aspect that, although is not necessary, will add scalability to the node is the open-source nature of the project, using development boards that follow pinout standards. Using the plug-and-play interface that these boards provide, it is not difficult to swap shields and modules and re-program the WSN over-the-air (**OTA**).

3.2 Hardware chosen for WSN Autonomy

The autonomous use of the node will be achieved by combining energy sources, low-power hardware and communication technologies, and an algorithm that can provide reassurance that the hardware does not spend unnecessary energy.

From the specifications mentioned above, and based on the research about IoT development platforms that was presented on the first chapter (table 1), the Waspmote from Libelium was chosen as the node. The Waspmote offers low-power capabilities that are of great importance in WSN applications, it comes with a plethora of shields that target specific IoT areas (agriculture, air-quality, water-quality etc.) and with certified sensors. Libelium also provides the user with APIs and examples that help with rapid development and prototyping; for every sensor or shield there is an API, which significantly reduces the time required to produce code.

With the Waspmote as basis, the **Waspmote Sensor Gases Version 3.0 Board** (see figure 6) was chosen as the shield that will be plugged on top of the Waspmote and host the sensors. The shield can have as many as 7 sensors working simultaneously, sensing fourteen types of gases as well as environmental parameters like the temperature, the humidity and the atmospheric pressure [38]. There is also a PRO edition of the gases shield that accept seventeen different types of calibrate sensors, but for a higher price [39]. The V 3.0 board uses non-calibrated sensors, and to achieve more accurate results Libelium provide its users

with a calibration library that can be used to calibrate the sensors with software. To ensure the integrity of the data, an SD card will be plugged to the Waspmote to help with storing the samples. This way, the data will not get lost even after a restart of the application.

As power sources, a rechargeable battery of 6600 mAh and a 25 mm X 55 mm solar panel will be used. The Waspmote offers an interface to plug both these sources. The main power source will be the battery, and the solar panel will be used to charge the battery when the node is in sleep mode. Since Libelium has stopped supporting an official solar panel for its boards, a solar panel from Panasonic was bought (AM-1801CA) instead.

The chosen solar panel does not have an interface for the Waspmote, so one had to be created. Since Waspmote can be charged from both the solar panel interface and the USB port, a USB cable was modified to act as a charging cable. The Data+ and Data- wires were cut out, and the Vcc and GND wires were attached to the solar panel pins.

For the wireless technology, a Zigbee module will be used which will operate on the 868 MHz band, to follow the European regulations. The Waspmote has an XBee interface that allows a plug-and-play approach. Zigbee ensures low-energy consumption, compared to WiFi and GPRS, and acceptable coverage (approximately 100 meters without LoS, can reach 8 km with LoS). LoRa modules were not chosen as there is no official support from Libelium. The XBee module can be seen in figure 23. The XBee interface on the Waspmote, can be seen at the right side of the development board.

Figure 23: The Waspmote (left), the XBee module (right) and its antenna (top) (source: https://www.cooking-hacks.com/Waspmote-868-sma-4-5-dbi)

XBee will transmit the sampled data to a Meshlium gateway at the University of West Attica. From there, a cloud-based application from Libelium will be used to display the data to the user.

The hardware units were assembled and installed in a modified plastic box, which will provide the node with immunity to vibrations caused by the bus, to dust, to moisture and generally to adverse weather conditions. The assembled node can be seen in figures 24, 25 and 26.

The modifications enabled the GPS and the XBee antennas to stay out of the box, to achieve optimal communication. The GPS module was installed in a small plastic enclosure as suggested by the manufacturer, for electromagnetic shielding. The solar panel was attached to the top of the box, and a hole was drilled in order for the Vcc and GND wires to reach the USB charging cable. Finally, an area was removed from the top of the box, so the sensors have exposure to the environment. A dust filter was installed above the removed plastic area, to protect the board from dust.

Figure 24: Top-side view of the node. The dust filter enables the sensors to have exposure to the environment.

Figure 25: Top view of the node.

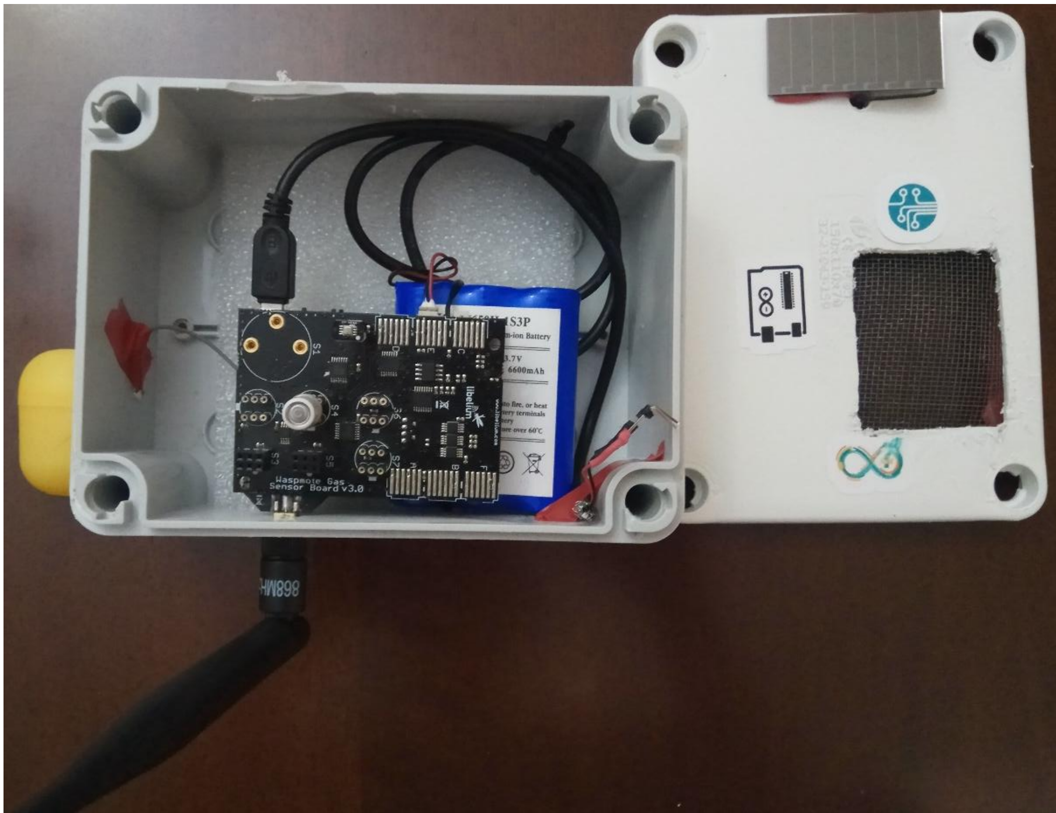

Figure 26: Inside view of the box. A soft layer under the Waspmote helps facing the vibrations that will be caused by the vehicle.

3.3 Software for minimal energy consumption

The source code was written in the Arduino-based IDE for Libelium's products, in C++. The code aims to support long sleep times, where all peripherals and modules are closed, and as short awake times as possible. In the awake state, each module or peripheral opens when it is necessary, and after its use it shuts down immediately, to preserve energy. The flow chart of the source code can be seen in figure 27.

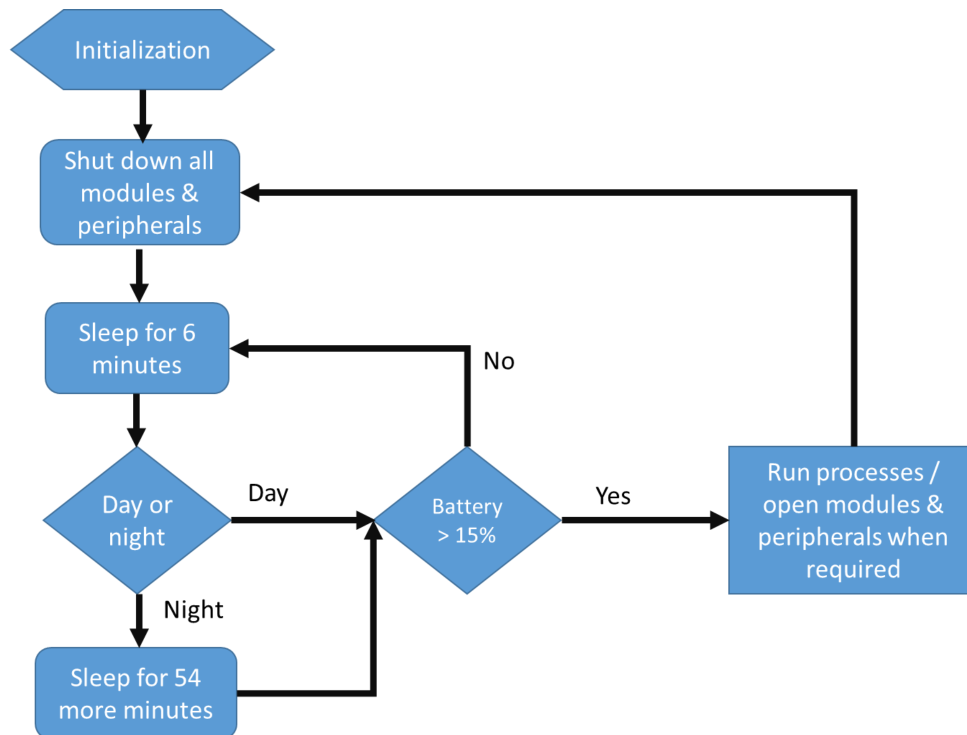

Figure 27: flow chart of the code that ensures low-power use of the node

The node goes to sleep for six minutes immediately after its initialization. When wakes up it checks whether it is day or night, using the **real-time clock** (**RTC**) that the Waspmote offers. If it is night, the node will sleep for another 54 minutes (one hour in total). If it is day, and there is enough power available on the battery (at least 15%), then the main application of sampling can be called. The same will happen in night-time, after the one-hour sleep, the node will check for the energy availability and will proceed in the main code execution if it is adequate.

During the initialization state, the Waspmote calls functions to setup the SD card, the GPS and RTC modules, and the sensors of the application, since the non-calibrated sensors are used. Next, the function declarations of the initialization state will be presented, alongside their inputs and outputs (where applicable). The complete source code can be found at **appendix A**, where the reader is also given a **Github** repository link to download the code.

```
/* Function: Setup_App_Sensors
 * ------------------------------------
 *    Calculates the logarithmic functions that are going to be used by non-linear
sensors
 *   Input:  none
 *   Output: none */
void Setup_App_Sensors (void)
```

```
/*Function: Setup_GPS_And_RTC
 * -----------------------------------
 *   Starts the GPS and stores the Ephemeris information to the SD card so it can
 *   start faster during the application. Sets the real time and date to the RTC
 *   Input:  none
 *   Output: none */
void Setup_GPS_And_RTC (void)
```

```
/* Function: Setup_SD_Card
 * ------------------------------
 *   Deletes old files and creates new files for logging the data from the sensors
```

```
 *   and the GPS
 *   Input:  none
 *   Output: none */
void Setup_SD_Card (void)
```

After the initialization, the application will put the device to sleep mode and follow the algorithm that the flow chart that is depicted on figure 27. The flow chart of the main application is depicted in figure 28.

