iscte

INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Network of Sensors for Measuring Environmental Pollution Using Open-Source / Open-Hardware Methodologies

Mariana de Almeida Barros

Master in Telecommunications and Computer Engineering

Supervisors:
Prof. Dr. Alexandre Passos Almeida, Auxiliar Professor,
ISCTE – Instituto Universitário de Lisboa

Prof. Dr. Pedro Santana, Auxiliar Professor,
ISCTE – Instituto Universitário de Lisboa

Outubro, 2020

iscte

TECNOLOG|AS
E ARQU|TETURA

Departamento of Information Science and Technology

Network of Sensors for Measuring Environmental Pollution Using Open-Source / Open-Hardware Methodologies

Mariana de Almeida Barros

Master in Telecommunications and Computer Engineering

Supervisors:
Prof. Dr. Alexandre Passos Almeida, Auxiliar Professor,
ISCTE – Instituto Universitário de Lisboa

Prof. Dr. Pedro Santana, Auxiliar Professor,
ISCTE – Instituto Universitário de Lisboa

Outubro, 2020

Resumo

Este trabalho envolve tanto hardware como software. A parte de hardware consiste numa rede de sensores low-cost preparada para medir a poluição na cidade de Lisboa. Os dados sobre a poluição ambiental são recolhidos por um sensor de gás, um sensor de temperatura, humidade e pressão, um sensor de partículas em suspensão e um módulo GPS. O sistema irá ser montado em cima de autocarros, no entanto é um sistema escalável o suficiente para ser montado noutros veículos. Este tem várias formas de alimentação energética, sendo uma delas panéis fotovoltaicos o que permite uma maior mobilidade. Os dados recolhidos pelos sensores são enviados por MQTT para um servidor Ubuntu e colocados numa base de dados PostgreSQL.

Foi também feita uma aplicação web em HTML, CSS e Javascript, que permite ao utilizador consultar gráficos com os valores recolhidos pelos sensores, consultar um HeatMap com zonas encarnadas, amarelas e verdes conforme os níveis de poluição nos locais e mais importante é lhe permitido criar uma rota que evita zonas poluídas com um ponto de início e fim nos sítios desejados.

Por último foram feitos alguns testes, uns com os painéis fotovoltaicos e com o sistema parado no mesmo sítio, outros testes foram feitos com recurso a um power bank potente também com o sistema parado. Os testes em movimento foram feitos com duas formas de fornecimento de energia, uma foi novamente o power bank e outra foi a bateria da bicicleta elétrica usada para mover o sistema pela cidade de Lisboa.

Palavras-chave: Rede de sensores, Poluição, Sensores, Gás, Partículas, Temperatura, Humidade, GPS, HeatMap, Painéis Fotovoltaícos, Bicicleta, Autocarro.

Abstract

This work involves both hardware and software. The hardware part consists of a low-cost sensor network prepared to measure pollution in the city of Lisbon. Data on environmental pollution is collected by a gas sensor, a temperature, humidity and pressure sensor, a dust sensor and a GPS module. The system will be mounted on buses, however, it is scalable enough to be placed on other vehicles. It has various forms of power supply, one of them is photovoltaic panels, which provide clean energy. The data collected by the sensors are sent by MQTT to an Ubuntu server and placed in a PostgreSQL database.

A web application was designed for the system using HTML, CSS and Javascript, which allows the user to view graphs with the values collected by the sensors, consult a HeatMap with red, yellow, and green zones according to the levels of pollution in the locations. Even more important, the application allows creating a route avoiding exposure to polluted areas with a starting point and endpoint in the desired places by the user.

Lastly, some tests were made with the photovoltaic panels with the system stopped. Other tests were done using a powerful power bank, also with the stopped system. The motion tests were made with two forms of power supply, one was again the power bank and the other was the battery of the electric bike used to move the system around the city of Lisbon.

Keywords: Network Sensor, Pollution, Gas, Dust, Temperature, Humidity, GPS, HeatMap, Photovoltaic Panels, Bike, Bus.

Acknowledgements

Firstly, I would like to thank my biggest lifetime support, my family. My mother Maria Barros, my father Pedro Barros, my sisters Madalena and Margarida Barros, my brother Martim Barros and my grandmother Cecília Almeida for providing this dissertation and all my education.

A big thanks to my boyfriend João Monge, for all the love, help, support and dedication that has been given me throughout the time. Also, I would like to thanks his parents, Cristina and José Monge.

A special thanks to all my friends, colleagues and laboratory teammates for the companionship aid provided.

I would like to thanks my advisors, Professor Alexandre Almeida and Professor Pedro Santana for the orientation, support, motivation and encouragement given during the development of the dissertation.

Finally, to the Telecommunications Institute at ISCTE-IUL, thanks for providing all the material and resources needed for this dissertation.

Contents

Resumo iii

Abstract v

Acknowledgements vii

List of Figures xiii

List of Tables xv

Abbreviations xvi

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Context 2
- 1.3 Research Questions 3
- 1.4 Goals 3
- 1.5 Research Method 4

2 State of the Art **5**
- 2.1 Internet of Things (IoT) 5
- 2.2 Sensors 6
- 2.3 Computational Unit 7
- 2.4 Communication 9
- 2.5 Power Supply 10
- 2.6 Messaging Protocol 11
  - 2.6.1 Why MQTT 11
  - 2.6.2 How MQTT works 12
- 2.7 Related Work 13

3 System Hardware **17**
- 3.1 Energy Requirements 18
  - 3.1.1 Photovoltaic Panel 20
  - 3.1.2 Voltage Converter 20
  - 3.1.3 Power Bank 21
  - 3.1.4 Power Cable 22

3.2 Sensors 23
3.2.1 Temperature, Humidity and Pressure 23
3.2.2 Dust 24
3.2.3 Gas 25
3.3 Analog-to-Digital Converter 26
3.4 Global Positioning System 27
3.5 Data Processing 27
3.6 Wireless Communication 28
3.7 Prototype 29
3.7.1 First Prototype 29
3.7.2 Prototype evolution 30
3.7.3 Box Prototype 34

4 System Software **37**
4.1 Hosting 37
4.2 Database 38
4.3 Testing Server 40
4.4 Node Software 41
4.4.1 Comparison of programming languages 41
4.4.2 Python Scripts 42
4.5 User Interface 44
4.5.1 Deployment of the Web App 44
4.5.2 Web App Activity 49
4.5.3 Implementation of the Web Platform 51

5 Tests and Results **53**
5.1 System preliminary prototype 53
5.1.1 Results 57
5.1.2 Photovoltaic Panels Test 57
5.1.2.1 GPS Test 58
5.1.2.2 Gas Test 59
5.1.2.3 Temperature Test 62
5.1.2.4 Dust Test 63

6 Conclusions and Future Work **65**
6.1 Conclusions 65
6.2 Future Work 66

Appendices **71**

A Python Scripts **71**

Bibliography **75**

List of Figures

1.1 IoT Branches 2
1.2 Model of the research method 4

2.1 Sensor BME280 8
2.2 Sensor B5W-LD0101 8
2.3 Sensor MQ-135 8
2.4 Arduino Uno 9
2.5 Raspberry Pi 3 9
2.6 Bus with photovoltaic panel on top 11
2.7 MQTT Protocol 13

3.1 System Architecture 17
3.2 Consumption Test 18
3.3 Running no peripherals Interface 19
3.4 Running RPi and peripherals Interface 19
3.5 Photovoltaic Panel 20
3.6 Voltage Regulator Schematics 21
3.7 Power Switch Circuit 22
3.8 First Prototype 30
3.9 Final System schematics 32
3.10 Final system Breadboard 32
3.11 Final System Print Circuit 33
3.12 System Upper View 33
3.13 System Front View 33
3.14 System Bottom View 34
3.15 Box Lateral View 34
3.16 Box Side View 34
3.17 Aerodynamic Test Front 35
3.18 Aerodynamic Test Front Side 35
3.19 Aerodynamic Test Lateral 35

4.1 Hosting Schematics 38
4.2 Database Diagram 40
4.3 MQTT Real Scenario 43
4.4 Homepage on a phone 45
4.5 Homepage 45

4.6 Route Selection 46
4.7 Gas Generate Graph 46
4.8 HeatMap 47
4.9 Route from ISCTE to IST avoiding pollution 48
4.10 Route from ISCTE to IST avoiding pollution 49
4.11 Fastest Route from ISCTE to IST 49
4.12 Activity Diagram 50
4.13 Data Communication between Client and Database 51

5.1 Stopped Test 54
5.2 Stopped Test with photovoltaic panels 54
5.3 System Attached to the Bike 55
5.4 System Attached to the Bike Side View 56
5.5 System Powered by Bike Battery 56
5.6 Solar Exposure During a Daytime 57
5.7 Strava Ride 59
5.8 GPS Test 59
5.9 Gas Test 60
5.10 Gas Test Chamber 60
5.11 Gas Test 62
5.12 Gas Test 62
5.13 Dust Test 64

A.1 Script for Exporting Data to csv File part 1 71
A.2 Script for Exporting Data to csv File part 2 72
A.3 Script for Exporting Data to csv File part 3 72
A.4 Client Publisher 73
A.5 Client Subscriber 74
A.6 Message Exchanged with QoS=2 74

List of Tables

2.1 Sensors Comparison Part 1 7

2.2 Sensors Comparison Part 2 7

3.1 Peripherals Consumption 18

3.2 System Consumption 18

3.3 Comparison DHT22 BME280 24

3.4 ADS1015 features 26

3.5 GPS features 27

3.6 Power Consumption of RPi 28

3.7 Parameters of Wi-Fi and GPRS 29

4.1 Comparison between Java and Python 41

Abbreviations

IoT	**I**nternet **o**f **T**hings (see page **??**)
GPRS	**G**eneral **P**acket **R**adio **S**ervice (see page 27)
CO2	**C**arbon Dioxide (see page 6)
UFPs	**U**ltra **F**ine **P**articles (see page 6)
WSNs	**W**ireless **S**ensor **N**etworks (see page 6)
OS	**O**perating **S**ystem (see page 8)
MQTT	**M**essaging **Q**ueuing **T**elemetry **T**ransport (see page 11)
HTTP	**H**ypertext **T**ransfer **P**rotocol (see page 12)
QoS	**Q**uality **o**f **S**ervice (see page 12)
GUI	**G**raphic **U**ser **I**nterface (see page 19)
hPa	**h**ecto **P**ascal (see page 23)
SCK	**S**PI **C**lock (see page 23)
SDI	**S**erial **D**ata **I**n (see page 23)
SCL	**S**erial **C**lock **L**ine (see page 23)
SDA	**S**erial **D**ata Line (see page 23)
GPS	**G**lobal **P**ositioning **S**ystem (see page 27)
csv	**C**omma **S**eparated **V**alues (see page 28)
PCB	**P**rinted **C**ircuit **B**oard (see page 31)
VPS	**V**irtual **P**rivate **S**erver (see page 37)
MVCC	**M**ulti **V**ersion **C**ucorrency **C**ontrol (see page 38)
HTML	**H**yper **T**ext **M**arkup **L**anguage (see page 51)
API	**A**pplication **P**rogramming **I**nterface (see page 51)
PHP	**H**ypertext **P**re**P**rocessor (see page 51)
VOC	**V**olatile **O**rganic **C**ompounds (see page 60)

ppm	**p**artes **p**er **m**illion (see page 60)
CAD	**C**omputer **A**ided **D**esign (see page 34)

Chapter 1

Introduction

1.1 Motivation

Internal migration, from rural to urban areas, impacts the rate of growth of cities. Pollution and air pollution, in particular, is a resulting product of that fast development. Air pollution is an essential factor in health care. Air pollution is responsible for 6% of death causes originating more than 40 000 cases per year in Europe alone. Half of the mortality caused by air pollution is due to motorized traffic, creating 25 000 new cases of chronic bronchitis in adults, more than 290 000 incidents of bronchitis in children and more than half a million asthma attacks [1].

This works research involves a mix of hardware/software where air pollution data is collected from small, low-cost environmental sensors. The data is then analyzed so that intelligent actions can be performed in real-time. Regarding people mobility, we propose to the user a route from an origin point to a destination point that minimizes the exposure to polluted areas. This aligns with the European 2050 carbon-neutral goal, a vision for a prosperous, modern, competitive, and climate-neutral economy by 2050.

1.2 Context

This project is related to monitoring environmental pollution using wireless communications to connect the sensors to the Internet. It can also be framed in the Internet of things (IoT) applications using low-cost sensors for analysis of environmental pollution, connected to the Internet either by Wi-Fi or by a mobile network. The IoT has attracted more and more interest, being considered by many the next big technology revolution [2]. At this moment it is easy to find IoT in everyday life such as in vehicles, roads in public transportation systems, wireless pill-shaped cameras in the system of digestive trails for health applications, air conditioner or other household things can be attached with sensors, used to track data in all these things [3]. Figure 1.1 illustrates the various branches of IoT. Recent approaches in sensing technology, more specifically in Wireless Sensor Networks (WSNs), now help monitor real-time environmental [4]. Several similar projects to this work have emerged due to the low-cost factor, contrary to the standard approaches based on fixed environmental stations as mobile allows data acquisition in a lot of places using just a set of sensors. Low-cost sensors offer significant advantages of compact size, low energy requirements, and recently getting greater attention for usage as portable monitors measuring particulate matter mass concentrations [5]. Among all air gas pollutants(SOx, NOx, CO, NH3, O3, etc.), there has recently been increasing attention to the study of particulate matter due to the significant negative impact on human health [6].

FIGURE 1.1: IoT Branches

1.3 Research Questions

In this section we present four research questions. These questions will be answered in the development of the dissertation.

- Will the system be able to create a route avoiding polluted areas?
- Will the system be able to monitor pollution accurately?
- What type of power supply is most suitable for the system?
- Is it possible to test the sensors without resorting to expensive laboratory equipment?

1.4 Goals

This project has the purpose of being low cost, involving some hardware components and system software in a single and complete work.

The main goal is to develop a scalable system that measures environmental pollution, which can be installed in motorized vehicles of public transport or on a simple bike. Using periodic sampling, this system should acquire a set of environmental variables, including temperature, humidity, pressure, oxide gases, and the number of particles in suspension, performing adequate signal processing. The prototype consists of a microcomputer. After the system completes data acquisition, this data will be transmitted via wireless communication that will work with a protocol for messages transmissions to a server and a database.

Another goal is to develop a Web App that, after the acquisition of the pollution variables, should be able to give its users feedback on environmental pollution in the city of Lisbon through a heatmap that informs the user about the most and least polluted areas. The Web App should also make it possible to propose a route, with a point of origin and destination, through areas where there is almost no pollution, something innovative in Lisbon.

1.5 Research Method

- **1st Step** Consists of a thorough investigation and consequently, a clear definition of the objectives and elaboration of a development plan. At the end of this phase, all the planning of the systems must be completed, and the choice of materials needed to carry out the project.
- **2nd Step** This step consists of the development of the project. Here, you take advantage of the previous phases material and detail all the system components and their interaction.
- **3rd Step** This step consists of implementing the entire project using the materials elaborated in the previous actions.
- **4th Step** At this stage of the project, the whole system will be tested. For this, we will have the help of base stations that will help us to understand the values that our sensors should present and help in their calibration. After this, all faults must be corrected before the next step.
- **5th Step** Finally, the final step consists of evaluating the entire project.

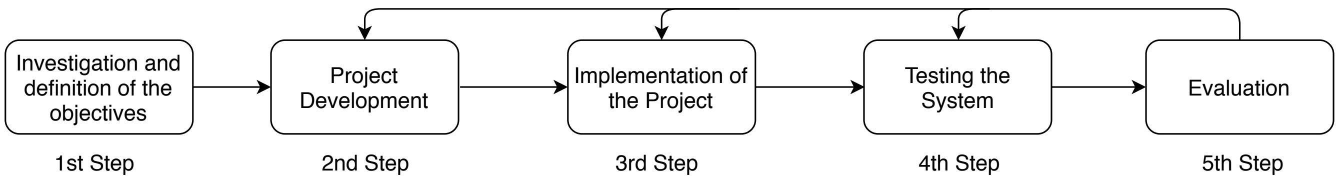


FIGURE 1.2: Model of the research method

Chapter 2

State of the Art

This work aims to develop a compact unit with sensors that could be mounted on public transportation buses, to monitor air quality in urban environments. With the objective of measure the pollution geographically dispersed, it is necessary a mode of transport, which could be the buses. This project does not cause any more pollution, since buses will make their regular journey; in other words, they will not have a different route to see the pollution everywhere. If we chose another mode of transport for our system, this would not be polluting. Since it is not necessary, considering that there are many buses with several routes in the city of Lisbon, there are many buses. Hence the need to replicate the prototype to place it on top of several buses and the need for low-cost sensors.

2.1 Internet of Things (IoT)

IoT represents the new step of the internet, allowing us to connect anything to it from our houses, cars, and even our heart condition through large sensor networks that could improve our lives. The possibilities are enormous, but the problems that need to be solved are also an issue [2]. Security is one concern because our houses, our cars, are connected to the internet. Another critical problem is privacy; for example, IP Cameras can be accessed virtually anywhere in the world. While IoT problems have a significant impact on the future of humans, the introduction of revolutionary new technologies and tools will improve our lives [7], like this project.

2.2 Sensors

Wireless Sensor Networks (WSNs) can increase the spatial density of measurements currently achieved. In the last decade, there has been growing interest in the development and deployment of such networks using low-cost air quality sensors [6]. To monitor air quality accurately, it is necessary to use the sensors that are suitable for what we want to measure, in this case: Humidity and Temperature; Carbon Dioxide (CO2) and other gases; and Ultrafine Particles (UFPs) . The parameters for the sensors choice were the fact that the system should be autonomous and low-cost because of the need to replicate the prototype. Given a wide range of sensor possibilities, taking into account the parameters for choice, we chose these three sensors: BME280, presented in figure 2.1, for measuring Temperature, Humidity, and Pressure. This sensor supports a higher operating temperature, a critical parameter since the sensor will be exposed to the sun all day long.

B5W-LD0101, presented in figure 2.2, is for measuring UFPs. This sensor has a better particulate matter detectable range, and that is important to have a better analysis, have relatively lower power consumption, and compare with the PPD60PV sensor [5] presented in table 2.1 is much cheaper.

The MQ-135 sensor, presented in figure 2.3, was the chosen one for measuring CO2 and other gases. All gas sensors, such as particulate matter sensors, are much more expensive than the temperature and humidity sensors, comparing MQ-135 with MG-811. The first sensor has a better operating temperature, much lower cost, and suitable for this project. Even the quantity of gases detected is from a broader range of options, like CO2, benzene, alcohol, smoke, nitrogen oxides, and other gases. As we can see in tables 2.1 and 2.2, based on [8], [9], [10], [11] [12], the choices of our sensors were based in research work having in mind the fact that the sensors should be low-cost, portable and low power consumption.

	PPD60PV	**DHT22/AM2302**	**BME280**	**B5W-LD0101**
Manufactorer	Shinyei	MikroElektronika	Adafruit	Omron
Measurement	PM	Temperature and Humidity	Temperature and Humidity	PM
Dimension	88 x 60 x 20	14 x 18 x 5.5	18 x 19 x 2	17.6 x 52.3 x 39.3
Detectable Range	$\sim 0.5\mu$m	N/A	N/A	$\sim 0.5\mu$m
Op. Voltage	5V	5V	5V	5V
Current Consumption	N/A	300μA	4mA	90mA
Op. Temperature	-30° C ~ 60° C	10° C ~ 40° C	0° C ~ 65° C	0° C ~ 45° C
Humidity Accuracy	<95%	<60%	0-100%	N/A
Price	213.93 €	8.95 €	17.46 €	11.69 €

TABLE 2.1: Sensors Comparison Part 1

	MG-811	**MQ-135**	**IRC-AT**
Manufactorer	Hanwei Electronics	Hanwei Electronics	Alphasense
Measurement	CO2	Gas	CO2
Dimension	32 x 42	18 x 17 x 6	20 x 32
Detectable Range	N/A	1000 ppm	5000ppm
Op. Voltage	5V	3-5V	5V
Current Consumption	N/A	15mA	20mA - 60mA
Op. Temperature	28° C	-20° C ~ 70° C	-20° C ~ 50° C
Humidity Accuracy	65%	<65%	95%
Price	60.90 €	5.00 €	174.40 €

TABLE 2.2: Sensors Comparison Part 2

2.3 Computational Unit

When there is a need to do a portable system based on hardware, it is common to use a microcontroller. However, there are several variants of microcontrollers; they all share some common ground. Nevertheless, sometimes it is necessary more computational power that microcontrollers do not have [13], so microcomputers are used instead.

An option for our system was the microcontroller Arduino presented in figure 2.4, which is a board based on the Microchip ATmega328P. However, we opted for the use of a microcomputer because of the better computational resources. The microcomputer we are going to use is the Raspberry Pi 3 Model B+ presented in figure 2.5, since this is perfect for IoT projects. The Raspberry Pi has several advantages regarding Arduino, like Wifi, Bluetooth, and an ethernet port for connectivity without resorting to external shields, thus becoming more compact.

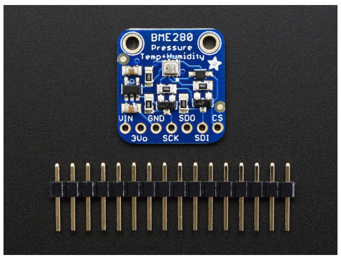


FIGURE 2.1: BME280 - Temperature, Humidity and Pressure Sensor

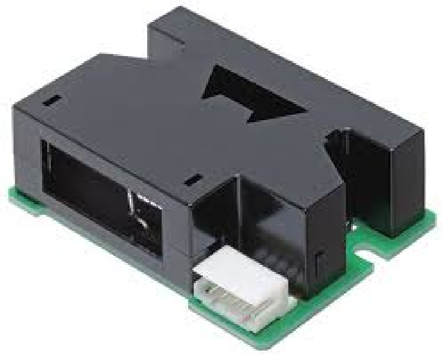


FIGURE 2.2: B5W-LD0101 - PM Sensor

FIGURE 2.3: MQ-135 - Gas Sensor

Raspberry pi supports multiple operating systems (OS) [14] while the Arduino microcontroller does not have OS. Raspberry pi has more processing power, better CPU, GPU, and more ram than the Arduino. Arduino can only be programmed in C while Raspberry pi has a more excellent range of compatible programming languages, which is easier for us.

Raspberry pi has a micro SD card, allowing us to increase the memory if necessary [15]. In contrast, the Arduino does not have this option without resorting to an external shield. Since our system will take many measurements during a long day, we need a lot of memory capacity to keep all the data temporary and only send it in the best moment.

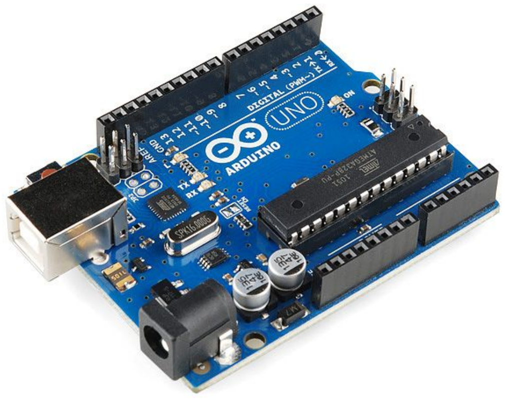


FIGURE 2.4: Arduino Uno

FIGURE 2.5: Raspberry Pi 3

2.4 Communication

This section is going to discuss what kind of communication we will use. Since city buses have a local Wi-Fi for their users, we will use that service to transmit the real-time data of our sensors placed on the top of the bus, which is compatible with the micro-computer we are going to use, and no external shield is required. It also has low power consumption features, works automatically with (Raspbian), and has a small size and low-cost [16]. Suppose we use another mode of transport without Wi-Fi. In that case, the solution adopted is General Packet Radio Service (GPRS) because it works well with Raspberry Pi with the insertion of an external shield [17].

2.5 Power Supply

This system is going to have more than one possibility for the power supply. One will make the system completely autonomous, so for that to happen, we have photovoltaic panels planted on the top of the buses, as illustrated in figure 2.6, the photovoltaic panel is a crucial element as a source of energy.

A photovoltaic system is based on specific materials' ability to convert the energy radiated by the sun into electrical power. The amount of solar energy that illuminates a given area is called irradiation and is measured in watts per square meter (W/m^2) [18]. The photovoltaic panels used in the system are six panels, and each one generates 9V and 150mA [**?**]. Since they are mounted in parallel, the 9V is kept constant, and the max current is 900mA to power the system. Since the Raspberry pi 3 model B+ works at 5V we need a voltage converter.

Using a photovoltaic brings a problem panel since if the bus enters a zone where there is no radiation from the sun, the system will power off due to energy failure. To avoid this from happening, a mighty power bank was added to the system as a power backup allowing the system to work in the event of energy failure. Another alternative of a power supply is 12V power cable supply, which is available at the bus power system. The system will have a power cable coming from the bus, and this power supply will allow the system to work all day and night as the panel will be charging the power bank through an electronic charging unit.

FIGURE 2.6: Bus with photovoltaic panel on top

2.6 Messaging Protocol

In this subchapter, the communication protocol will be introduced and explained. The protocol we will use is Message Queuing Telemetry Transport (MQTT), which has specific characteristics challenging to find in another protocol.

2.6.1 Why MQTT

MQTT is a lightweight protocol, so it is easy to implement in software and fast in data transmission; it is a publish-subscribe-based messaging protocol. The data packets are minimized. Consequently, the low network is used, and low power usage means that it saves the connected device's battery, which is a massive advantage to our system. It is a real-time protocol, and it makes it easy to establish communication between multiple devices. That is precisely what makes it perfect for IoT projects and our project specifically.

We choose to use the MQTT protocol and not use Hypertext Transfer Protocol (HTTP) because HTTP uses bigger data packets to communicate with the server, so it is slower than MQTT. HTTP request opens and closes the connection at each request, while MQTT stays online to make the channel always open between the broker "server" and clients. HTTP takes a longer time and more data packets; therefore, it uses more power [19]. Since our project uses battery cells charged by energy radiated by the sun, we need a protocol that consumes the least possible energy amount.

2.6.2 How MQTT works

MQTT protocol is based on clients and a server called 'Broker.' The server is responsible for handling the client request to receive or send data between each other to provide flexibility and simplicity. It uses a routing mechanism (one-to-one, one-to-many, many-to-many). Clients are connected devices in our case, the Web Application.

There are essential components in the MQTT protocol which are: the Broker, in the figure 2.7 is the central component, handles with the data transmission between clients deciding who is interested in what and publishing the message to all subscribed clients, just like is illustrated in figure 2.7; a topic which is the way a client registers interest for incoming messages or how a client specify where he wants to publish the message; the messages are another critical concept that is the information that is exchanged between devices; finally publish/subscribe system, in publishing and subscribe method a device can post a message on a topic or it can be subscribed to a particular topic to receive notifications as shown in figure 2.7 [20], [21].

The MQTT protocol defines three levels of Quality of Service (QoS) . The QoS levels represent how hard the broker/client will try to ensure that a message is received, messages may be sent at any QoS level, and clients may try to subscribe to topics at any QoS level [22]. In QoS(0): the message is delivered to the client just once, no need to acknowledge. In QoS(1): the message is delivered at least once, confirmation of receipt is required. In QoS(2): is uses a four-way handshake to ensure that the message is sent exactly once and has better communication between clients. For example, MQTT uses the keep-alive time to avoid data loss when a connection for any reason is interrupted. A durable session keeps messages when a subscriber is disconnected and sends the will to all subscribers when the publisher leaves.

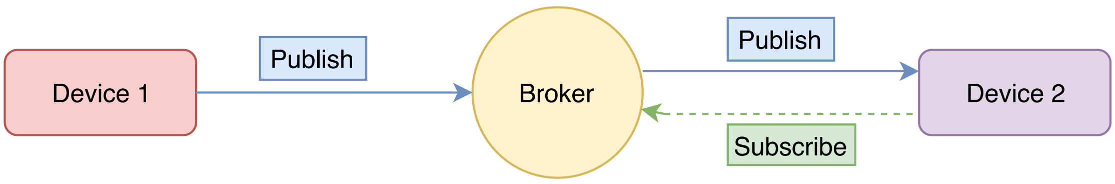


FIGURE 2.7: MQTT Protocol

2.7 Related Work

More than 7 million premature deaths are annually linked to air pollution, from which 2.6 million are mainly caused by urban outdoor air pollution [23]. In [24] one of the largest spatially resolved ultrafine particle (UFP) data set publicly available today that contains more than 50 million measurements was collected using mobile sensor nodes installed in public vehicles in Zurich, Switzerland. This collected data allowed the development of pollution maps; Global System transmitted Mobile Communications (GSM) data in real-time. This work has relevance because it contains pollution maps and similar sensors that are going to be present in this thesis too. The main difference between these projects and the one developed in the present dissertation is creating route features, which is a great advantage to our work. To improve the system, some sensors that measure other air pollution variables could be added

.