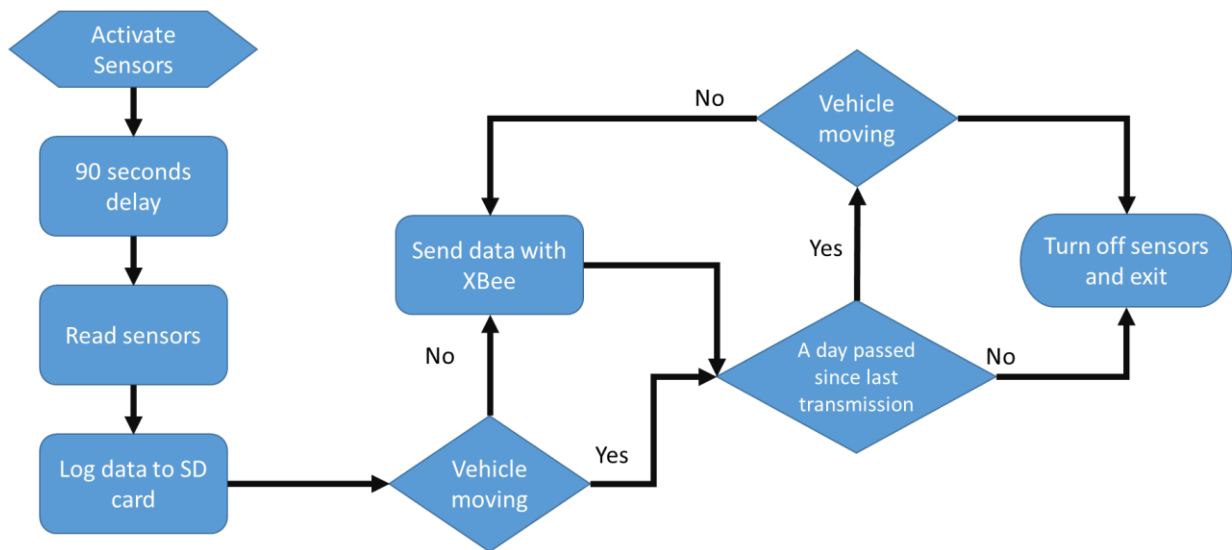

Figure 28: Flow chart of the node's main process

When the node wakes up from sleep mode, it will execute the actions that are described in figure 28. The function that calls all this actions is called Run_application

```
/* Function: Run_Application
 * --------------------------------
```

```
 *   Starts the sensor board. Starts the sensors and wait 90 seconds for them to
 *   warm up. Reads samples from sensors, stores them to frames and send them
 *   via XBee if necessary.
*   Input:  none
 *   Output: none */
void Run_Application (void)
```

Then, the Run_Application function will call the following functions:

```
/* Function: Read_Sensors
 * -----------------------------
 *   Reads all the sensors of the application, and stores their values to global
 *   variables (CO2, NO2, Temperature etc)
 *   Input:  none
 *   Output: none */
void Read_Sensors (void)
```

```
/* Function: Print_Sensors_Values
 * --------------------------------------
 *   Prints the values of the sensors to Serial Monitor, if we are in debug mode
 *   Input:  none
 *   Output: none */
void Print_Sensors_Values (void)
/* Function: Create_Ascii
```

```
 * ---------------------------
 *   Add the values of the sensors to the frame, and write the frame to the SD card
 *   Input:  none
 *   Output: none */
void Create_Ascii (void)
```

```
/* Function: Read_GPS_Coordinates_And_Send
 * -------------------------------------------------------
 *   Parses the SD card for the GPS coordinates that need to be sent to the Meshlium,
 *   and sends them using XBee
 *   Input:  index
 *           Indicates the line of the file containing the coordinates
 *   Output: none */
void Read_GPS_Coordinates_And_Send (int index)
```

```
/* Function: Read_Logged_Data_And_Send
 * ------------------------------------------------
 *   Parses the SD card for the values of the sensors that need to be sent to
 *   the Meshlium, and sends them using XBee
 *   Input:  index
 *           Indicates the line of the file containing the coordinates
 *   Output: none */
void Read_Logged_Data_And_Send (int index)
```

```
/* Function: Check_The_Date
 * --------------------------------
 *   Checks if a day has passed since we started the application. Checks
 *   if it is day or night outside
 *   Input:  none
 *   Output: TRUE    if a day has passed
 *                FALSE   if no day has passed  */
bool Check_The_Date (void)
```

```
/* Function: Send_Data
 * ---------------------------
 *   If the bus is not moving, send frames via XBee
 *   Input:  none
*    Output: none */
void Send_Data (void)
```

```
/* Function: Ninety_Seconds_Delay
 * --------------------------------------
 *   waits 90 seconds for the gases sensors to warm up
 *   Input:  none
 *   Output: none */
void Ninety_Seconds_Delay (void)
```

```
/* Function: See_Battery_Status
 * -----------------------------------
 *   If we are in debug mode, print to the serial monitor the status of the battery
 *   Input:  none
 *   Output: none */
void See_Battery_Status (void)
```

Each of the aforementioned function was separately tested before getting integrated in the code. This method is known as unit testing, and there are frameworks and automated tools for various programming languages. Unfortunately, Libelium's IDE does not support any of these tools and frameworks, so the testing was done manually.

Once the functions were tested and integrated to the code, the complete application had to be tested. Since the sampling frequency is extremely low (6 minutes during the day, 60 minutes during the night), the period was lowered in order to observe the behaviour of the system. The daytime sampling during the testing process occurred every 32 seconds, while the night-time sampling every 80 seconds.

The variables that set the sampling frequency during the day or the night are defined in a header file that is included in the source code. They can be easily modified according to the needs of every user and sent to the node over-the-air, as an update. It is possible that one node may have different sampling periods than others, if the OTA update targets only this node.

In that header file there are also variables that are used by the software calibration process, during the initialization of the application. These variables can also be modified and sent wirelessly to the node to ensure that the sensors are always calibrated.

CHAPTER 4:
Results of application

4.1 Solar panel efficiency

For the solar panel, it was calculated that the **Maximum Power Point** (**MPP**), which can be found taking into account the 80% of the maximum voltage and maximum current, is around 5.1 mWatts.

For the calculations, alongside the solar panel, a 10 Ω resistor and a multimeter are needed. The tests took place in a sunny August noon at the city of Athens, Greece. Experiments indicated that an average voltage of 5.9 volts can be produced during a sunny day. Then using the resistor in series with the panel and the multimeter, the current during the sunny day was measured to be approximately 1.35 mA.

The formula to calculate the MPP is then:

$$E = (0.8 * V) * (0.8 * I)$$

By replacing the measured voltage and current in the formula, the MPP was calculated to be 5.1 mWatts.

To test the power efficiency of the solar panel, the node was programmed to be on sleep-mode constantly. The test occurred during an afternoon, and the node was logging the battery status to the SD card every 60 seconds. The test lasted for one hour, and the results can be seen in figure 29.

As seen in figure 29, there were periods that the battery was charging, periods that the battery was discharging, and periods that the battery remained at the same level. During the periods that the battery was charging, the sun was clear and the solar panel produced high voltages, whereas when the battery was discharging or remained stable, the sun was covered by clouds or the solar panel was in the shade.

The test began with the battery having 82% available power, and it ended after one hour with the battery still having 82% availability, proving that the solar panel can help charge the battery. The test required the node to wake up, enable the SD card, and log the frames that contain the battery info in the SD card, a process that requires some significant energy. Given that the sampling of the battery's status occurred every minute, we can safely assume that the battery will charge sufficiently during large periods of sleep mode.

In order to get the percentage of available power of the battery, an API from Libelium was used. APIs was one of the major reasons that the Waspmote was chosen, since they help with rapid prototyping. The API that was used is **PWR.getBatteryLevel()**, which returns a string that contains the remaining percentage. In order to add this information to the frame and save it to the SD card, the API **frame.addSensor(SENSOR_BAT, PWR.getBatteryLevel())** was used.

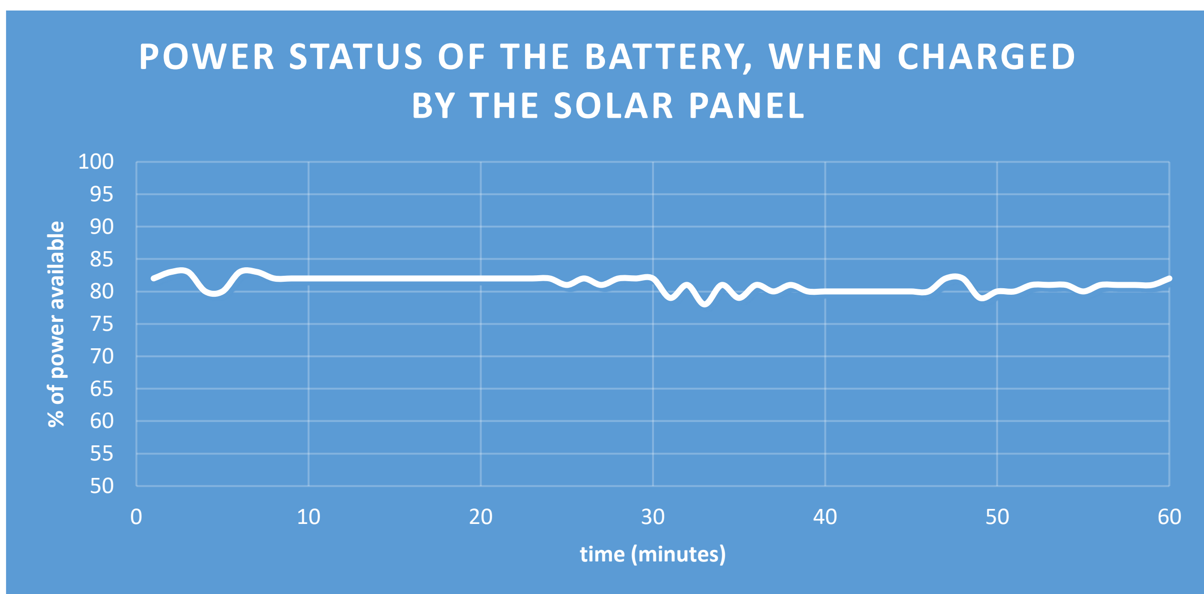

Figure 29: Charging / discharging status of the battery, using the solar panel

4.2 Code efficiency

As discussed earlier, the code was designed with low-power consumption in mind. For that purpose, the node stays in sleep mode for 6 minutes during the morning and for 60 minutes during the night, before waking up to gather samples.