Another relevant work purposes a network of AQMesh platforms for air quality monitoring implemented in Oslo, Norway. AQMesh units are stationary, battery-powered platforms that measure four gaseous components: carbon monoxide (CO) nitrogen oxide (NO) nitrogen dioxide (NO2) ozone (O3) particle counting [25] The AQMesh platform also measures air temperature, relative humidity, and atmospheric pressure. The data is post-processed by the manufacturer to correct cross-interference and the effect of temperature and relative humidity [25].

HazeEst, a machine learning model, combines sparse fixed-station data with dense mobile sensor data to estimate the air pollution surface for any given hour on any given day in Sydney [26]. The obtained results can be visualized using a web-based application customized for metropolitan Sydney. Their system's continuous estimates can better inform air pollution exposure and its impact on human health. A limitation of this system is that it only measures underground air pollution and only monitor Carbon Monoxide (CO) as the pollutant.

Both papers have a similar context to our work. In [26], the prototype is placed at the top of public transport. In this case, the measurement of pollution is limited to the metropolitan area, including closed stations, while our system is always outdoors, which is an advantage. In the AQMesh, [25], are measured more gaseous components. However, the network is not mobile like ours; creating a mobile system is an excellent improvement to the project.

In [27] is presented a project carried out in Portugal with the name Urbisnet, which made a system like the others to measure air quality. The system aims to be placed on top of taxis; it seeks to be low-cost, portable, and transmit the data collected remotely through a wireless interface, just like our project. This system consists of five gas sensors (CO2, CO, O3, SO2, NO2), a temperature and humidity sensor, a GPS unit, an ADC, a microprocessor, and a Wi-Fi shield. This system is very similar to the project we are developing; however, this software aims to transmit data directly to a computer. It is possible to display the Urbisnet Station measurements and send new settings to the Urbisnet Station. In our work, we use the data collected by sensors to give useful information about exposure to pollution to the Web Application user that was developed. Since this system is low cost and portable, it could be placed in more places like garbage trucks.

At [28] the University of Cambridge, together with a well-known manufacturer of gas sensors, Alphasense Ltd., has produced an air quality monitoring device for portable, low-cost measurement while maintaining acceptable accuracy, such as ours. The sensors used were CO-AF, NO2-A1, and NO-A1 for measuring CO, NO2, and NO, respectively. This study is detailed and covers detailed aspects of sensor operation, calibration methods, and various field studies. The tests were done walking a defined path, and the data was recorded and overlaid on the Google Maps image. It will also be necessary to use Google Maps or other similar tools to test the GPS module's reliability in our project. More information was obtained by comparing different displacement methods. They compared pedestrians, cyclists, and vehicles. From the results of present in [28], we observe that cycling produced the lowest exposure to CO, and car trips the highest. In this work, only three aspects of air quality are measured and are all gas. In our daily routines, we should be aware of air quality aspects, so in our project, we added the sensor dust, thus not stating that we measure all air pollution levels.

LearnAir in [29] is a portable air quality monitor that measures environmental conditions such as temperature, humidity, and light. It connects to a smartphone application that can display results, send timestamp and georeferencing data to the cloud, and receive and display a measurement forecast based on the machine applied to data in the cloud. The main circuit boards for LearnAir connect to a multi-platform smartphone app written in Javascript. This library will compile Javascript in iOS, and Android applications even have plugins to access the GPS of the phone. Our software will also use Javascript. Although our application is Web and not a native application for smartphones, the coding language is the same. This project app shows the sensors' results; however, it has no feature like create a route, which gives an advantage to our work. An improvement to this project would be to make the app web too.

Finally in [30] is presented a concept called BaaS: Bus as a Sensor, an air quality monitoring system based on mobile sensor nodes placed on top of buses in Catania, Italy. Each node was equipped with Temperature, Humidity, Pressure, and Noise Level Sensors. Also, having a prototype to measure CO gas concentration and dust sensors could improve the system. The nodes sensor sends the data to an information process center in real-time. The data is processed but is not immediately used to transmit information on environmental pollution to citizens, as in our project.

Chapter 3

System Hardware

This chapter describes the system architecture and the description of all its components represented in figure 3.1. In section 3.1, we explain the energy requirements of the system and its consumption. In 3.2, we present all the used sensors. Then the analog-to-digital converter is described in section 3.3, the other peripheral, GPS is explained in section 3.4. Section 3.5 presents the processing done by a Raspberry Pi, then the wireless communication used protocol is mentioned in section 3.6. Finally, on 3.7 the prototypes are described.

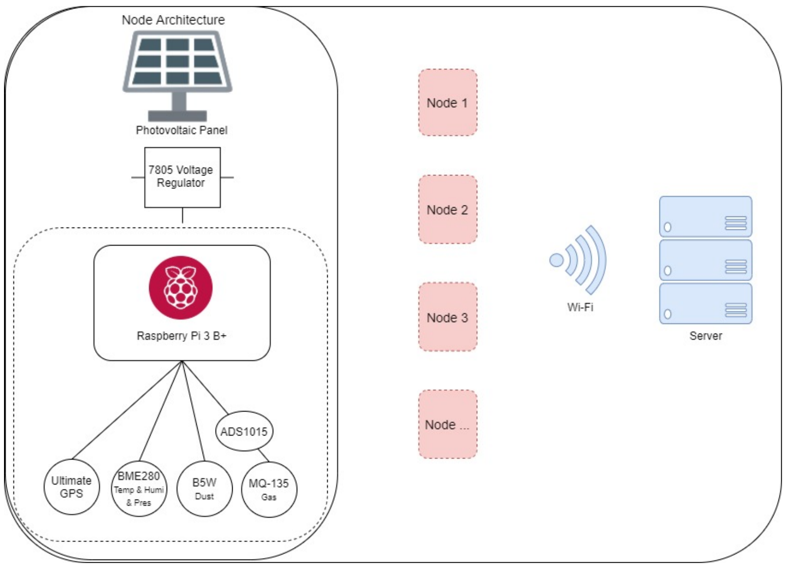


FIGURE 3.1: System Architecture

3.1 Energy Requirements

This section will cover the topic of power supply, its components, and current consumption. The data consumption presented in table 3.1 and in table 3.2 were taken from tests done with a multimeter in a laboratory, as illustrated in figure 3.2, this specific consumption is of the all system turning on.

	3.3V	5V
GPS	25 mA	-
Dust Sensor		77 mA
Gas Sensor	150 mA	-
ADC		
Temp, Humi and Pres	-	20 mA

TABLE 3.1: Peripherals Consumption

System State	Current Consumption
Sleeping	50 mA
Running	600 mA

TABLE 3.2: System Consumption

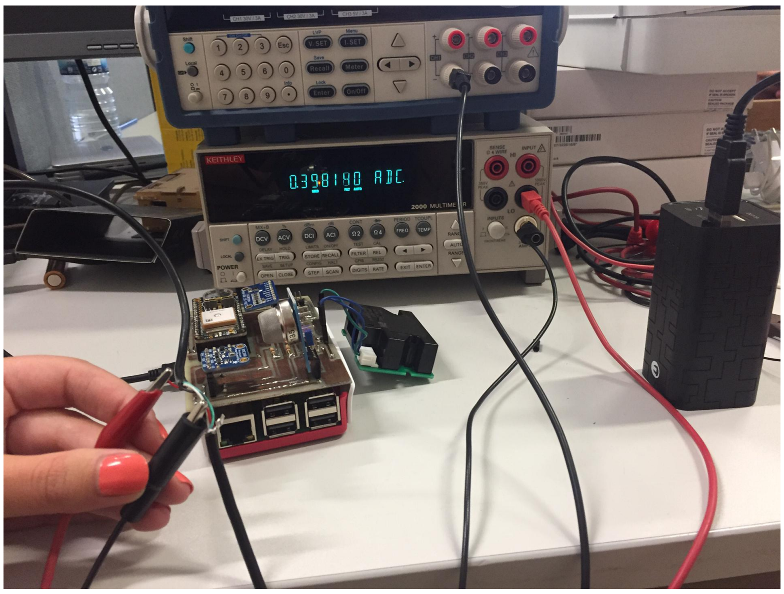


FIGURE 3.2: Consumption Test

Keithley 2000 multimeter was used to perform the current measurements connected to a computer using the protocol RS232 USB. On the computer, a script was created in LabView that provided a Graphic User Interface (GUI) , which plotted current data overtime. In figure 3.3, we can see the graphical characteristics of the system running with no peripherals and having an average of 400 mA of current and in figure 3.4 the interface with RPi and all peripherals running, having an average of 600 mA of current.

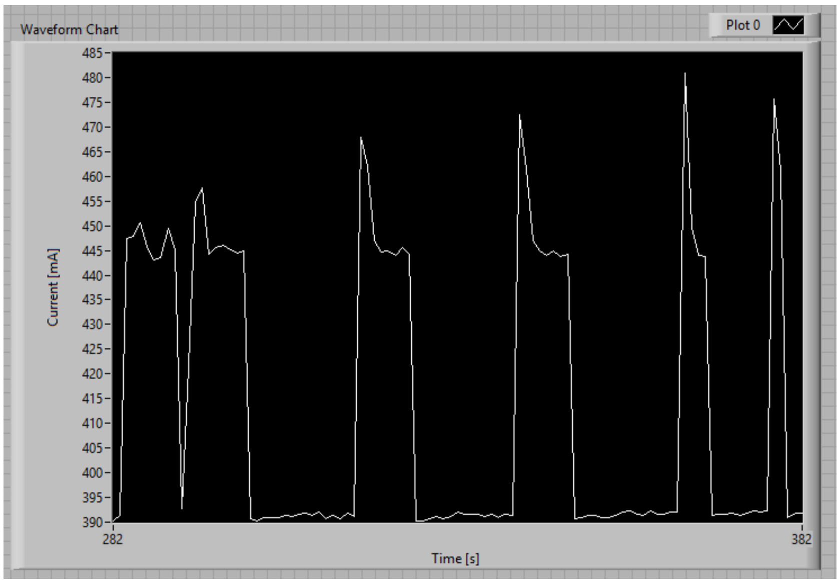


FIGURE 3.3: Running no peripherals Interface

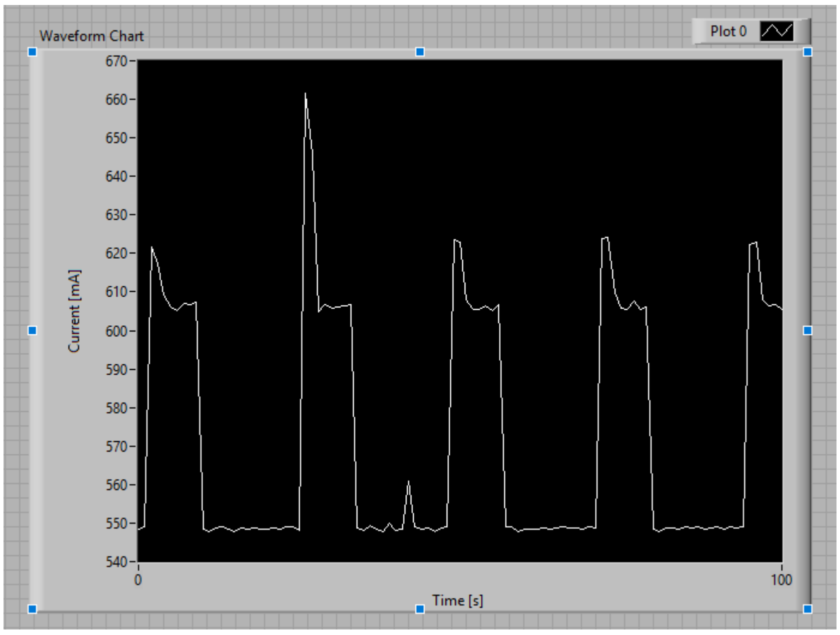


FIGURE 3.4: Running RPi and peripherals Interface

3.1.1 Photovoltaic Panel

The system can be powered by a photovoltaic panel. The panels used are from Conrad Electronics, presented in figure 3.5, which has 65 mm length and 125 mm width, and the nominal current is 150 mA each. The panels could be attached in parallel or series. The panels are attached in parallel for our project, so all six panels have 600 mA (150mA * 6 panels = 900 mA) and 9V. Raspberry Pi needs more or less 400 mA to turn on, and then running all peripherals consumes 600 mA, as shown in table 3.2.

FIGURE 3.5: Photovoltaic Panel

3.1.2 Voltage Converter

Usually, buses and trucks are 24V, but sometimes they use 12V. Hence, we decide to choose this voltage regulator because it is scalable to the two scenarios, provides a constant output voltage for a varied input voltage. For our project, the 7805 voltage regulator will provide a 5V output voltage.

The voltage conversion is done through the circuit present in figure 3.6 using a diode 1N4001 represented by D1, a capacitor with 0.1μF 25V represented by C1, and finally, a second capacitor represented by C2 with 10μF 25V.

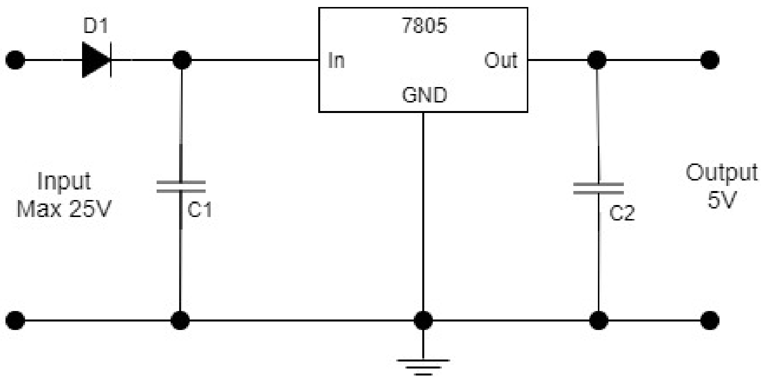


FIGURE 3.6: Voltage Regulator Schematics

The circuit 3.6 is a simple circuit in which one diode is sufficient to convert an alternating voltage into a continuous voltage and a filter capacitor to make the voltage stable. A DC voltage powers the circuit, the diode acts as a protection against reverse polarity and protects the regulator. This regulator allows currents of up to 1.5 A when connected to a heat dissipator. This circuit does not allow to regulate the voltage but allows to regulate the current, which is the perfect application for this project.

3.1.3 Power Bank

For some tests, a power bank was used as a portable power unit, with a capacity of 10400mAh/38.48Wh. This unit also has an output of 5V/2.1A, which is enough to power the entire system for several hours, collecting data from the sensors. This was a perfect solution because some tests were made using a bike, and the power bank allows the system to be powered and collect data for several hours and kilometers. One of the tests was done by pedaling for 14.22 kilometers with an average speed of 11 kilometers per hour without charging the power unit. According to 3.1, it gives about 1 hour and 17 minutes of continuous measurements. These measurements were being collected every 20 seconds. Other test took 9.84 kilometers of distance, with an average speed of 11.2 kilometers per hour, which gives 52 minutes of working system powered by a power bank. In 3.1, *Distance* is the distance traveled in kilometers, *Speed* in kilometers per hour is the average speed during the entire route, and *Power Time* refers to the number of hours that the power bank can keep the system running for a certain distance traveled at a certain speed.

$$PowerTime[h] = \frac{Distance[km]}{AverageSpeed[km/h]} \tag{3.1}$$

3.1.4 Power Cable

The system is going to have the possibility to be powered by a cable coming from the bus. The power of the bus is going to be 24V. The main advantage of this type of power is that it always has continued energy to power the entire system. This is very good for us since the bus can get into a tunnel, have no sunlight, or even stop under a tree that does not let the sun directly onto the photovoltaic panel. In either of these scenarios, the system will continue to function by switching from solar power to bus power.

To switch the power source from solar 9V to bus 24V automatically, we developed a circuit. To use this circuit, first, it is necessary to transform the 9V or 24V in 5V using the circuit illustrated in figure 3.7. The switching part is done by a JCZ-11F relay, that when it is on normally closed (NC) are the photovoltaic panels powering the system and when are on normally open (NO) are the power from the bus. A BC547 transistor was also used to open or close the relay based on the output of the photovoltaic panels, to keep the system always on, supercapacitors were added to the circuit, this way when it is transitioning from one power source to the other, the system keeps running.

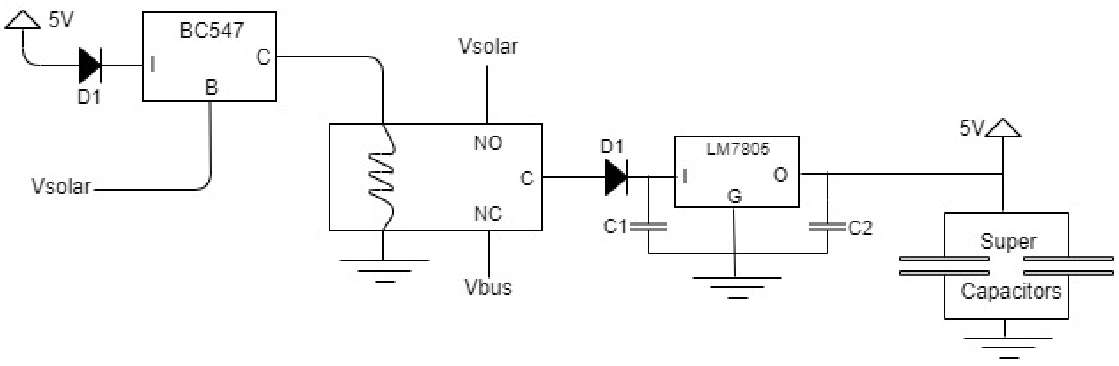


FIGURE 3.7: Power Switch Circuit

3.2 Sensors

This system combines different sensors to give the best results possible since it is a low-cost product and needs to be replicated. Attached to the Raspberry Pi 3 Model B+ is three different sensors.

3.2.1 Temperature, Humidity and Pressure

Initially, on the first sensor node made, the sensor used to measure temperature and humidity was DHT22, a sensor from MikroElektronika. Later, when building the final prototype, the sensor used was the Adafruit BME280. The sensor has been updated since although the DHT22 is a low-power sensor, which for this system is essential, it has a lower power consumption than the Adafruit sensor. BME280 measures temperature, humidity, and pressure, which the other sensor does not measure, has a wider range of operational temperature and humidity accuracy, as shown in table 3.3. This is relevant since it is a temperature and humidity sensor, and as shown in the table below, DHT22 only allows operating voltage of 5V while BME280 allows 3.3V or 5V. It also includes a voltage regulator on board that will take 3-5 VDC and safely converts it down.

This sensor is a perfect low-cost sensor solution for measuring humidity at about 3% accuracy, barometric pressure with an absolute accuracy of plus or minus one hectoPascal (hPa) and temperature of plus or minus 1° C, the sensor can even be used in both I2C and SPI. For simple wiring, the system goes with I2C. For that, the sensor has the I2C logic pins, SPI Clock (SCK) pin which connects to Raspberry Pi to Serial Clock Line (SCL) pin and Serial Data In (SDI) pin connects to Serial Data Line (SDA) pin also on Raspberry Pi. With the I2C logic pins and power pins connected, a library of Adafruit, available on GitHub [31], was used to start the sensor.

	DHT22	BME280
Measure	Temp, Humi	Temp, Humi, Pres
Consumption	300μA	4mA
Op. Voltage	5V	3.3V or 5V
Op. Temperature	10° C - 40° C	0° C - 65° C
Humidity	<60%	0%-100%
Manufactorer	MikroElektronika	Adafruit

TABLE 3.3: Comparison DHT22 BME280

3.2.2 Dust

B5W-LD0101 is a dust sensor that gives the number of particles >0.5μm in the air. This sensor does not have any library. It was necessary to understand the sensor operation through the datasheet, and the result is in the illustrative code presented in algorithm 1. The sensor delivers pulses according to the detected particles. Thus, when the pulse is at 5V, particles are detected. It is crucial to ensure that the pulse at 5V lasts at least 0.5ms. To count one particle, it is necessary to have at least four high pulses. The measurement only starts after the sensor is online for one minute. The count value is directly related to the measurement in μg/m^3. After a measurement period of 20 seconds, the count is reset before a new measurement period starts.

Algorithm 1 Dust Sensor Code

```
Result: Gives the amount of particles in the air
initialization
wait one minute
counter=0
start time = current time in seconds
dust = 0
while True do
    if sensor output is equal to low then
        if start time+20 is less or equal to current time then
            dust = counter/4
            start time = current time
        else
            print dust value
        end
    else
    end
end
```

3.2.3 Gas

For the gas measurement, we used MQ-135, a pollution gas sensor with an analog output. Since Raspberry Pi has only digital inputs, the ADS1015, analog-to-digital converter establishes the gas sensor analog signal into a digital signal like used in Raspberry Pi. When there is pollution gas, the conductivity of the sensor increases further, along with rising levels of gas concentration. The sensor uses simple electronic circuits to convert the conductivity charge to match the gas concentration output signal. The gas sensor booster converter is PT1301. The operating voltage of this gas sensor is between 2.5V and 5V.

The MQ-135 gas sensor has high sensitivity in ammonia (NH3), nitrogen oxides (NOx), benzene, alcohol, smoke, carbon dioxide (CO2), and other gas damage. The gas sensor has a lower conductivity for cleaning air as a gas sensing material. The detecting concentration scope are 10ppm-300ppm for NH3, 10ppm-1000ppm for benzene and 10ppm-300ppm for alcohol. Since the beginning, this was the chosen sensor to measure gas, senses gas presence efficiently, low cost, and suitable for the application.

3.3 Analog-to-Digital Converter

The ADS1015 is an excellent analog to digital converter that is easy to use with the Raspberry Pi, using I2C communication. It is a tremendous step up from the MCP3008 or other simple ADCs. The ADS1015 is a 12-bit ADC with four channels. It also has a programmable gain to amplify small signals and read them with higher precision, an internal voltage reference, and a clock oscillator. This converter has low current consumption, as shown in table 3.4 since it has two available conversion modes: single-shot mode and continuous conversion mode. In the first mode, the ADC performs one conversion of the input signal upon request and stores the value to an internal result register then the device enters in a low-power shutdown mode. This mode is intended to provide significant power savings in systems that only require periodic conversions or when there are long periods between conversions.

Since this is an Adafruit product and Adafruit has libraries of their products on GitHub, the systems use a valuable library to get good gas results [32]. The ADS1015 is connected to Raspberry Pi via I2C, as said above, so the SDA pin is connected to the SDA pin on RPi and SCL pin to SCL pin on RPi, and the ADS is connected to MQ-135 by the analog pin A0 [33].

	ADS1015
Manufactorer	Adafruit
Consumption	200μA
Op. Temperature	-40° C - 125° C
Power Supply Voltage	2V - 5.5V

TABLE 3.4: ADS1015 features

3.4 Global Positioning System

The Global Positioning System (GPS) chosen for the system was Ultimate GPS featherwing from Adafruit since it provides an accurate, sensitive, and location identification anywhere in the world. The GPS module can track up to 22 satellites on 66 channels, has an excellent high-sensitivity receiver as shown in table 3.5, and power consumption is low even in navigation, also presented in table 3.5 [34]. The GPS has a built-in LED that blinks while searching for satellites and blinks once every 15 seconds when a fix is found to conserve power. The GPS module is connected to RPi through TX, RX, the TX of GPS is connected to RX of the RPi and RX to the TX of RPi. This module also has an available library on GitHub that was used to simplify its use of [35].

	Adafruit GPS
Sensitivity	-165 dBm
Consumption in navigation	20 mA
Warm/Cold Start	34 seconds
Power Supply Voltage	3V - 5.5V

TABLE 3.5: GPS features

3.5 Data Processing

The processing is done by a Raspberry pi 3 Model B+. This computational unit is a low-cost computer with acceptable power consumption, as shown in table 3.6, also with sufficient processing capacity and memory that support I/O ports and the use of standard peripherals. Raspberry has an Operating System, which is Debian OS. Raspberry Pi allows a wide range of programming languages. However, the chosen one for this project was Python, a simple and intuitive language, whose vast library has allowed to adapt all the needed sensors. The only downside of Raspberry Pi is the lack of analog ports. Raspberry Pi has USB ports, which are very useful since the system can be powered on via USB by a power bank or cable, also have Wi-Fi in it, without the need of an external shield, which is the communication protocol that the system uses but in the case of not having Wi-Fi Raspberry Pi also has the possibility to insert a card General Packet Radio Service (GPRS) using an external shield.

All the information collected by the sensors and the GPS is recorded in the Raspberry Pi SD card, using a Comma Separated Values (csv) file, then the information is sent by MQTT protocol to the database. The choice of this type of file to save the data depends on the fact that if there is a problem with the connection, there is no loss of information, the data is stored and is not lost before there is the possibility of being sent. With the use of the csv file, we still benefit with the database because they are organized by commas and, in this way, make it easy to insert each field in the database.

	Raspberry Pi 3 Model B+
Voltage	5V
Running (all peripherals)	600 mA
Running (no peripherals)	400 mA
Turning on	400 mA
Sleep	50 mA

TABLE 3.6: Power Consumption of RPi

3.6 Wireless Communication

The wireless communication protocol chosen for this project was Wi-Fi. The first and main reason for this choice was that the bus has free Wi-Fi for its users, and the system can take advantage of that and use it. Also, Raspberry Pi has a built-in chip and antenna, so there is no need for an external shield. Data is transmitted to the database via wireless communication, and Wi-Fi is a very safe protocol with high complexity and has an extensive range of nodes, as can be seen in table 3.7. This is suitable for the system because it allows having several systems and is intended in this project.

If the system is used on the bikes, Raspberry Pi has a place for a card with the introduction of an external shield, on bikes the systems changes from Wi-Fi to General Packet Radio Service (GPRS) 3.5 because Wi-Fi consumes a lot and on the bus, there is always cable power, so the consumption is not a big problem. On the bike, the system is powered by photovoltaic panels and supercapacitors, so consumption of the protocol matters, as presented in table 3.7, GPRS consumes much less than Wi-Fi.

	Wi-Fi	GPRS
Standard	IEEE 802.11	GPRS
Frequency Band	2.4GHz	850, 900, 1800, 1900 MHz
Nodes	32	-
Power Consumption	100 - 350 mA	20 mA
Coverage	1 - 100 m	30 km
Complexity	High	High
Security	WPA/WPA2	Handset Autentication/Ciphering(AN CP, UP)

TABLE 3.7: Parameters of Wi-Fi and GPRS

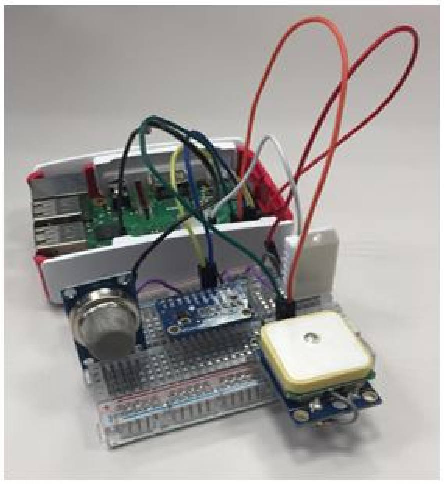
3.7 Prototype

In this section we present the two prototypes that were developed for this system and the box.

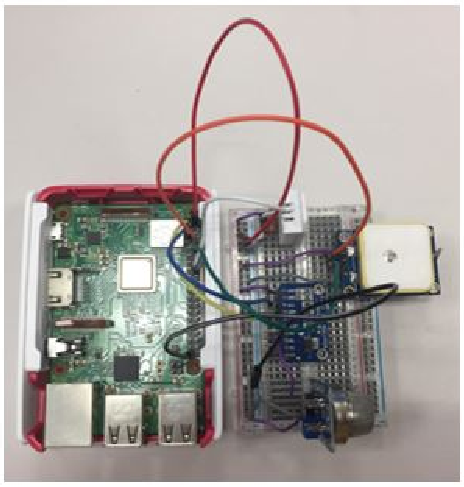
3.7.1 First Prototype

On the first prototype, the sensor used for measure temperature and humidity was the DHT22, the gas sensor used was the same (MQ-135), so the ADS1015 was needed as well, and there was no dust sensor. The GPS was also another one, the GY-GPS6MV2, with a ceramic antenna and built-in EEPROM for saving the configuration data when powered off. The disadvantage is that the module takes about 15 minutes with a clear view of the sky to get the first fix after power on. DHT22 sensor needs a resistance as shown in picture 3.8. Back in that time, we still counted with the help of a breadboard, also illustrated in figure 3.8. This first prototype did not have the power supply section, which was powered by a Raspberry Pi power adapter.