To test the effect of the code on the battery, two tests were run; two during the morning and one during the night. After each sampling period, the status of the battery was logged to the SD card. The results can be seen in figures 30 and 31.

For the sampling process, the following APIs were used:

- **Gases.ON()**, which switches on the sensors shield
- **COSensor.ON()**, which enables the CO sensor
- **GPS.ON()**, which turns the GPS on
- **SD.ON()**, which turns the SD card on
- **Gases.getTemperature()**, which samples the temperature sensor
- **Gases.getHumidity()**, which samples the humidity sensor
- **Gases.getPressure()**, which samples the pressure sensor
- **COSensor.readConcentration()**, which samples the CO sensor
- **SD.OFF()**, which turns the SD card off
- **GPS.OFF()**, which turns the GPS off
- **PWR.sleep(WTD_8S, ALL_OFF)**, which turns off every peripheral and sets the node to sleep mode for 8 seconds

For the logging process, the following frame APIs were used:

- **frame.createFrame(ASCII)**, which creates a new ASCII frame
- **frame.addSensor(SENSOR_GASES_CO, COPPM)**, which adds the CO data to the frame
- **frame.addSensor(SENSOR_GASES_TC, temperature)**, which adds the temperature data to the frame
- **frame.addSensor(SENSOR_GASES_HUM, humidity)**, which adds the humidity data to the frame
- **frame.addSensor(SENSOR_GASES_PRES, pressure)**, which adds to the pressure data to the frame
- **frame.addTimestamp()**, which adds the timestamp to the frame
- **frame.addSensor(SENSOR_GPS,GPS.convert2Degrees(GPS.latitude, GPS.NS_indicator),GPS.convert2Degrees(GPS.longitude, GPS.EW_indicator))**, which adds the coordinates to the frame
- **memset(toWrite, 0x00, sizeof(toWrite))**, which initialises a buffer "toWrite" that will store the frame
- **Utils.hex2str(frame.buffer, toWrite, frame.length)**, which writes to the "toWrite" buffer the frame that contains the data
- **SD.appendln(filename, toWrite)**, which appends the frame "toWrite", to the .txt file "filename"

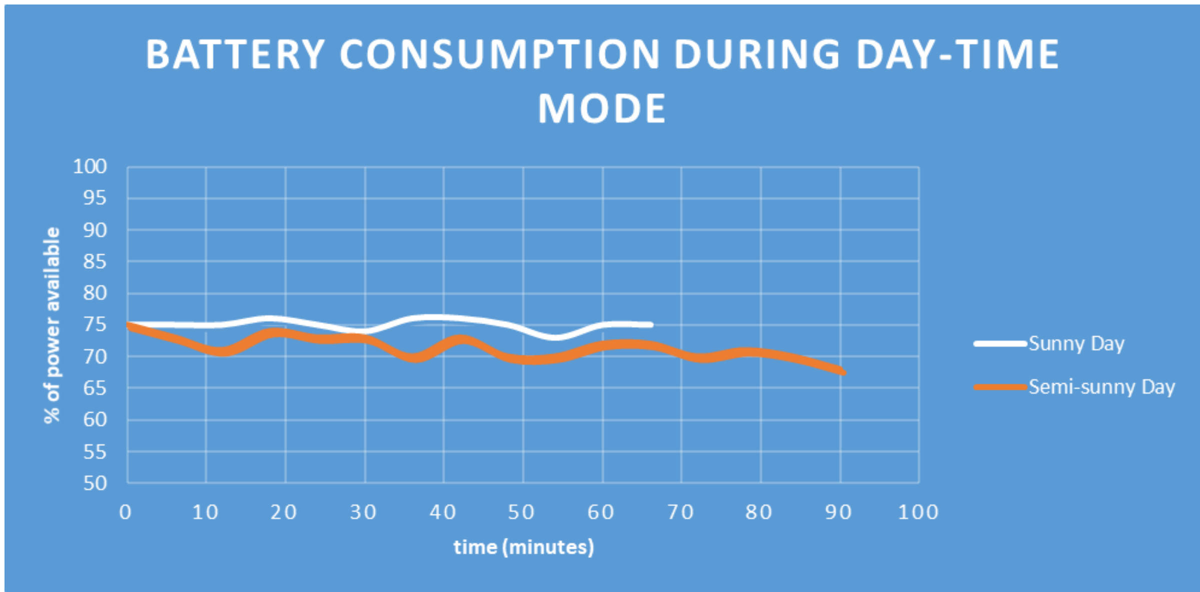

Figure 30: Percentage of the battery's available after 2 tests, during the day-time mode

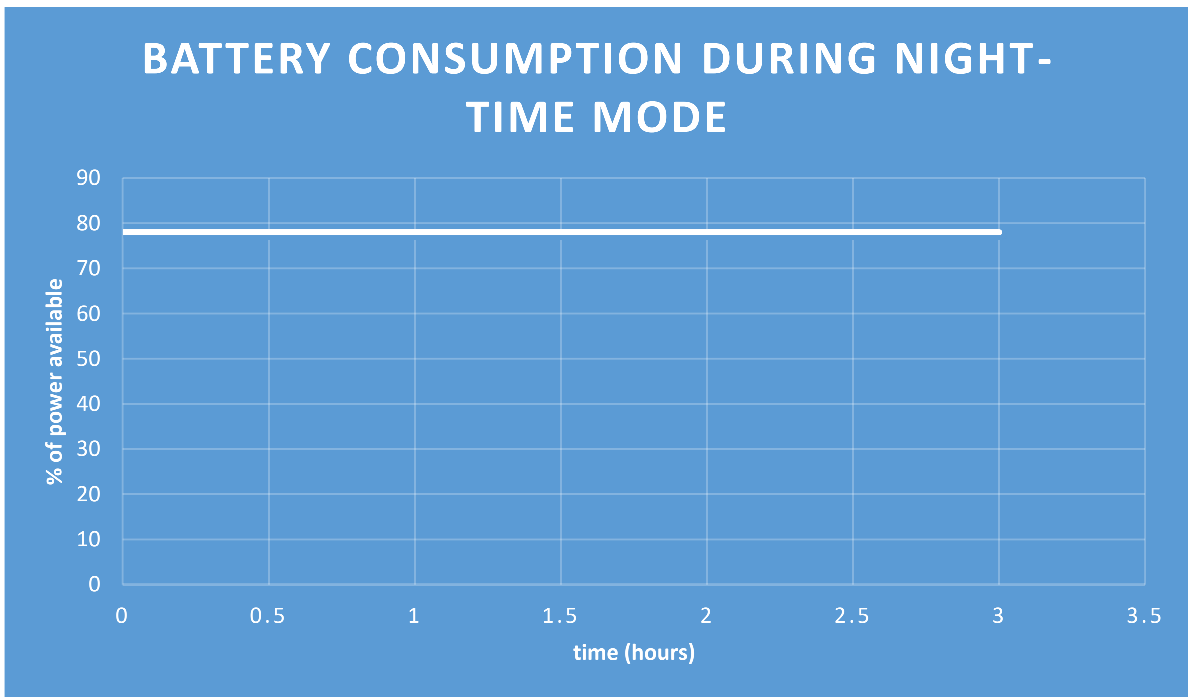

Figure 31: Percentage of the battery's available power after 3 hours, during the night-time mode

In figure 30, we see the results from the two tests; one test ran during a sunny day and one during a semi-cloudy day. We can see periods that the battery was charging, even if the sampling occurs every 6 minutes. When the battery is discharging, the clouds were covering the sun, thus preventing the solar-panel from charging efficiently while the node was under the sampling and logging processes. Also, the discharging of the battery is faster when the GPS takes a lot of time to connect to the satellites. The GPS has to establish connection with the satellites within 4 minutes, and this hard time-limit can be modified in the source code according to the needs of the user.

As seen in figure 31, the battery remained stable throughout the three-hour duration of the nightly test. This justifies the selection of the sampling period during the nights, as the node will not spend a lot of power.

It must be mentioned here though that the tests did not include communication with the Meshlium, only the sampling of the sensors and the logging processes occurred.

4.3 XBee communication

The ZIgbee protocol requires that the frame that will be transmitted must be smaller than 100 bytes. During the testing period, the node had a CO sensor, a temperature-pressure-humidity sensor, and a GPS module. When the samples of the sensors and the coordinates taken from the GPS were added to the frame that was about to be sent to the Meshlium, the communication failed because the size of the frame was greater than 100 bytes.

For example, one frame that was constructed after a sampling looked like this:

```
<=>†#3F383F057C105428#node1_data#0#CO:741.947#TC:30.03#HUM:43.4#PRES:10086
5.02#TST:1536253581##GPS:37.977802;23.349562#
```

Where:

- <=> is a three-byte delimiter that identifies the start of every frame
- Three more bytes follow, one that defines the type of the frame, another that defines the number of fields, and the separator (#)
- 3F383F057C105428 is a 16-byte serial ID
- node1_data is the ID of the node (10 bytes)
- 0 is a one-byte frame sequence indicator
- And then we see the samples of CO, temperature, humidity, pressure and the coordinates from the GPS

The size of this frame is 120 bytes, and the communication failed when the node tried to transmit it.

The solution that was given is to have two different logging files, one that stores a frame containing the samples and the other a frame containing the coordinates of the place that the sample occurred.

Every time that the node has to transmit data, it will first send the frame that contains the samples and then the frame that contains the GPS coordinates. The Meshlium will understand that these two transmissions correspond to one sampling period from the node ID and the timestamp, which will be the same on both frames.

The solution was tested successfully, two frames were sent, one containing the node ID, the samples and the timestamp, and the other containing the same node ID, the GPS coordinates and the same timestamp. The Meshlium captured both transmissions and updated successfully the information on the web-based service.