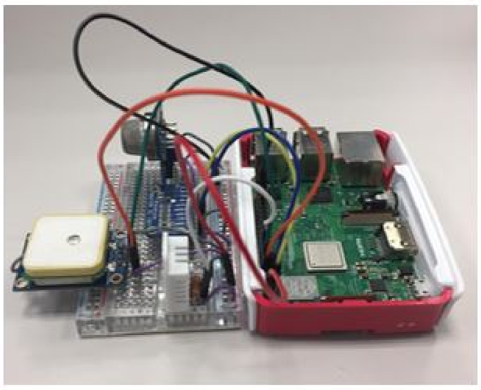
This prototype was still built with limited material and no power system because, at the time, we did not have the material yet. However, it was possible to build a rudimentary prototype. The gas sensor remained from one prototype to another and the ADC, so everything was taken advantage of from one prototype to the next. It was very advantageous to start building this prototype right away because it helped a lot to understand how the GPS worked. We saw that we needed one that would take less time to get a fix and to know how the sensors and the necessary connections worked.



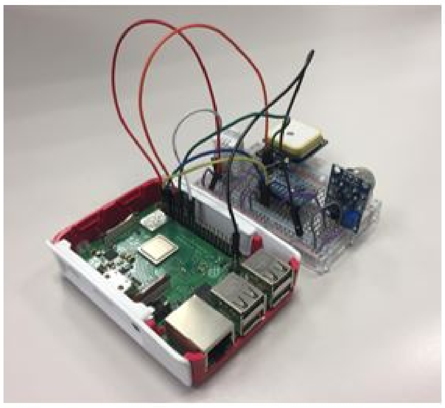
a) Side View



b) Upper View



c) Down View



d) Lateral View

FIGURE 3.8: First Prototype

3.7.2 Prototype evolution

The final prototype is the final system, used for tests to collect data. This system is composed of:

- **BME280, Temperature, humidity and pressure sensor**
- **MQ-135, Gas sensor**
- **B5W, Dust sensor**
- **GPS**

- **ADS1015, analog to digital converter**
- **Raspberry Pi 3 Model B+**

All those components are attached to a Printed Circuit Board (PCB) projected to power 5V. We developed that entirely for this project because it was a vast and messy system with the breadboard and the wires. The wires made the system complicated, and if when we had to test one component that had to turn off the others, it was difficult, and most of the time, it did not work. The board made everything simpler. It is no longer necessary to use wires. The system gets smaller because then the board is attached to the Raspberry Pi by headers that are soldered into the PCB, as shown in figure 3.14 the back view of the system. For the development of the PCB, the first thing to do was the system design in Fritzing, which is an open-source software to develop electronics. The result is present in figures 3.9, 3.10 and 3.11.

Figure 3.9 was where we connected all the components of the system using the schematics of the sensors, Raspberry Pi, and the GPS. After that, there is an option, which is the breadboard view represented in 3.10. In the end are presented the view of the print circuit, like in figure 3.11.

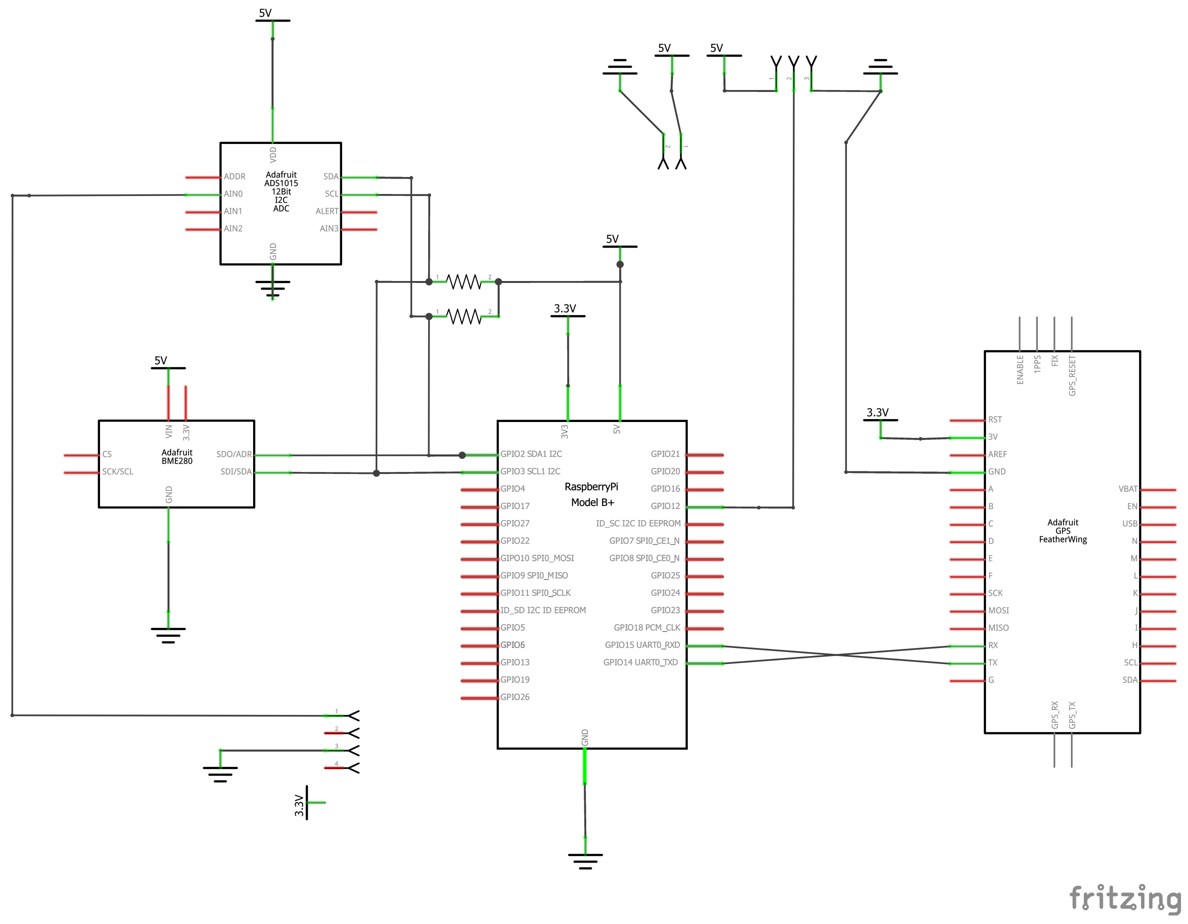


FIGURE 3.9: Final System Schematics

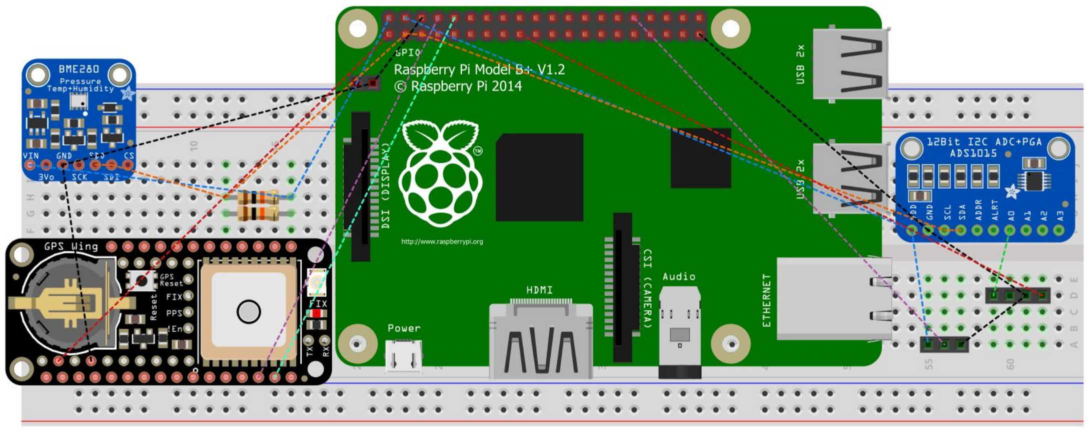


FIGURE 3.10: Final System Breadboard

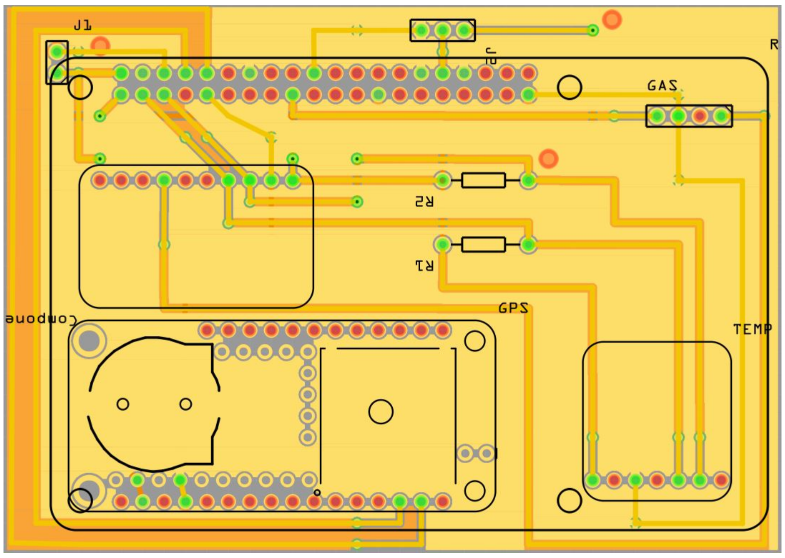


FIGURE 3.11: Final System Print Circuit

Then the PCB was made using a copper plate with two blank sides. A permanent marker was used to paint the pathways according to the circuit present in figure 3.11. Then a mixture of muriatic acid and acetone was made to etch the PCB. Afterward, it was necessary to test all the connections with a multimeter to check that everything was well connected and, even more critical, to check that there were no shorts. After welding the headers to connect all the components and place all the components in their position, the final system is what can be seen in figures 3.12, 3.13 and 3.14.

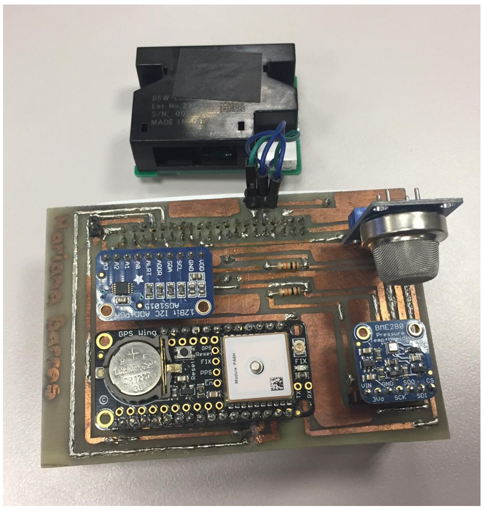


FIGURE 3.12: System Upper View

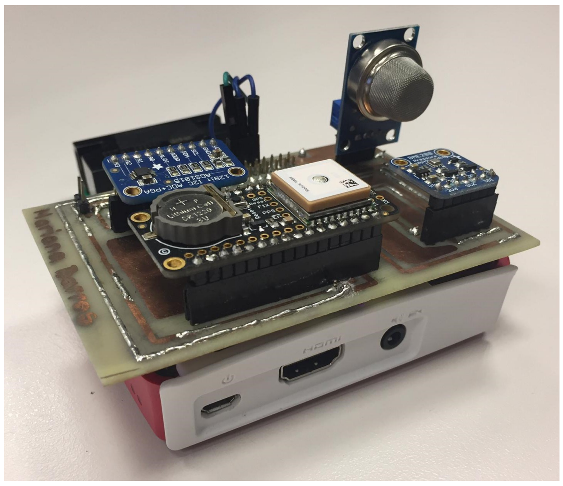


FIGURE 3.13: System Front View

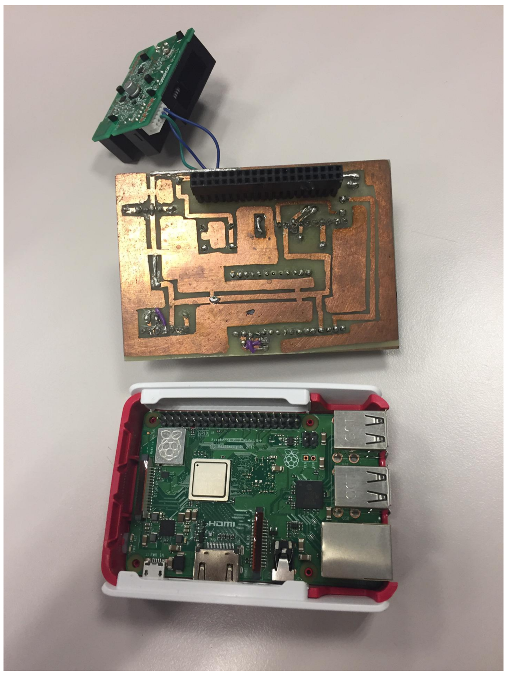


FIGURE 3.14: System Bottom View

3.7.3 Box Prototype

The system should be protected by a watertight box prototype that would allow the system to be placed in all weather conditions without being damaged, but it was never manufactured. However, the box was thought and designed using Computer-Aided Design (CAD) software, namely Siemens Solid Edge ST9, and for the aerodynamic analysis, FloEFD was used. The resulting output can be seen in figures 3.15 and 3.16.