The frames look like the following example:

Data frame

```
<=>†#3F383F057C105428#node1_data#0#CO:1044.025#TC:41.87#HUM:0.3#PRES:10151
1.95#TST:1536683577#
<=>†#3F383F057C105428#node1_data#2#CO:1509.284#TC:53.72#HUM:0.0#PRES:10157
6.81#TST:1536678994#
<=>†#3F383F057C105428#node1_data#4#CO:1509.284#TC:53.21#HUM:0.0#PRES:10156
0.66#TST:1536679534#
<=>†#3F383F057C105428#node1_data#6#CO:1438.019#TC:52.47#HUM:0.0#PRES:10153
1.19#TST:1536680074#
<=>†#3F383F057C105428#node1_data#8#CO:1388.411#TC:52.88#HUM:0.0#PRES:10152
5.77#TST:1536680614#
<=>†#3F383F057C105428#node1_data#10#CO:1372.525#TC:52.72#HUM:0.0#PRES:1015
26.39#TST:1536681154#
<=>†#3F383F057C105428#node1_data#12#CO:1264.694#TC:51.90#HUM:0.0#PRES:1015
13.83#TST:1536681694#
```

<=>†#3F383F057C105428#node1_data#14#CO:1145.652#TC:51.00#HUM:0.0#PRES:1015

11.23#TST:1536682235#

GPS frame

<=>†#3F383F057C105428#node1_data#1#GPS:37.986301;23.343319#TST:1536683577#

<=>†#3F383F057C105428#node1_data#3#GPS:37.986397;23.343294#TST:1536678994#

<=>†#3F383F057C105428#node1_data#5#GPS:37.986382;23.343298#TST:1536679534#

<=>†#3F383F057C105428#node1_data#7#GPS:37.986389;23.343294#TST:1536680074#

<=>†#3F383F057C105428#node1_data#9#GPS:37.986393;23.343327#TST:1536680614#

<=>†#3F383F057C105428#node1_data#11#GPS:37.986359;23.343275#TST:1536681154#

<=>†#3F383F057C105428#node1_data#13#GPS:37.986423;23.343302#TST:1536681694#

<=>†#3F383F057C105428#node1_data#15#GPS:37.986454;23.343281#TST:1536682235#

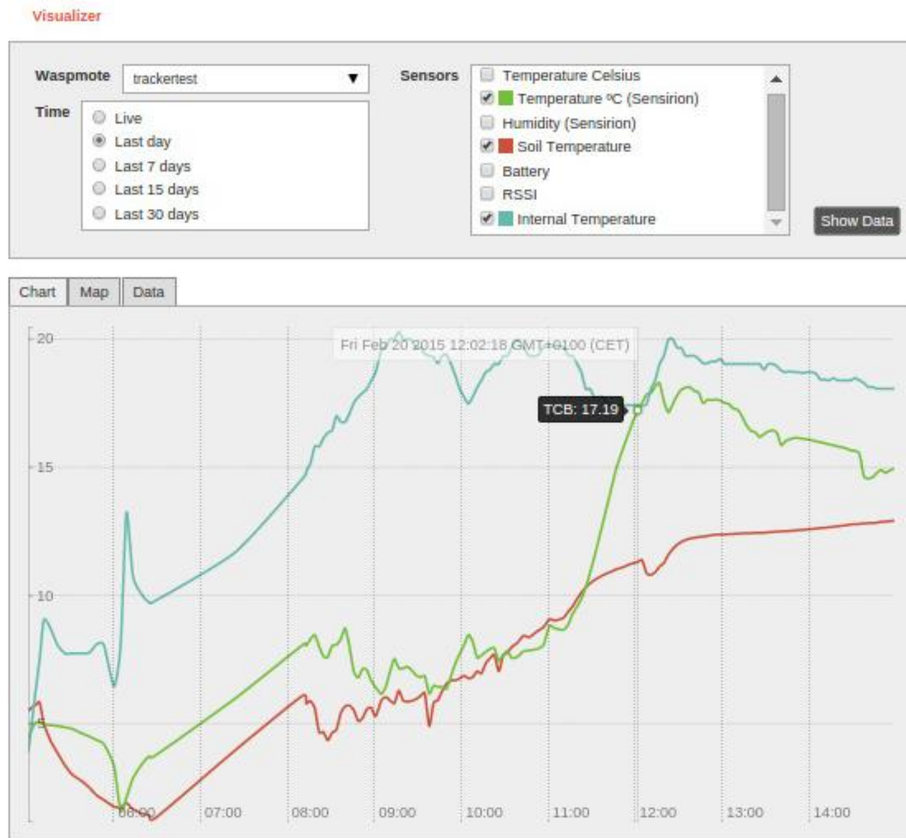

Figure 32: Meshlium visualizer tool. The data that have been captured will be illustrated as time-series charts (Source: [40])

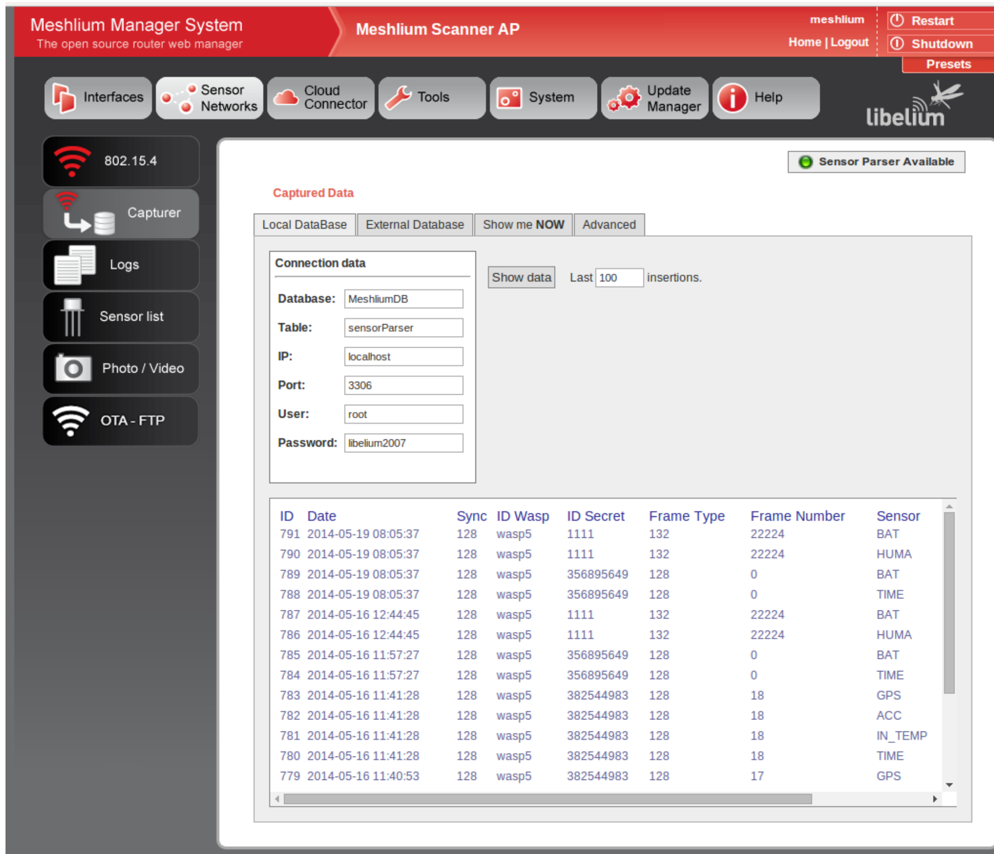

Figure 33: The cloud application of Meshlium. Data transmitted to the gateway will be parsed and illustrated here.

As seen from these examples, every even-numbered frame carries the samples taken from the sensors, while every odd-numbered frame carries the GPS coordinates from the place that the measurement took place. Hence, even if the cloud-based application from Liblium had not been used, a simple application in any programming language could have been created to parse the incoming frames and store them in a database. The parsing should scan for an even sequence indicator, n, and for the odd sequence indicator n+1. The data from the sensors would be extracted from the even number and the GPS data from the odd number. For verification, the timestamp of these two consecutive frames must be the same.

It must be noted that the size of the first frame is close to the hard limit of 100 bytes. It is therefore suggested that if more sensors will be added in a future edition, the samples of these sensors have to be added to the frame of the GPS coordinates, since it has a smaller size.

The node was used both indoors and outdoors. Indoors, the communications is heavily affected by the infrastructure of the building, where thick blocks of concreate can block the communication. In a lab environment, the maximum distance that was measured was 50 meters. Whereas, when the node was outdoors and there was line of sight with the Meshlium, the maximum distance was measured to be approximately 3000 meters.

4.4 GPS reliability

As seen from the previous examples, the frames carried information about the coordinates where the sampling occurred. The coordinates that the GPS sampled are accurate, being just a couple of meters away from the actual position.

There were a few connectivity issues though, when the GPS could not establish communication with the satellites within four minutes. These connectivity issues occurred mostly indoors or just outside a building. When the node was in an open field or in a wide road, the connection was usually established within 45 seconds.

The frames need to carry information about the samples and the coordinates, thus if the GPS communication cannot be established, the sampling will not occur and the

frame will be filled with zero values in both the data from the sensors and the coordinates.

CHAPTER 5:
Analysis of the results

The results that were extracted from the tests, which were presented in chapter 4, prove that the proposed WSN is capable of autonomously functioning in an urban environment, being attached on a public transportation vehicle.

Even during a cloudy day, the node only consumed 5% of the available power. During a relatively sunny day, there was no consumption, as the battery was able to get charged by the solar panel.

Initially, the node was set to understand that the day-time mode lies between 6 AM and 7 PM, hence sampling every 6 minutes for 13 hours. Given that during a cloudy day, the battery may consume 5% of the available power within an hour, in 13 hours the battery could consume 5 % * 13 = 65 % of the available power.

For this reason, the day-time mode was reprogrammed to lie between 8 AM and 8 PM, where the traffic is usually heavier than the other hours of the day, hence providing more useful data about the quality of the air, and leaving the Waspmote in sleep mode a bit more, so it can recharge.

Also, a condition which checks for the availability of the power before entering the main application was used. If the available power is less than 15%, the main process will not be executed, hence the node will continue to sleep until the battery gets recharged.

The problem with the big frames that were not possible to be sent to the Meshlium using the Xbee, as there is a restriction that allows up to 100 bytes to be sent, was solved by splitting the frame into 2 frames, which will be identified as one by the Meshlium using the common timestamp that they share.

Furthermore, it must be noted that when the XBee was set to broadcast mode, there was a significant loss of captured packets by the Meshlium, which Bennamar also observed [33]. In unicast mode, such a loss was not observed.

During the first stages of the development, the node had a strange behaviour, which was causing it to restart constantly. After some research it was found that the battery has to be always connected to the node when using a shield or a wireless communication module, as the USB does not provide adequate power. The reader has to keep that in mind for future projects.

The manufacturer of the GPS module informs us that there are three modes for booting up the GPS. The cold start, where the GPS takes approximately 35 seconds to start, as it is not provided with the almanacs or the ephemeris. The warm start, that takes 30 seconds if the GPS is given the ephemeris. And the hot start, where the GPS is provided with both the almanacs and the ephemeris.

In order to save time from starting the GPS, the warm start method was chosen. After the initial boot of the GPS, the ephemeris were saved into the SD card so the next time that the GPS had to start, it would take less time. Unfortunately the tests were not successful, the GPS did not seem to connect faster to the satellites, or sometimes the

ephemeris were not correctly saved so the GPS was driven to a timeout when it tried to reconnect. For this reason, the cold start was finally chosen, though in the source code there are remnants of the warm start, in case the viewer wants to test it.

The results that were achieved are not easily comparable with the projects that were mentioned in chapter 2, as the vast majority of the projects use WSNs on non-moving stations.

Regarding the energy consumption, the results can be compared with Pieri's and Michaelides' solution for a stationary WSN, using the Waspmote, where their battery did not drop below 75% [23]. Their node was in sleep mode for thirty minutes though, with the node sampling the sensors upon waking up and sending the data immediately to the gateway, before going back to a thirty-minute sleep session. The authors faced the same communication problem using the XBee, without LoS, which in their case could approximately reach 60 meters.

Bennamar et al [33] set their Waspmote into a fifteen minutes sleep state, before waking up to read the samples. The authors do not provide with energy consumption information though, due to the fact that the node is powered from the electricity sockets, as its purpose is for indoors air-quality monitoring.

Wong et al [34] created one of the first projects about WSN attached to vehicles in 2009, where autonomy was not a big issue, hence they do not provide us with energy consumption information. They used GPRS and Bluetooth modules for transmitting the

data to the gateway, which leads us to assume that their node would probably spend more energy than the proposed WSN of this study.

Kang et al [32] set their nodes into five-minute-long sleep sessions, and they did not use an energy-harvesting module. Every node starts in sleep, and the gateway will wake up for the sampling process. When the battery runs out of energy, the gateway will understand its absence and it will wake up another node to continue sampling.

This study focused on the autonomy of the node, as it was not clear whether or not the transportation company would allow modifications on the buses that will allow the node to have constant power. For this reason, the Waspmote was set on long sleep sessions, with every module, peripheral and shield being shut down. When the node wakes up, it has to power on again the shield, the sensors, the SD card, the GPS and the XBee and this sometimes can cause high energy consumption, especially when the GPS cannot establish connection to the satellites quickly.

Nonetheless, this study proves that even a small energy-harvesting module, like a 25 mm X 55 mm solar panel, is enough to charge the battery during the six-minute-long sleep sessions. The capability to understand whether the node is functioning in the morning or in the night helped preserving even more battery power. The condition that maintains the node to sleep mode until the battery has adequate power ensures that at least one measurement will be taken from the sensors every day.

CHAPTER 6:
Conclusion - Proposals

This study proposed a WSN-based solution, which will be attached on top of public transportation infrastructure, for measuring the quality of the air in cities. The reader is introduced initially to the Internet of Things, sensor-nodes, and wireless sensor networks notions, presenting the key technologies behind hardware, software and telecommunications that led to their development.

These notions will help transform cities to smart-cities; urban areas that will consume less energy, spend just enough natural resources, produce less pollution, and help its citizens in every possible way – from transportation to weather forecasting.

One very important service for the citizens is the real-time monitoring and prediction of the quality of the air inside the city, and the timely warning if the pollution of the air is increasing. Given that nearly the one eight of deaths annually is caused by the air-pollution, one can truly understand the important of this service.

The focus has been switched to low-power, autonomous, wireless sensor-nodes, that can easily be installed on buildings, vehicles and other public infrastructure. Up to recently, the majority of the research was about stationary sensor-nodes and methods of improving their autonomy.

A new trend is the embedment of these sensor-nodes to public buses. This solution can provide better monitoring of the progress of the air pollution, since the sensors are not stationary, but rather they move within the city. Thus they can sense areas that a stationary sensor would not reach.

This study proposed a low-power microcontroller-based development platform, a low-power wireless telecommunication technology, and programming techniques in order to preserve energy.

The node can understand if it is functioning in day-time mode or in the night-time mode, adjusting the sampling period based on this mode. This adjustment of the sampling frequency helped in lowering the energy consumption of the node during the night, when the solar panel cannot charge adequately the battery.

Future studies might expand this function, by making the node understand the season of the year and the luminosity of the sun. The GPS can be used to gather information about the month, hence adjusting the 8 AM to 8 PM window, which was set to be the day-time mode. For example, during the winter, the time window can be set from 9 AM to 5 PM. A luminosity sensor can be used to sense if the sun is covered by thick clouds, hence providing a more dynamic sampling window, based on the sun’s light. The luminosity sensor from Libelium occupies the same slot with the temperature-pressure-humidity sensor though, so the researcher must know whether or not the information about the temperature, the pressure, and the humidity are valuable to him/her, or use a luminosity sensor that is not supported by Libelium.

Furthermore, if the node has access to a constant energy source, for example from the bus, the processes that shut down every module and sensor can be neglected, saving a lot of time waiting for the GPS to connect to the satellites, or filling the frames with zero values when the GPS fail to connect.

Finally, for this study, a plastic box was drilled and modified in order to create a prototype that can safely host the node while being able to get attached on a bus. The box

took some time to get modified, thus in a future project it might be better if a box was designed in a CAD program and 3D-printed, which will lead in saving time from handcrafting and making it easier to be mass-produced.

Although there are not as many researches and projects for these mobile sensor-nodes, as there are for the stationary ones, we can expect that in the future more papers and articles will appear and offer hardware, software and telecommunication proposals for smart-monitoring the air quality in an urban environment.

References

1. Domingo, M. G., & Forner, J. M. (2010). Expanding the learning environment: combining physicality and virtuality - The Internet of Things for eLearning. 2010 10th IEEE International Conference on Advanced Learning Technologies, 730-731.
2. International Telecommunication Union. (retrieved 2018, August 8). Internet of Things Global Standards Initiative. Retrieved from www.itu.int: https://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx
3. Kushner, D. (2011, October 26). The Making of Arduino. Retrieved from https://spectrum.ieee.org: https://spectrum.ieee.org/geek-life/hands-on/the-making-of-arduino
4. Wikipedia. (retrieved 2018, August 8). Sensor Node. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Sensor_node
5. Holler, J., Tsiatsis, V., Mulligan, C. K., Avesand, S., & Boyle, D. (2014). From Machine-to-Machine to the Internet of Things: Ontroduction to a New Age of Intelligence. Oxford: Academic Press.
6. Geekbench Browser. (retrieved 2018, August 19). iPhone X Benchmarks. Retrieved from Geekbench Browser: http://browser.geekbench.com/ios_devices/52
7. www.wikipedia.org. (retrieved 2018, August 19). Internet of things. Retrieved from www.wikipedia.org: https://en.wikipedia.org/wiki/Internet_of_things
8. Camarinha Matos, L. M., Tomic, S., & Graca, P. (2013). Technological Innovation for the Internet of Things, 4th IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2013. Springer.
9. www.st.com. (retrieved 2018, August 22). B-L475E-IOT01A. Retrieved from www.st.com: https://www.st.com/en/evaluation-tools/b-l475e-iot01a.html
10. Mulligan, G. (2007). The 6LoWPAN architecture. EmNets '07 Proceedings of the 4th workshop on Embedded networked sensors (pp. 78-82). Cork, Ireland: ACM.
11. Kushalnagar, N., Montenegro, G., & C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals", RFC
12. www.electronicdesign.com. (retrieved 2018, August 23). What’s The Difference Between IEEE 802.15.4 And ZigBee Wireless? Retrieved from Electronic Design: https://www.electronicdesign.com/what-s-difference-between/what-s-difference-between-ieee-802154-and-zigbee-wireless
13. Ott, A. (2012). Wireless Networking With IEEE 802.15.4 and 6LoWPAN. Embedded Linux Conference – Europe, November 5, 2012.
14. Zigbee Alliance. (retrieved 2018, August 27). ZigBee RF4CE: A Quiet Revolution is Underway. Retrieved from web.archive.org: https://docs.zigbee.org/zigbee-docs/dcn/12/docs-12-0629-01-0mwg-zigbee-rf4ce-a-quiet-revolution-is-underway-webinar-slides.pdf
15. Wikipedia. (retrieved 2018, August 27). XBee. Retrieved from www.wikipedia.org: https://en.wikipedia.org/wiki/XBee
16. Ray, B. (retrieved 2018, August 27). ZigBee Vs. Bluetooth: A Use Case With Range Calculations. Retrieved from LinkLabs: https://www.link-labs.com/blog/zigbee-vs-bluetooth
17. Wikipedia. (retrieved 2018, August 27). Bluetooth Low Energy. Retrieved from www.wikipedia.org: https://en.wikipedia.org/wiki/Bluetooth_Low_Energy
18. Prajzler, V. (retrieved 2018, August 27). LoRa, LoRaWAN and LORIOT.io. Retrieved from LORIOT.io: https://www.loriot.io/lorawan.html