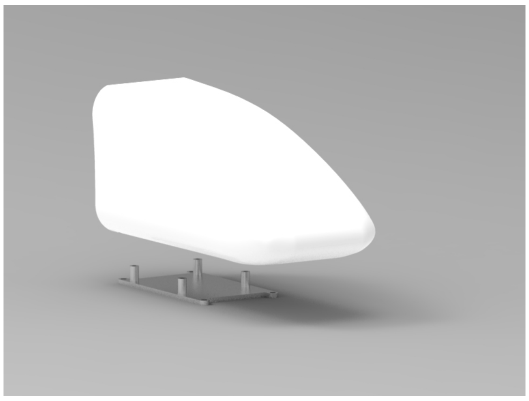


FIGURE 3.15: Box Lateral View

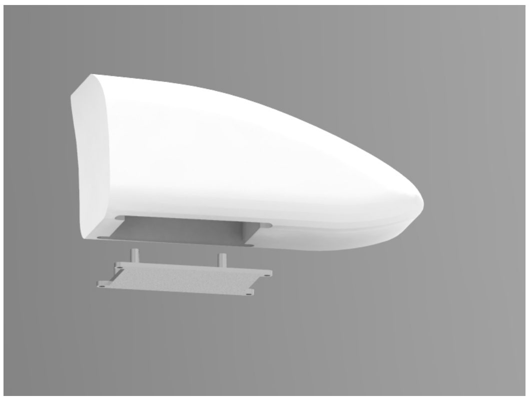


FIGURE 3.16: Box Side View

The design of the box was thought this way to avoid noise. The friction that the wind makes on the box causes noise, which is disturbing and makes much force, which can be dangerous. With this rounded shape, we reduce the box's noise and risk becoming detached from the roof of the bus. It was also considered the size of the system, length and width, and the height of all peripherals attached to the board in the design of the box as it should accommodate all the system components. The box was designed aerodynamically to reduce the noise generated on top of the bus and the applying forces. A study was performed to optimize the aerodynamics and presented in figures 3.17, 3.18 and 3.19. This study replicates the wind force generated by the bus movement. The sensor needs fresh air to analyze the pollution, so the box being watertight becomes more difficult. Thus the solution we found was to apply a waterproof air filter that will protect the electronics of this system while only letting air in; also, the air inlet was designed to be placed in an area that is more protected from the rain.

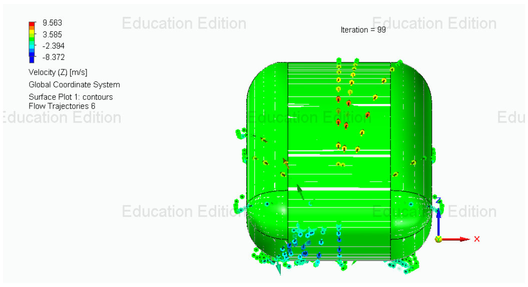


FIGURE 3.17: Aerodynamic Test Front

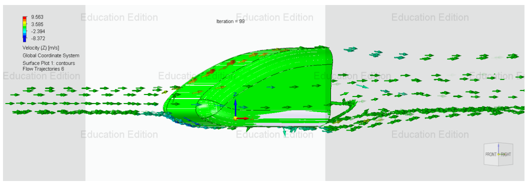


FIGURE 3.18: Aerodynamic Test Front Side

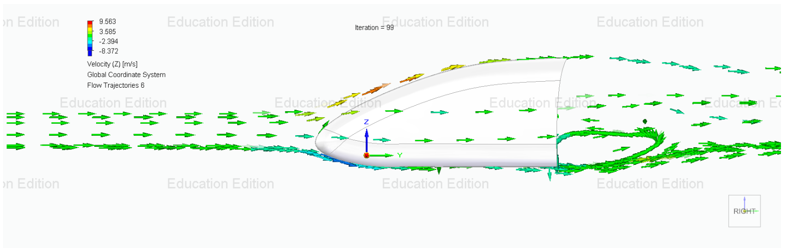


FIGURE 3.19: Aerodynamic Test Lateral

Chapter 4

System Software

This chapter contains the description of the software needed to get the project up and running, namely the server, the database, processing code, and lastly, the web application.

4.1 Hosting

In this project, a hosting service is required to store the software main components, specifically the database and the Web app, as illustrated in figure 4.1. We preferred to have our own server easier to configure. As the server is in our laboratory, it ends up being less costly than subscribing to a hosting service and more convenient to change configurations in the future.

Renting a dedicated server proved to be expensive for this work. We surveyed to see prices of renting Portuguese dedicated servers. As present in [36], [37], and [38] all Virtual Private Server (VPS) services are too expensive as this work is intended to be low cost, the decision to have our server significantly reduced costs.

For our MQTT server, we choose to use Mosquitto because it is a very popular MQTT server or broker, has excellent community support, and is easy to install and configure. So to install Mosquitto and set up our broker to use SSL to secure our password-protected MQTT communications, is needed an ubuntu 16.04 server with a non-root, sudo-enable user and a basic firewall set up.

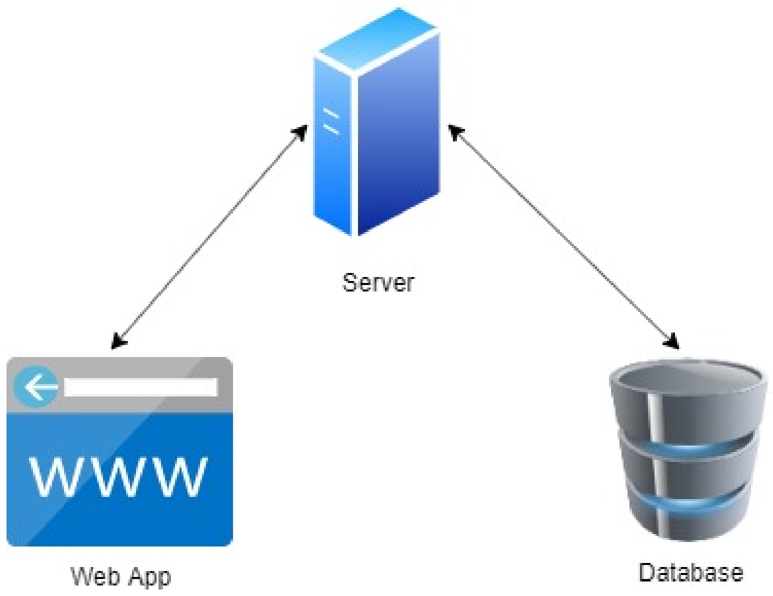


FIGURE 4.1: Hosting Schematics

4.2 Database

PostgreSQL is a relational object database management system developed as an open-source project. PostgreSQL could perform complex and high volume data operations [39].

It uses multi-version concurrency control (MVCC) , which allows several publishers and subscribers to work on the system at once. PostgreSQL is completely capable of handling multiple tasks simultaneously and efficiently. The database chosen was PostgreSQL and not MySQL, Oracle Corporation is the most popular database with over 10 million installations, given the following comparisons. PostgreSQL is open-source and completely free, while MySQL is owned by Oracle and offers commercial versions and a free one. Despite being open-source, PostgreSQL supports the widest range of programming languages like C/C ++, Java, JavaScript, Python, among others. In contrast, MySQL only supports a single language which is not extensible [40].

This is an excellent database for our project because it can install on Debian OS, and it is easy to use, using the PSYCOPG adapter. PSYCOPG has several extensions that allow access to many of the features offered by PostgreSQL.

We organized the database, as illustrated in the diagram of figure 4.2. It is composed of 6 tables:

- **measurement** Each measurement is unique and identified by id as primary key, also with the timestamp and latitude and longitude attribute. This table links to each one of the sensor tables (temphumipres, nPart, gas), and even to the NodeSensors table that has inherited the nodeid attribute as foreign key. It has a many-to-one connection because there are many measurements for a sensor node.
- **temphumipres** This table refers to the measurement made by the temperature, humidity, and pressure sensor, hence receiving each of these values (temp, humi, pres) has a one-to-one connection, because each measurement only corresponds to a temperature value, humidity value, and pressure value, this table is identified by timestamp as primary key from table measurement.
- **nPart** The nPart table refers to the measurement of the number of particles in suspension in the air, have a one-to-one connection with the measurement table because one measurement only has one value of the number of particles and a number of particles corresponds to a measurement, this table is also identified by measurementid as primary key from table measurement.
- **gas** This table, just like the above two, has a connection of one-to-one with the measurement table because one measurement only has one value of gas and otherwise. The table identified by measurementid as primary key from table measurement.
- **NodeSensors** This table has an id on the node sensor as primary key and description of the sensors on this node and is connected to the table measurement on a connection one-to-many where one node could have many measures. Receive as foreign key the busid because it is also connected to the bus table.

- **bus** This table refers to the bus, and as primary key is the bus number, is connected to the table NodeSensors on a connection one-to-one because one bus only has one node sensor and one node sensor are in only one bus, still has licenseplate as an attribute.

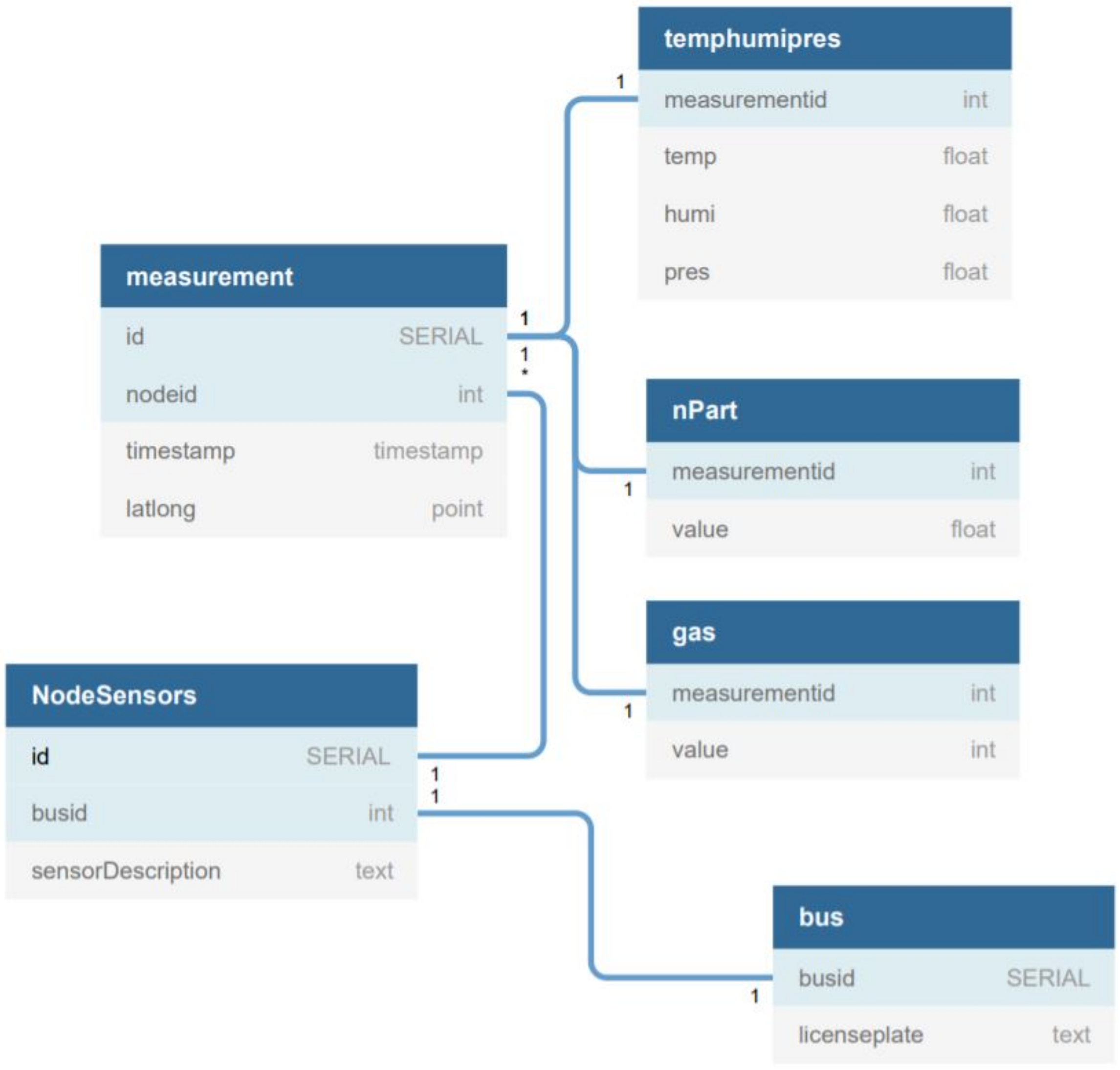


FIGURE 4.2: Database Diagram

4.3 Testing Server

For testing purposes, this project uses Apache through XAMPP. It provides local availability reducing costs, and provides an easy way to detect and correct bugs before making the project available on the web. XAMPP is a cross-platform, open-source server containing MariaDB, the Apache web server, and the interpreters for scripting languages: PHP and Perl. This is only used for supporting the web app [41].

4.4 Node Software

In this section, we will cover the choice of programming language between java and python, and we also present the results in terms of code scripts.

4.4.1 Comparison of programming languages

The choice for processing was Raspberry Pi 3 Model B+, as discussed in Chapter 3 section 3.5, so the code option comes along. The options that we had were java and python. A language comparison is presented in table 4.1. However, the choice between java and python should be made, thinking that the language will be running on Raspberry Pi.

	Java	**Python**
Code	Longer lines of code: public class Hello{ public static void main(String []args){ System.out.println("Hello World");}}	print("Hello World")
Syntax	At the end of statement if semicolon is miss it throws an error. In java is necessary define particular block using braces without it code will not work.	Statement do not need a semicolon to end. In python there is no sight of curly braces but indentation is mandatory. Indentation also improves readability of code.
Dynamic	Necessary to declare type of the data.	Codes are dynamic typed, do not need to declare a type of the variable.
Speed	Faster comparing with python.	Slower, because is an interpreter and also determines data type at runtime.
Easy to use	Java is not easy to use as compared to python because there is no dynamic programming concept and codes are longer than python.	Python codes are shorter than java, follows dynamic programming it is easy to use and easy to understand because of indentation.

TABLE 4.1: Comparison between Java and Python [42]

Python is one of the fastest-growing languages and is older than Java. For Raspberry Pi, most libraries and code examples are python-based. Python also has a broader range of functionalities because the standard Raspbian Linux distribution, which comes with the Raspberry Pi, is well set up for Python, including extensions for Raspberry Pi specific features like GPIO access, which is used for the coding of the dust sensor, seen in figure A.1. Java language requires more computational requirements to run on Raspberry Pi than Python, we have tried the hello world program, and Java takes up to 1 second (it takes up to 1 second every time, so it is not about the cache).

4.4.2 Python Scripts

In figures A.1, A.2 and A.3 from Appendix A, is presented the python code used to put all the peripherals connected to the Raspberry Pi working, collecting data and saving the measurement data on a csv file stored on the SD card. This script has many examples of the easy and efficient way that python works with the RPi. The GPIO is used on this file to set up the dust sensor on A.1 and shows the easy way a file is opened. Otherwise, it is created with that name in figure A.2. In the same picture, it reads the temperature, humidity, pressure, and GPS using serial. At last, in figure A.3 from Appendix A all the peripherals readers are updated (temperature, humidity, and pressure, GPS, dust, and gas), then written on a csv file. On all the three figures, there are also many non declared variables type.