19. Miron, R. (retrieved 2018, August 27). Develop with LoRa for low-rate, long-range IoT applications. Retrieved from MWee: http://www.mwee.com/design-center/develop-lora-low-rate-long-range-iot-applications/page/0/1
20. Schneider, S. (retrieved 2018, August 28). What's the Difference between Message Centric and Data Centric Middleware? Retrieved from ElectronicDesign: https://www.electronicdesign.com/embedded/whats-difference-between-message-centric-and-data-centric-middleware
21. Stanford-Clark, A., & Linh Truong, H. (2013, November 14). MQTT For Sensor Networks (MQTT-SN) Protocol Specification.
22. Wikipedia. (retrieved 2018, August 28). MQTT. Retrieved from www.wikipedia.org: https://en.wikipedia.org/wiki/MQTT
23. Pieri, T., & Michaelides, M. P. (2016). Air Pollution Monitoring in Lemesos using a Wireless Sensor Network. Proceedings of the 18th Mediterranean Electrotechnical Conference.
24. David Chavez, M., River Quispe, T., Jorge Rojas, M., Andres Jacoby, K., & Guillermo Garayar, L. (2015). A Low-Cost, Rapid-Deployment and Energy-Autonomous Wireless Sensor Network for Air Quality Monitoring. 2015 Ninth International Conference on Sensing Technology.
25. Upton, E. (2018, March 14). RASPBERRY PI 3 MODEL B+ ON SALE NOW AT $35. Retrieved from Raspberry Pi: https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/
26. Cellan-Jones, R. (2011, May 5). A 15 pound computer to inspire young programmers. Retrieved from BBC: http://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2011/05/a_15_computer_to_inspire_young.html
27. Wikipedia. (retrieved 2018, August 29). Raspberry Pi. Retrieved from www.wikipedia.org: https://en.wikipedia.org/wiki/Raspberry_Pi#cite_note-5
28. Brown, E. (2018, june 28). Raspberry Pi 3 B+ wins hacker board reader survey. Retrieved from linuxgizmos.com: http://linuxgizmos.com/raspberry-pi-3-b-wins-hacker-board-reader-survey/
29. Wang, W., Dey, S., Zhouy, Y., Huang, X., & Moessner, K. (2017). Distributed Sensor Data Computing in Smart City Applications. 2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)
30. Balasubramaniyan, C., & Manivannan, D. (2016). IoT Enabled Air Quality Monitoring System (AQMS) using Raspberry Pi. Indian Journal of Science and Technology, Vol 9(39)
31. Desai, N. S., & Alex, J. S. (2017). IoT based air pollution monitoring and predictor system on Beagle bone black. 2017 International Conference on Nextgen Electronic Technologies: Silicon to Software (ICNETS2)
32. Kang, L., Poslad, S., Wang, W., Li, X., Zhang, Y., & Wang, C. (2016). A Public Transport Bus as a Flexible Mobile Smart Environment Sensing Platform for IoT. 2016 12th International Conference on Intelligent Environments
33. Benammar, M., Abdaoui, A., H.M. Ahmad, S., Touati, F., & Kadri, A. (n.d.). A Modular IoT Platform for Real-Time Indoor Air Quality Monitoring. Sensors 18(2), 581
34. Wong, K.-J., Chua, C.-C., & Li, Q. (2009). Environmental Monitoring using Wireless Vehicular Sensor Networks. 5th International Conference on Wireless Communications, Networking and Mobile Computing
35. Gao, Y., Dong, W., Guo, K., Liu, X., Chen, Y., Liu, X., & Chen, C. (2016). Mosaic: A low-cost mobile sensing system for urban air quality monitoring. IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications
36. Dong, W., Guan, G., Chen, Y., Guo, K., & Gao, Y. (2015). Mosaic: Towards City Scale Sensing with Mobile Sensor Networks. 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)

37. Reshi, A. A., Shafi, S., & Kumaravel, A. (2013). VehNode: Wireless Sensor Network platform for automobile pollution control. 2013 IEEE CONFERENCE ON INFORMATION AND COMMUNICATION TECHNOLOGIES
38. Cooking-hacks. (retrieved 2018, September 7). *Waspmote Gases Sensor Board*. Retrieved from www.cooking-hacks.com: https://www.cooking-hacks.com/Waspmote-gasses-sensor-board
39. Cooking-hacks. (retrieved 2018, September 7). *Waspmote Gases Sensor Board PRO* . Retrieved from www.cooking-hacks.com: https://www.cooking-hacks.com/Waspmote-gases-sensor-board-pro
40. Libelium. (2017). *Waspmote technical guide v7.4.* Libelium Comunicaciones Distribuidas S.L.

Appendix A: Source Code

The Waspmote uses 2 files to get programmed. The first file, air_quality.pde hosts the state machine and the functions that are used by it. The second file, coefficients.h, hosts the definitions of the variables that are used by the application.

The source code can also be downloaded from the following Github link: https://github.com/nikmonios/IoT_Urban_Air_Quality

coefficients.h

```
#ifndef COEFFICIENTS_H
#define COEFFICIENTS_H

/* Concentrations used in calibration process (PPM Values) */
#define POINT1_PPM_CO2 350.0  //   <-- Normal concentration in air
#define POINT2_PPM_CO2 1000.0
#define POINT3_PPM_CO2 3000.0

#define POINT1_PPM_NO2 10.0   // <-- Normal concentration in air
#define POINT2_PPM_NO2 50.0
#define POINT3_PPM_NO2 100.0

/* Calibration vVoltages obtained during calibration process (Volts) */
#define POINT1_VOLT_CO2 0.300
#define POINT2_VOLT_CO2 0.350
#define POINT3_VOLT_CO2 0.380

#define POINT1_RES_NO2 45.25 // <-- Rs at normal concentration in air
#define POINT2_RES_NO2 25.50
#define POINT3_RES_NO2 3.55

#define POINT1_PPM_CO 100.0   // <--- Ro value at this concentration
#define POINT2_PPM_CO 300.0
#define POINT3_PPM_CO 1000.0

/* Calibration resistances obtained during calibration process */
#define POINT1_RES_CO 230.30 // <-- Ro Resistance at 100 ppm. Necessary value.
#define POINT2_RES_CO 40.665 //
#define POINT3_RES_CO 20.300 //

/* Define the number of calibration points */
#define numPoints 3
```

```
/**** GLOBAL & DEFINES ****/
float NO2PPM = 0;    /* Stores the NO2 value */
float COPPM = 0;    /* Stores the CO value */
float temperature = 0; /* Stores the temperature in Celsius */
float humidity = 0;     /* Stores the relative humidity in %RH */
float pressure = 0;    /* Stores the pressure in Pa */
float CO2PPM = 0; /* stores the CO2 value */

/* CO2 Sensor must be connected physically in SOCKET_2 */
CO2SensorClass CO2Sensor;
/* NO2 Sensor must be connected physically in SOCKET_3 */
NO2SensorClass NO2Sensor;
/* CO Sensor must be connected physically in SOCKET_4 */
COSensorClass COSensor;

float COconcentrations[] = { POINT1_PPM_CO, POINT2_PPM_CO, POINT3_PPM_CO };
float COres[] = { POINT1_RES_CO, POINT2_RES_CO, POINT3_RES_CO };
float NO2concentrations[] = { POINT1_PPM_NO2, POINT2_PPM_NO2,
POINT3_PPM_NO2 };
float NO2res[] = { POINT1_RES_NO2, POINT2_RES_NO2, POINT3_RES_NO2 };
float CO2concentrations[] = { POINT1_PPM_CO2, POINT2_PPM_CO2, POINT3_PPM_CO2 };
float CO2voltages[] = { POINT1_VOLT_CO2, POINT2_VOLT_CO2, POINT3_VOLT_CO2 };

/* node ID */
char node_ID[] = "node1_data";
char node_gps[] = "node1_gps";

/* a unit that updates every 8 seconds */
uint16_t timer_unit = 0;

/* 45 * 8 seconds = 360 seconds. Sampling period during the day */
uint16_t sample_time_day = 45;

/* 450 * 8 seconds = 3600 seconds. Sampling period during the night */
uint16_t sample_time_night = 450;

/* Destination Meshlium MAC address */
//////////////////////////////////////////
char MESHLIUM_ADDRESS[] = "000000000000FFFF"; /* broadcast mode */
//////////////////////////////////////////

/* define variable to check if we can communicate with Meshlium */
uint8_t error;

/* define GPS timeout when connecting to satellites */
/* this time is defined in seconds (240sec = 4minutes) */
#define TIMEOUT 240
```

```
/* define status variable for GPS connection */
bool status;

/* variables for data logging */
/* define file name: MUST be 8.3 SHORT FILE NAME */
char filename[]="FILE1.TXT";
/* define file name for GPS coordinates logging */
char filename_gps[] = "GPSDATA.TXT";

char filename_bat[] = "BATLOG.TXT";

/* buffer to write into Sd File */
char toWrite[256];

/* define variables to read stored frames */
uint8_t frameSD[MAX_FRAME + 1];
uint16_t lengthSD;
int32_t numLines;

/* variables to define the file lines to be read */
int startLine;
int endLine;

/* define variable */
uint8_t sd_answer;

/* battery level indicator */
uint8_t power_level;

/* Day or Night? for night mode */
char* daytime = "DAY"; /* initialize in day */

/* temporary day of week variable, to see if a day has passed */
char* temp_day = "Mon";

/* Indices that shows which sample inside the .txt file is next to be sent, both data and gps */
int transmission_index_data = 0;
int transmission_index_gps = 0;

/* variable to store whether we are operating during the day or during the night */
int daytime_mode;

/* from 8 AM, start taking samples every 6 minutes */
int morning_limit = 8;

/* from 8 PM, start taking samples every one hour */
int night_limit = 20;
```

```
#endif
/*************************************************************************/
```