Python also works well with MQTT, the messaging protocol that our systems use to send data has Eclipse Paho as an MQTT library in python. Figure A.4 from Appendix A, represents a script of the publisher client where the Paho, csv, and JSON libraries are imported. On function "on_connect" the Raspberry Pi connects with the broker on port 1883. After connecting to the broker, the file that is to be sent is opened and read as csv file, then "JSON.dumps" returns a string representing a JSON object from each line of the csv. On function "on_publish" is published to the broker a message returned from JSON, using the topic "test_channel". It is also in this script that we define the QoS of the message delivery, in the code line client.publish(MQTT_PATH, message) the QoS=0 by definition, to change the QoS to 2 gets client.publish(MQTT_PATH, message, qos=2), we can confirm that the message was changed with QoS=2 in figure A.6 from Appendix A.

Figure A.5 from Appendix A is a script written in python, but this time is from an MQTT client subscriber. This script was made to test the messaging protocol and see if the data arrived correctly. The script was running on a Windows portable computer. For testing purposes, the computer was also the broker (server), and that is why in the figure below, the variable "MQTT_SERVER" has the correspondence "localhost". This script was written using the text editor Visual Studio Code another tool that works efficiently with python programming. The script starts importing the Paho library and import csv file because it is necessary to organize data into files. In function "on_connect" it connects with the broker on the usually MQTT port 1883 and receives the message from the topic that is subscribed "test_channel." In function "on_message" the message is printed on the screen. If a csv file does not exist to save the data, then one is opened or created.

In picture 4.3 are the real MQTT scenario of the usage of both python scripts were the publisher client is Raspberry Pi, the broker is our Ubuntu server where the database is stored, and the MQTT broker is running. The subscriber client then subscribes to the MQTT topic and stores the data in the database, thus the web application presents the new data.

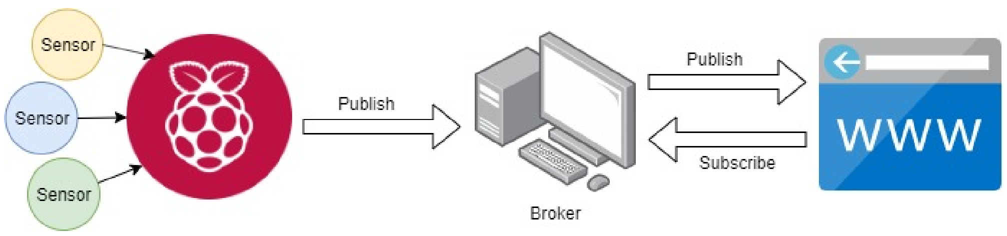


FIGURE 4.3: MQTT Real Scenario

MQTT is designed for reliable communication over unreliable networks, which may be the reason that one of the most popular features is the Quality of Service level of a message that can be sent [43]. With QoS 2, the message is delivered exactly once. The broker might re-deliver a message if the subscriber application did not receive it or did not acknowledge the message. MQTT has another detection of a failure in delivering message built-in, via a heartbeat mechanism, the keep-alive. Each client can define its heartbeat interval with the keep-alive feature when sending a CONNECT message. Other MQTT resilience feature is message queuing. We subscribe to messages with QoS 2, so these messages will be queued for our MQTT client when it is offline. As soon as our client reconnects, these messages will be delivered to the client, without being lost. In the application scenario, the expected latencies of the MQTT protocol working with QoS 2 is 108 ms when connected [44], which is less than 1 second, so it is quite good for the outcome.

4.5 User Interface

This section will approach all the deployment, the code and the output of the Web App and its functionalities.

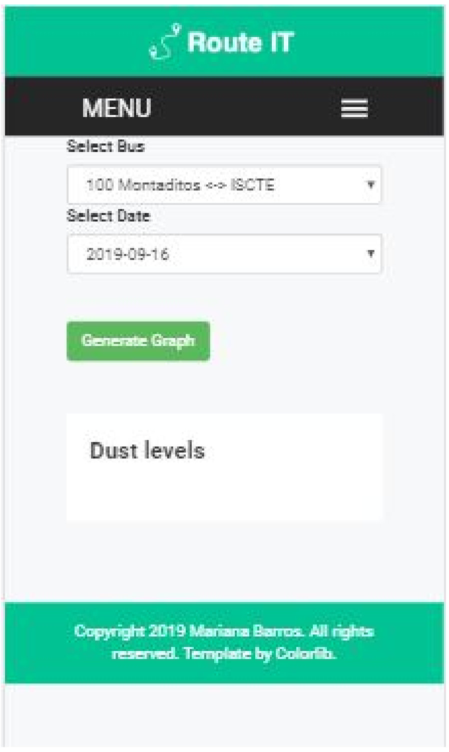
4.5.1 Deployment of the Web App

The developed application is a Web App that contains all the information collected by the sensors and sent to the database. In the following will be explained the main functions of the web app for its users, as noted above.

To access the web application, use the following link:

https://www.intelectrosense.com/routeit

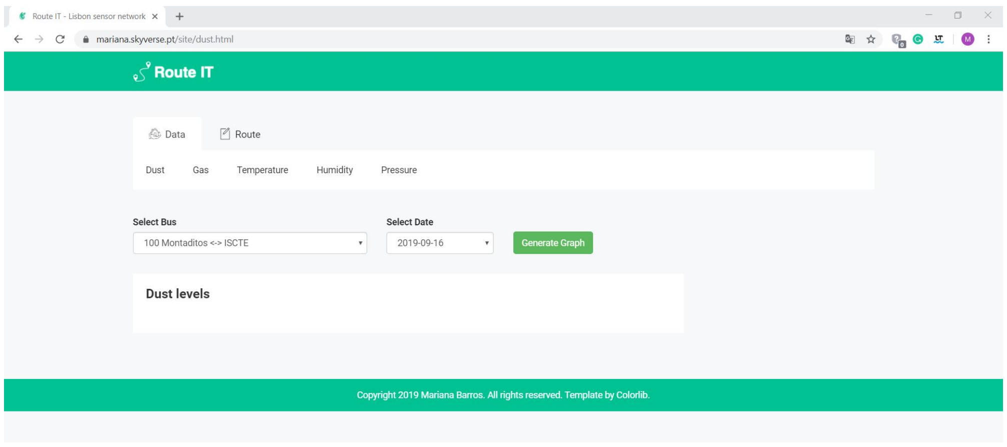
This part of this project has been developed as a tool that allows the user to access the dust, gas, temperature, humidity, and pressure levels to which he is subject in a specific path, or the levels to which he has already been subjected. This tool is built in such a way that the user can access data from previous days. A web app's choice makes it possible to access any operating system, for example, iOS, Android, or Windows, and can be on any device, computer, tablet, or mobile phone. It is compatible with everything, in figure 4.4 is presented the web application's homepage on a mobile phone. The device must have Internet access; that is, a mobile network (3G, LTE) or Wi-Fi network is required to access it.



Copyright 2019 Mariana Barros. All rights reserved. Template by Colorlib.

FIGURE 4.4: Homepage on a phone

After accessing the site on a computer will appear just like in image 4.5. We can choose to see dust, gas, temperature, humidity, and pressure graphs in the Data tab.



Copyright 2019 Mariana Barros. All rights reserved. Template by Colorlib.

FIGURE 4.5: Homepage

Exemplified in figure 4.6 is the mechanism to present the data to the user. Selecting the pretended bus route, the application will verify all the available data for this route on the database and create an option for each day available. After selecting the pretended date, clicking the Generate Graph button will render the graph and present it to the user.

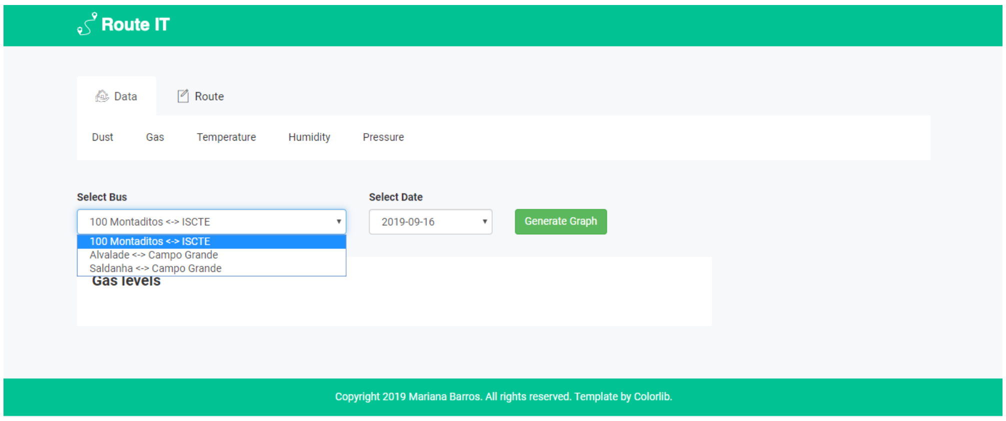


FIGURE 4.6: Route Selection

The following figure 4.7 presents the generated gas graph after following the previous steps. These operations can be done in any of the date tab options, and it will always appear the option to choose the path and the desired date.

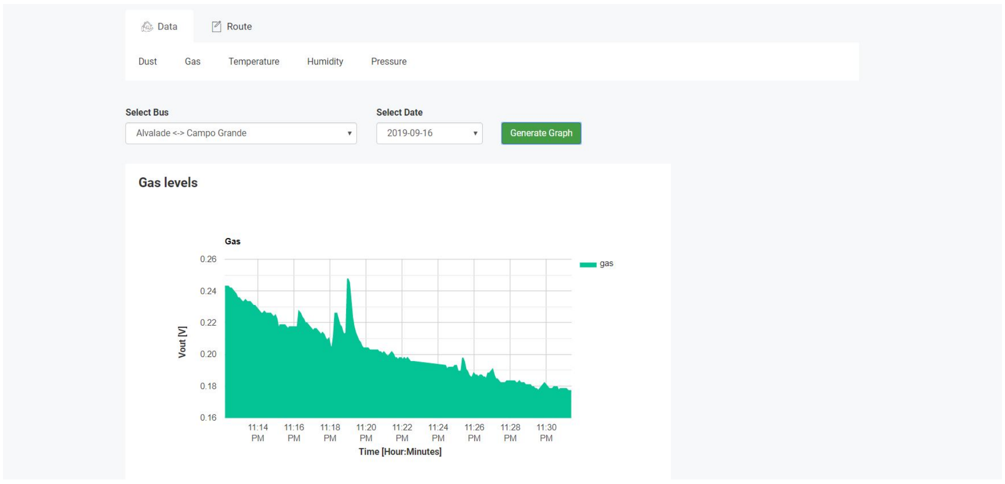


FIGURE 4.7: Gas Generate Graph

In the Route tab, we have two more features of our web app. Pollution HeatMap allows observing in real-time the pollution in some areas of Lisbon, where the areas of greater concentration of pollution are painted in red, less polluted areas in yellow, and virtually unpolluted areas in green. It is possible to zoom in on the map to see better the areas that are to be avoided in the pedestrian path, as can be seen in image 4.8.

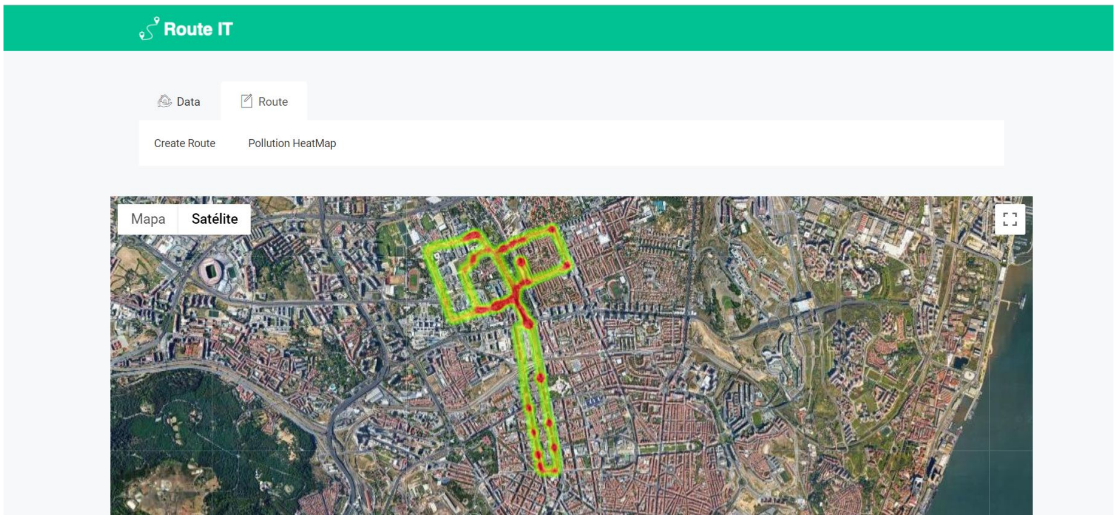


FIGURE 4.8: HeatMap

The web app has another functionality: Create Route. This feature enables the user to choose a starting point of the route by clicking on the map's desired point, which will be highlighted with a marker pin on the map and the same for the endpoint of the route. The web app presents a path to go from the starting point to the arrival point, avoiding zones exposed to pollution.

In image 4.9 is a marker pin in ISCTE and another marker pin in the University Stadium of Lisbon. It is possible to perceive that the route the web app indicates to go from one point to another is not the fastest because the pollution present in the fastest route is being avoided.