air_quality.pde

```
#include <WaspSensorGas_v30.h>
#include <WaspFrame.h>
#include <WaspXBee868LP.h>
#include <WaspGPS.h>
#include "coefficients.h"

/*#define PDEBUG 1*/  /* enable serial print for debug reasons */
/*#define MODE2  1*/   /* mode 2, when enabled, will read more sensors */

void setup()
{
 #ifdef PDEBUG
 USB.ON();   /* configure the USB port */
 USB.println(F("Board is starting..."));
 #endif

 /* Set the Waspmote ID */
 frame.setID(node_ID);

 /* setup the sensors that will be used in this application */
 Setup_App_Sensors();

 /* setup the SD card */
 Setup_SD_Card();

 /* setup the GPS and RTC modules */
 Setup_GPS_And_RTC();

 #ifdef PDEBUG
 USB.println(F("starting application now"));
 #endif
}


void loop()
{
 PWR.sleep(WTD_8S, ALL_OFF); /* sleep for 8 seconds, kill all sensors and boards */

 #ifdef PDEBUG
 USB.ON();
 #endif

 if( intFlag & WTD_INT ) /* when the 8 seconds of sleep pass, wake up */
 {
  #ifdef PDEBUG
```

```
  USB.println(F("---------------------"));
  USB.println(F("WTD INT captured"));
  USB.println(F("---------------------"));
  #endif

  intFlag &= ~(WTD_INT); /* when you wake up, clean the flag */
 }

 timer_unit++;  /* update the timer unit every time you wake up */

 daytime_mode = strcmp(daytime,"DAY"); /* is it day or night? */

 if(daytime_mode == 0) /* if it is day , the sampling will occur every 6 minutes*/
 {
  if (timer_unit == sample_time_day) /* have we got 45 8_Sec interrupts (360 secs)? */
  {
   timer_unit = 0; /* reset the timer unit */

   /* read the battery level */
   power_level = PWR.getBatteryLevel();

   /* run the application only if there is sufficient power ( > 15% ) */
   if(power_level >= 15)
   {
    Run_Application(); /* main routine */

    /* check if battery is charging from solar panel, in debug mode */
    #ifdef PDEBUG
    See_Battery_Status();
    #endif
   }
  }
 }
 else /* else, if it is night, the sampling will occur every 1 hour */
 {
  if (timer_unit == sample_time_night) /* have we got 450 8_Sec interrupts (3600 secs)? */
  {
   timer_unit = 0; /* reset the timer unit */

   /* read the battery level */
   power_level = PWR.getBatteryLevel();

   /* run the application only if there is sufficient power ( > 15% ) */
   if(power_level >= 15)
   {
    Run_Application(); /* main routine */
   }
  }
```

```
 }

}

/***************************** FUNCTIONS *****************************/

/*
 * Function: Read_Sensors
 * ----------------------
 *   Reads all the sensors of the application,
 *   and stores their values to global variables
 *   (CO2, NO2, Temperature etc)
 *
 *   Input:  none
 *
 *   Output: none
 */
void Read_Sensors(void)
{
 /* Start the GPS*/
 GPS.ON();

 /* load ephemeris previously stored in SD */
 GPS.loadEphems();

 status = GPS.waitForSignal(TIMEOUT); /* wait some time for the GPS to connect */

 if( status == true ) /* if connection was established, read the sensors */
 {
  /**************MODE 2 ONLY********************/
  #ifdef MODE2

  /*            READ CO2               */
  CO2PPM = CO2Sensor.readConcentration();
  /*********************************************/

  /*            READ NO2               */
  NO2PPM = NO2Sensor.readConcentration();
  #endif
  /***************MODE 2 ENDS********************/


  /* READ TEMPERATURE, HUMIDITY, PRESSURE       */
  temperature = Gases.getTemperature();
  delay(100);
  humidity = Gases.getHumidity();
  delay(100);
  pressure = Gases.getPressure();
```

```
    delay(100);

    /*              READ CO                  */
    COPPM = COSensor.readConcentration();
    /*********************************************/
  }
}

/*
 * Function: Print_Sensors_Values
 * ------------------------------
 *    Prints the values of the sensors
 *    to Serial Monitor, in Debug mode
 *
 *    Input:  none
 *
 *    Output: none
 */
void Print_Sensors_Values(void)
{
  /* MODE 2 DEBUG INFO STARTS HERE */
  #ifdef MODE2

  USB.print(F(" NO2 concentration Estimated: "));
  USB.print(NO2PPM);
  USB.println(F(" ppm"));

  USB.print(F(" CO2 concentration estimated: "));
  USB.print(CO2PPM);
  USB.println(F(" ppm"));
  #endif
  /* MODE 2 DEBUG INFO ENDS HERE */

  USB.print(F(" CO concentration Estimated: "));
  USB.print(COPPM);
  USB.println(F(" ppm"));

  USB.print(F(" Temperature: "));
  USB.print(temperature);
  USB.println(F(" Celsius Degrees |"));

  USB.print(F(" Humidity : "));
  USB.print(humidity);
  USB.println(F(" %RH"));

  USB.print(F(" Pressure : "));
  USB.print(pressure);
  USB.println(F(" Pa"));
```

```
}

/*
 * Function: Create_Ascii
 * ----------------------
 *   Add the values of the sensors
 *   to the frame, and write the frame
 *   to the SD card
 *
 *   Input:  none
 *
 *   Output: none
 */
void Create_Ascii(void)
{
  /* power up SD card */
  SD.ON();

  /* Create new frame (ASCII) */
  frame.createFrame(ASCII); /* frame.createFrame(ASCII, node_ID) */

  /**************MODE 2 ONLY******************/
  #ifdef MODE2

  /* Add CO PPM value */
  frame.addSensor(SENSOR_GASES_CO2, CO2PPM);

  /* Add NO2 PPM value */
  frame.addSensor(SENSOR_GASES_NO2, NO2PPM);

  #endif
  /**************MODE 2 ENDS******************/

  /* Add CO2 PPM value */
  frame.addSensor(SENSOR_GASES_CO, COPPM);

  /* Add temperature */
  frame.addSensor(SENSOR_GASES_TC, temperature);

  /* Add humidity */
  frame.addSensor(SENSOR_GASES_HUM, humidity);

  /* Add pressure */
  frame.addSensor(SENSOR_GASES_PRES, pressure);

  /*  get the actual time from thr RTC */
  RTC.getTime();
```

```
/* add the timestamp to the samples */
frame.addTimestamp();

/* print the frame for debug reasons */
#ifdef PDEBUG
frame.showFrame();
#endif

/********* store these samples to the dedicated file ********/
/************************************************************/
/* save the frame to the SD card now */
/* init the buffer that will hold the frame info */
memset(toWrite, 0x00, sizeof(toWrite) );

/* Conversion from Binary to ASCII */
Utils.hex2str( frame.buffer, toWrite, frame.length);

/* some debug help here */
#ifdef PDEBUG
USB.print(F("Frame to be stored:"));
USB.println(toWrite);
#endif

/* now append data to file */
sd_answer = SD.appendln(filename, toWrite);

if( sd_answer == 1 )
{
  #ifdef PDEBUG
  USB.println(F("Frame WITHOUT GPS appended to file"));
  #endif
}
else
{
  #ifdef PDEBUG
  USB.println(F("Append of data WITHOUT GPS failed"));
  #endif
}

/********* store the GPS coordinates to the dedicated file now ********/
/**********************************************************************/

/* first, create a new frame that will hold the GPS coordinates */
frame.createFrame(ASCII);

/* now add the coordinates */
frame.addSensor(SENSOR_GPS,
```

```
                GPS.convert2Degrees(GPS.latitude, GPS.NS_indicator),
                GPS.convert2Degrees(GPS.longitude, GPS.EW_indicator) );

/* now add the same timestamp */
frame.addTimestamp();

/* print the frame for debug reasons */
#ifdef PDEBUG
frame.showFrame();
#endif

/* save the frame to the SD card  now */
/* init the buffer that will hold the frame info */
memset(toWrite, 0x00, sizeof(toWrite) );

/* Conversion from Binary to ASCII */
Utils.hex2str( frame.buffer, toWrite, frame.length);

/* some debug help here */
USB.print(F("Frame to be stored:"));
USB.println(toWrite);

/* now append data to file */
sd_answer = SD.appendln(filename_gps, toWrite);

if( sd_answer == 1 )
{
  #ifdef PDEBUG
  USB.println(F("Frame WITH GPS appended to file"));
  #endif
}
else
{
  #ifdef PDEBUG
  USB.println(F("Append of data WITH GPS failed"));
  #endif
}

/* close GPS - it has been on since the Read_Sensors() was called */
GPS.OFF();

/* close SD */
SD.OFF();


/* reset values */
COPPM = 0;
temperature = 0;
```

```
 humidity = 0;
 pressure = 0;

 #ifdef MODE2 /* if we work on full demo, clear other sensors */
 CO2PPM = 0;
 NO2PPM = 0;
 #endif
}

/*
 * Function: Read_GPS_Coordinates_And_Send
 * --------------------------------------
 *   Parses the SD card for the GPS coordinates
 *   that need to be sent to the Meshlium,
 *   and sends them using XBee
 *
 *   Input:  index
 *         Indicates the line of the file
 *         containing the coordinates
 *
 *   Output: none
 */
void Read_GPS_Coordinates_And_Send(int index)
{
 /* speed variable will check if the vehicle is moving or not */
 int vehicle_speed = 0;

 /* init XBee */
 xbee868LP.ON();

 /* open SD card */
 SD.ON();

 /* get number of lines in file */
 numLines = SD.numln(filename_gps);

 /* get specified lines from file
   get only the last file line*/
 startLine = index;
 endLine = numLines;

 /* iterate to get the File lines specified */
 for( int i = startLine; i < endLine ; i++ )
 {
  /* Get 'i' line -> SD.buffer */
  SD.catln( filename_gps, i, 1);

  /* initialize frameSD */
```

```
memset(frameSD, 0x00, sizeof(frameSD) );

/* conversion from ASCII to Binary */
lengthSD = Utils.str2hex(SD.buffer, frameSD );


/* not actually needed here -- just for debug */
/* Conversion from ASCII to Binary */
#ifdef PDEBUG
USB.print(F("Get previously stored GPS frame:"));
#endif

for(int j = 0; j < lengthSD; j++)
{
 #ifdef PDEBUG
 USB.print(frameSD[j],BYTE);
 #endif
}
#ifdef PDEBUG
USB.println();
#endif

/*********************************************
* At this point 'frameSD' and 'lengthSD' can be
* used as 'frame.buffer' and 'frame.length' to
* send information via some communication module
*********************************************/

vehicle_speed = (int)GPS.speed;

if (vehicle_speed < 5) /* vehicle has slowed down or not moving */
{
 /* here send the frame via XBEE */
 /* send the frame to anyone listening */
 error = xbee868LP.send( MESHLIUM_ADDRESS, frameSD, lengthSD );

 // check TX flag
 if( error == 0 )
 {
  transmission_index_gps++; /* update gps data index */
  USB.println(F("send GPS ok"));
 }
 else
 {
  USB.println(F("send GPS error"));
 }
}
else
```

```
  {
   /**** if we are moving, abort transmission ****/

   /* close SD card */
   SD.OFF();

   /* turn ZigBee off, to save power */
   xbee868LP.OFF();

   /* return to other code execution */
   return;
  }

 }

 /**** transmission ended, close modules ****/
 /* close SD card */
 SD.OFF();

 /* turn ZigBee off, to save power */
 xbee868LP.OFF();
}

/*
 * Function: Read_Logged_Data_And_Send
 * ----------------------------------
 *   Parses the SD card for the values of
 *   the sensors that need to be sent to
 *   the Meshlium, and sends them using XBee
 *
 *   Input:  index
 *          Indicates the line of the file
 *          containing the coordinates
 *
 *   Output: none
 */
void Read_Logged_Data_And_Send(int index)
{
 /* speed variable will check if the vehicle is moving or not */
 int vehicle_speed = 0;

 /* init XBee */
 xbee868LP.ON();

 /* open SD card */
 SD.ON();

 /* get number of lines in file */
```

```
numLines = SD.numln(filename);

/* get specified lines from file
   get only the last file line*/
startLine = index;
endLine = numLines;

/* iterate to get the File lines specified */
for( int i = startLine; i < endLine ; i++ )
{
  /* Get 'i' line -> SD.buffer */
  SD.catln( filename, i, 1);

  /* initialize frameSD */
  memset(frameSD, 0x00, sizeof(frameSD) );

  /* conversion from ASCII to Binary */
  lengthSD = Utils.str2hex(SD.buffer, frameSD );


  /* not actually needed here -- just for debug */
  /* Conversion from ASCII to Binary */
  #ifdef PDEBUG
  USB.print(F("Get previously stored frame:"));
  #endif

  for(int j = 0; j < lengthSD; j++)
  {
    #ifdef PDEBUG
    USB.print(frameSD[j],BYTE);
    #endif
  }
  #ifdef PDEBUG
  USB.println();
  #endif

  /***********************************************
  * At this point 'frameSD' and 'lengthSD' can be
  * used as 'frame.buffer' and 'frame.length' to
  * send information via some communication module
  ***********************************************/

  vehicle_speed = (int)GPS.speed;

  if (vehicle_speed < 5) /* vehicle has slowed down or not moving */
  {
    /* here send the frame via XBEE */
    /* send the frame to anyone listening */
```

```
    error = xbee868LP.send( MESHLIUM_ADDRESS, frameSD, lengthSD );

    // check TX flag
    if( error == 0 )
    {
     transmission_index_data++; /* update sensor data index */
     USB.println(F("send ok"));
    }
    else
    {
     USB.println(F("send error"));
    }
   }
   else
   {
    /**** if we are moving, abort transmission ****/

    /* close SD card */
    SD.OFF();

    /* turn ZigBee off, to save power */
    xbee868LP.OFF();

    /* return to other code execution */
    return;
   }

  }

  /**** transmission ended, close modules ****/
  /* close SD card */
  SD.OFF();

  /* turn ZigBee off, to save power */
  xbee868LP.OFF();
}

/*
 * Function: Setup_SD_Card
 * ----------------------
 *    Deletes old files and creates
 *    new files for logging the data
 *    from the sensors and the GPS
 *
 *    Input:  none
 *
 *    Output: none
 */
```

```
void Setup_SD_Card(void)
{
 /* Set SD ON, to log samples */
 SD.ON();

 /* DELETE FILE WITH SAMPLES */
 /* Delete file if already exists, to discard old samples */
 sd_answer = SD.del(filename);

 if(sd_answer)
 {
  #ifdef PDEBUG
  USB.println(F("file containing samples deleted"));
  #endif
 }
 else
 {
  #ifdef PDEBUG
  USB.println(F("file containing samples NOT deleted"));
  #endif
 }

 /* create new file */
 sd_answer = SD.create(filename);

 if(sd_answer)
 {
  #ifdef PDEBUG
  USB.println(F("file containing samples created"));
  #endif
 }
 else
 {
  #ifdef PDEBUG
  USB.println(F("file containing samples NOT created"));
  #endif
 }
 /**********************************************************/

 /* DELETE FILE WITH GPS */
 /* Delete file if already exists, to discard old samples */
 sd_answer = SD.del(filename_gps);

 if(sd_answer)
 {
  #ifdef PDEBUG
  USB.println(F("file containing GPS coordinates deleted"));
  #endif
```

```
 }
 else
 {
  #ifdef PDEBUG
  USB.println(F("file containing GPS coordinates NOT deleted"));
  #endif
 }

 /* create new file */
 sd_answer = SD.create(filename_gps);

 if(sd_answer)
 {
  #ifdef PDEBUG
  USB.println(F("file containing GPS coordinates created"));
  #endif
 }
 else
 {
  #ifdef PDEBUG
  USB.println(F("file containing GPS coordinates NOT created"));
  #endif
 }

 USB.println(F("SD card was initialized"));
}

/*
 * Function: Setup_App_Sensors
 * --------------------------
 *   Calculates the logarithmic functions
 *   that are going to be used by non-linear
 *   sensors
 *
 *   Input:  none
 *
 *   Output: none
 */
void Setup_App_Sensors(void)
{
 /* Calculate the slope and the intersection of the logarithmic functions */
 COSensor.setCalibrationPoints(COres, COconcentrations, numPoints);         /* for CO  sensor */


 #ifdef MODE2 /* if we work on full demo, load other sensors */

 CO2Sensor.setCalibrationPoints(CO2voltages, CO2concentrations, numPoints);   /* for CO2 sensor
*/
```

```
 NO2Sensor.setCalibrationPoints(NO2res, NO2concentrations, numPoints);       /* for NO2 sensor */

 #endif

 USB.println(F("Sensors were initialized"));
}

/*
 * Function: Setup_GPS_And_RTC
 * ---------------------------
 *   Starts the GPS and stores the
 *   Ephemeris information to the SD card
 *   so it can start faster during the
 *   application. Sets the real time and
 *   date to the RTC
 *
 *   Input:  none
 *
 *   Output: none
 */
void Setup_GPS_And_RTC(void)
{
 /* define status variable for GPS connection */
 bool status;

 /* Init SD pins */
 SD.ON();

 /* Turn GPS on */
 GPS.ON();

 ///////////////////////////////////////
 // wait for GPS signal for specific time
 ///////////////////////////////////////
 status = GPS.waitForSignal(TIMEOUT);

 ////////////////////////////////////////////////////////////////////
 // if GPS is connected then store/load ephemeris to enable hot start
 ////////////////////////////////////////////////////////////////////
 if( status == true )
 {
   /* store ephemeris in "EPHEM.TXT" */
   GPS.saveEphems();

   #ifdef PDEBUG
   USB.println(F("Ephemeries were saved in a file"));
   #endif
```

```
    /* set time in RTC from GPS time (GMT time) */
    GPS.setTimeFromGPS();

    /* at this point, compensate the time, to match the Greek time */
    /* add 3 hours to the hour that was taken from the GPS */
    RTC.setTime(RTC.year, RTC.month, RTC.date, RTC.dow(RTC.year, RTC.month, RTC.day), RTC.hour
+ 3, RTC.minute, RTC.second);

    /* switch SD card off */
    SD.OFF();

    /* switch GPS off */
    GPS.OFF();

    USB.println(F("GPS was initialized"));

    USB.println(F("RTC was initialized"));

    Check_The_Date(); /* update the date, we do not want to have the initialized values - maybe the
app will start during the night */
  }

}

 /*
 * Function: Check_The_Date
 * ------------------------
 *    Checks if a day has passed since
 *    we started the application. Checks
 *    if it is day or night outside
 *
 *    Input:  none
 *
 *    Output: TRUE    if a day has passed
 *            FALSE   if no day has passed
 */
bool Check_The_Date(void)
{
  bool day_flag_updated = false;
  int temp_hour = 0;

  /* see the day (eg. is it Monday, Tuesday etc.) */
  char* Date = strtok(RTC.getTime(),",");

  if(strcmp(Date, temp_day) == 0) /* if the day hasn't changed, don't update the flag */
  {
    day_flag_updated = false;
```

```
 }
 else /* if a day has passed, update the flag */
 {
  temp_day = Date; /* update the temp day variable */
  day_flag_updated = true; /* update the flag, so the data can be sent */
 }

 temp_hour = (int)RTC.hour; /* check what time it is */

 /* update the daytime */
 if(temp_hour <= morning_limit || temp_hour >= night_limit)
 {
  daytime = "NIGHT"; /* between 19:00 and 6:00, it is night */
 }
 else
 {
  daytime = "DAY"; /* else, it is day */
 }

 /* if one day has passed, return TRUE */
 if(day_flag_updated)
 {
  return true;
 }
 else /* else return FALSE */
 {
  return false;
 }
}

/*
 * Function: Run_Application
 * ------------------------
 *   Starts the sensor board. Starts the sensors
 *   and wait 90 seconds for them to warm up.
 *   Reads samples from sensors, stores them
 *   to frames and send them via XBee if
 *   necessary.
 *
 *   Input:  none
 *
 *   Output: none
 */
void Run_Application(void)
{
  /* enable sensors board */
  Gases.ON();
```

```
/* enable the CO sensor */
COSensor.ON();


#ifdef MODE2 /* if we work on full demo, start other sensors */

CO2Sensor.ON();
NO2Sensor.ON();

#endif

/* wait 90 seconds for the sensors to warm-up */
Ninety_Seconds_Delay();


/**** READ SENSORS ****/
Read_Sensors();

#ifdef PDEBUG /* only print to serial if the debug mode is set */
/**** PRINT OF THE RESULTS ****/
Print_Sensors_Values();
#endif

/**** CREATE ASCII FRAME ****/
Create_Ascii();

/**** CHECK IF WE ARE STOPPED AND THERE IS A ZIGBEE STATION NEARBY ****/
Send_Data();

/* if a day has passed, send the logged data to Meshlium */
if(Check_The_Date() == true)
{
 /* power up SD card */
 SD.ON();

 /* power up the GPS */
 GPS.ON();

 /* load ephemeris */
 /* GPS.loadEphems(); */

 /* send the samples to Meshlium */
 Read_Logged_Data_And_Send(transmission_index);

 /* send the coordinates of the sent sample to Meshlium */
 Read_GPS_Coordinates_And_Send(transmission_index);

 /* close the GPS module */
```

```
    GPS.OFF();

    /* close SD card */
    SD.OFF();
   }
}

/*
 * Function: Send_Data
 * -------------------
 *   If the bus is not moving,
 *   send frames via XBee
 *
 *   Input:  none
 *
 *   Output: none
 */
void Send_Data(void)
{
 /* speed variable will check if the vehicle is moving or not */
 int vehicle_speed = 0;

 /* power up SD card */
 SD.ON();

 /* power up GPS */
 GPS.ON();

 // load ephemeris previously stored in SD
 /* GPS.loadEphems(); */

 status = GPS.waitForSignal(TIMEOUT);

 if( status == true ) /* we have GPS signal */
 {
  /* check if the vehicle is not moving */
  vehicle_speed = (int)GPS.speed;

  if (vehicle_speed < 5) // vehicle has slowed down or not moving
  {
    /* now it is a good time to check if there is a Zigbee Meshlium nearby and send it samples */
    Read_Logged_Data_And_Send(transmission_index);

    /* send the coordinates of the sent sample to Meshlium */
    Read_GPS_Coordinates_And_Send(transmission_index);
  }
  else
  {
```

```
      /**** we are moving , abort transmission ****/

      /* close the GPS module */
      GPS.OFF();

      /* close SD card */
      SD.OFF();

      /* return to other code execution */
      return;
    }
  }

  /* close the GPS module */
  GPS.OFF();

  /* close SD card */
  SD.OFF();
}

 /*
 * Function: Ninety_Seconds_Delay
 * -----------------------------
 *   waits 90 seconds for the gases
 *   sensors to warm up
 *
 *   Input:  none
 *
 *   Output: none
 */
void Ninety_Seconds_Delay(void)
{
  uint8_t i = 0; /* 90 seconds counter */

  for(i = 0; i < 90; i++)
  {
    delay(1000);
  }
}

/*
 * Function: See_Battery_Status
 * ---------------------------
 *   If we are in debug mode, print
 *   to the serial monitor the status
 *   of the battery
 *
 *   Input:  none
```

```
*
*   Output: none
*/
void See_Battery_Status(void)
{
  bool chargeState;
  uint16_t chargeCurrent;

  /* get charging state and current */
  chargeState = PWR.getChargingState();
  chargeCurrent = PWR.getBatteryCurrent();

  if (chargeState == true)
  {
   USB.println(F("Battery is charging"));
  }
  else
  {
   USB.println(F("Battery is not charging"));
  }

  /* Show the battery charging current (only from solar panel) */
  USB.print(F("Battery charging current (only from solar panel): "));
  USB.print(chargeCurrent, DEC);
  USB.println(F(" mA"));

  USB.println();
}
```