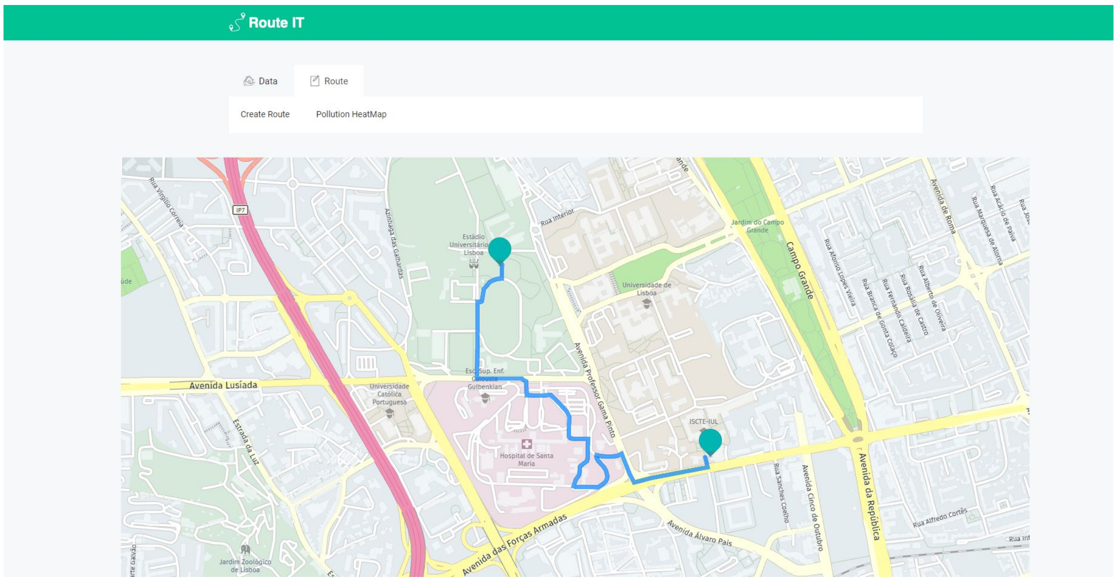


FIGURE 4.9: Route avoiding Polluted areas

On the route of figure 4.10, which has a marker pin in ISCTE and another marker pin in IST, another university in Lisbon, the system completely avoids pollution. In figure 4.8, in the heatmap, we can see that there are many polluted areas on the fastest way between ISCTE and IST. The fastest route is presented in figure 4.11 and comparing the fastest route with the route that avoids pollution, it is possible to see that the second does not pass through the roundabout of Entrecampos, which is an area with high affluence of cars, with traffic lights and consequently has a high concentration of gases and particles in suspension. In the heatmap, it is possible to see well this concentration of pollution. The effectiveness of the chosen route is still possible to verify when the route goes through Av. 5 de Outubro until Campo Pequeno Garden instead of going through Av. da República also avoiding the pollution there present.

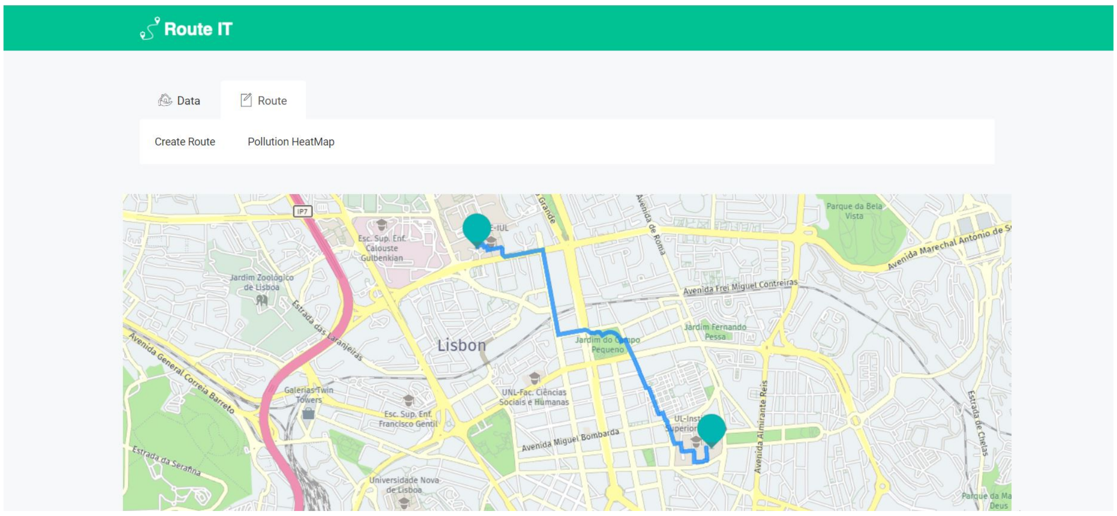


FIGURE 4.10: Route from ISCTE to IST avoiding pollution

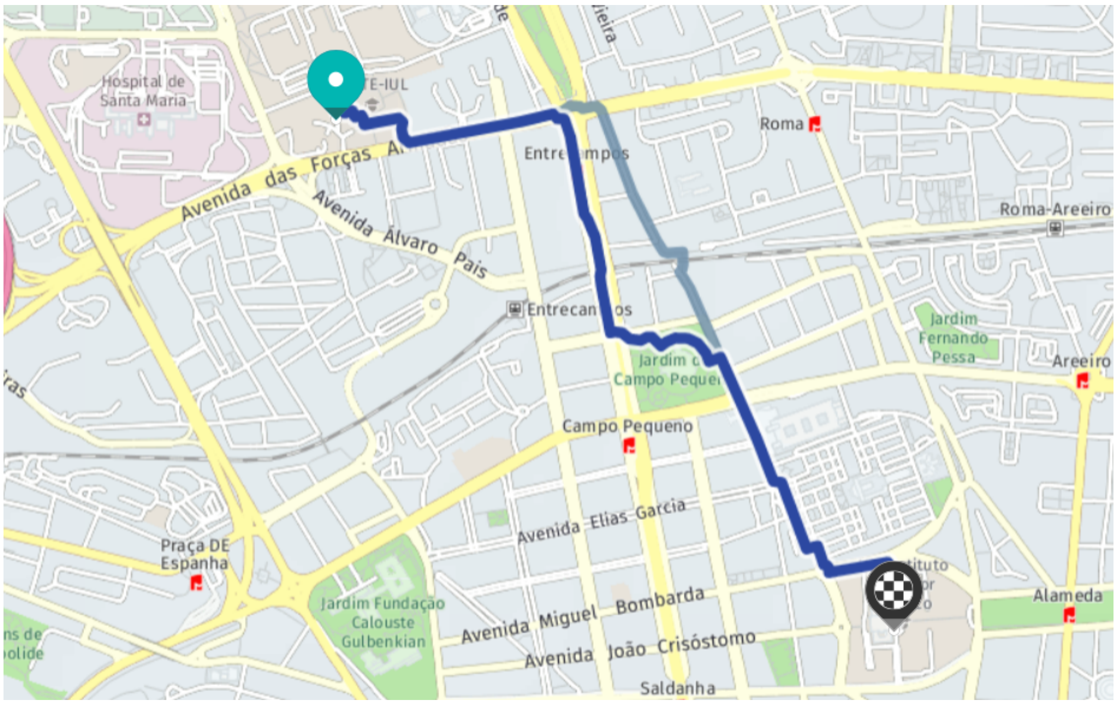


FIGURE 4.11: Fastest Route from ISCTE to IST

4.5.2 Web App Activity

In this Web app, it is unnecessary for any subscription in the application or creating an account. It is a tool for quick access, thinking about the lives of users who move from one place to another in the city of Lisbon.

In the activity diagram, the interaction between the web app and the users is described. It starts on the web application's homepage; then, the user can go to each of the tabs where all the data tab sub-tabs lead the user to choose the route and the date. The graphics or maps are displayed, thus ending the interaction between the application and the user. In the following figure (4.12) is presented the activity diagram:

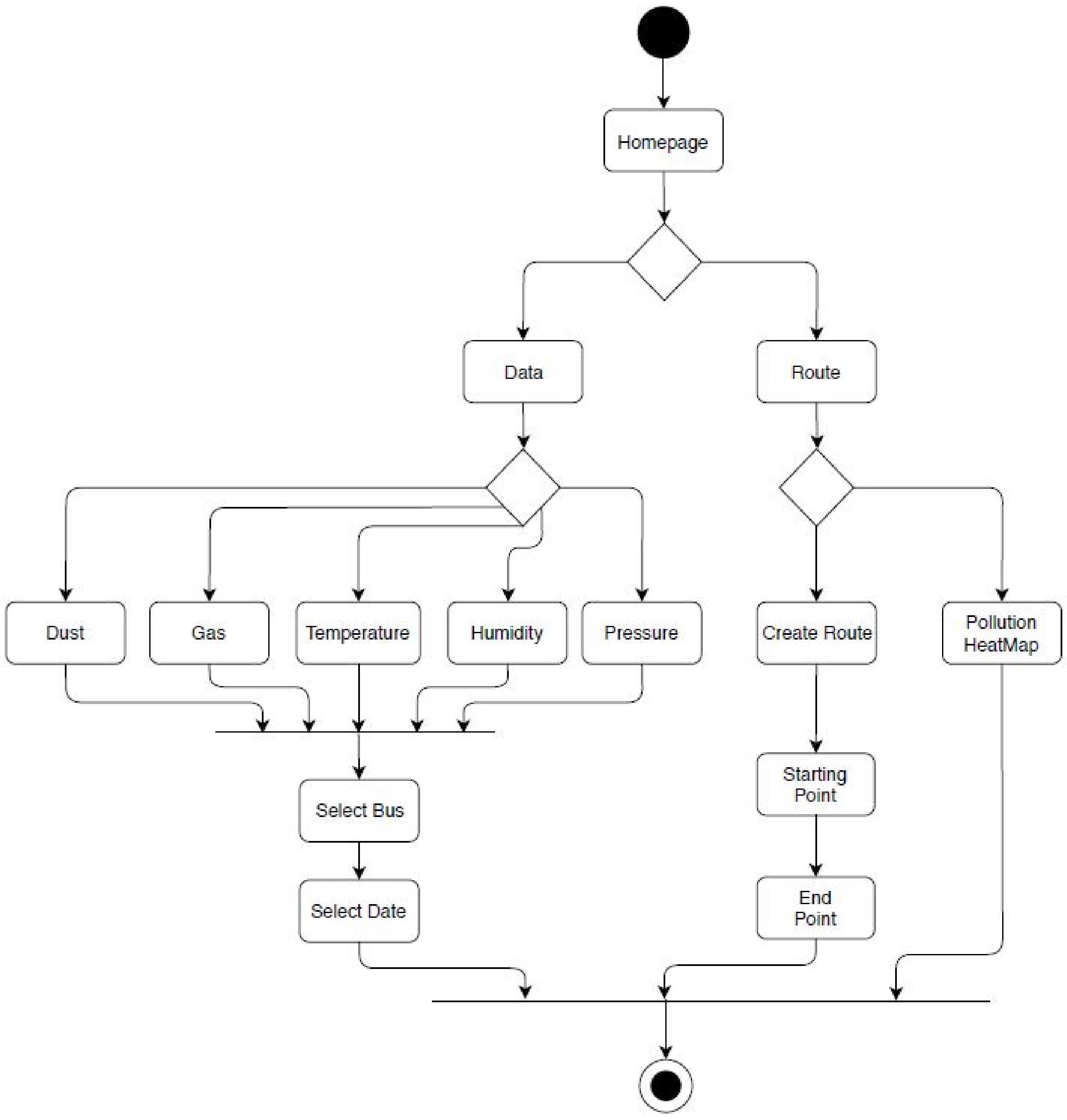


FIGURE 4.12: Activity Diagram

4.5.3 Implementation of the Web Platform

The code developed for the web application to work was HyperText Markup Language 5 (HTML5), that is, each tab of the web page has its HTML page: dust.html, gas.html, temperature.html, humidity.html, pressure.html, createroute.html, and heatmap.html.

A REST Application Programming Interface (API) makes the communication between the client and the database. The REST API uses Hypertext Preprocessor 7 (php7); this language is developed on the server-side to generate dynamic content on the World Wide Web. The PHP scripts contain the code that responds to an HTTP POST request with the parameters such as Date and Bus id to perform the SQL query that returns the data in JSON format with the requested values for that graph or map as illustrated in figure 4.13.

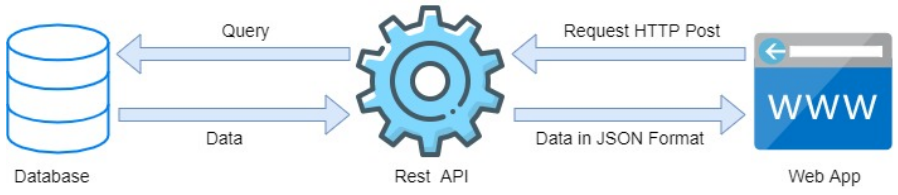


FIGURE 4.13: Data Communication between Client and Database

The heatmap feature uses Google Maps API heatmap layer to provide visual feedback to the application user about the places where the pollution is more concentrated. To implement the heatmap, we created an array with the data formatted in 'latitude, longitude, weight' about each geographic point captured by a node of our system to accomplish a complete city pollution heatmap. The 'weight' is the gas sensor's value and returned by the database divided by 1000. This value changes the color of the point on the heatmap from green to red if there is more pollution and changes the marker's radius, which increases or decreases correspondingly, providing a more significant visual zone on the more gas concentration area. This heatmap feature will work better as the sensor network grows, meaning more buses will give more information regarding pollution zones around Lisbon city.

The Create route feature uses Here Map service that provides a routing algorithm with an optional avoiding area feature to implement the mechanism that allows our application to create a route by placing the departure and arrival point, which avoids pollution in the nearby areas. The Here Map Routing API as an option for selecting the transporting method, and for this project, a pedestrian mode was selected.

To create a route avoiding pollution, the Rest API (createroute.html) will return the available points whose gas value is above the threshold, which by default is set to 280 and can only be increased in the back-end REST API. After a request is made to the database, the API will return the values present above the threshold in the last hours. We made the verification if the points are two kilometers away from the starting point and the endpoint. This verification is made through a function that calculates the distance between coordinates and returns the distance in kilometers, making a primary selection of the points near the route. There are still many points, so the point that was compared before is saved. After, the other point is compared with the following points. If the points are less than 100 meters away, it does not count. This is to avoid having more points because, as mentioned before, the algorithm with many points tends to become very slow and even unresponsive. Then the Rest API of Here Map with the avoiding areas feature makes a route from the departure point from the arrival point, avoiding all the areas polluted.

Chapter 5

Tests and Results

This chapter contains the tests made to the various peripherals attached to the Raspberry Pi and their results. It also includes the calibration made to certain sensors.

5.1 System preliminary prototype

Various tests were done; some of them in an earlier period were done with the system stopped at the same place, as seen in figure 5.1. There are no large fluctuations in temperature, humidity, or pressure, and the GPS was only tested with the exact coordinates. However, as they were made near a traffic light, detecting differences in gases and the number of particles in suspension with the prototype stopped. We also ran tests using the solar panels as the system's power supply, illustrated in figure 5.2. This test was also a fixed-position test because we could not fit the panels to the bike.

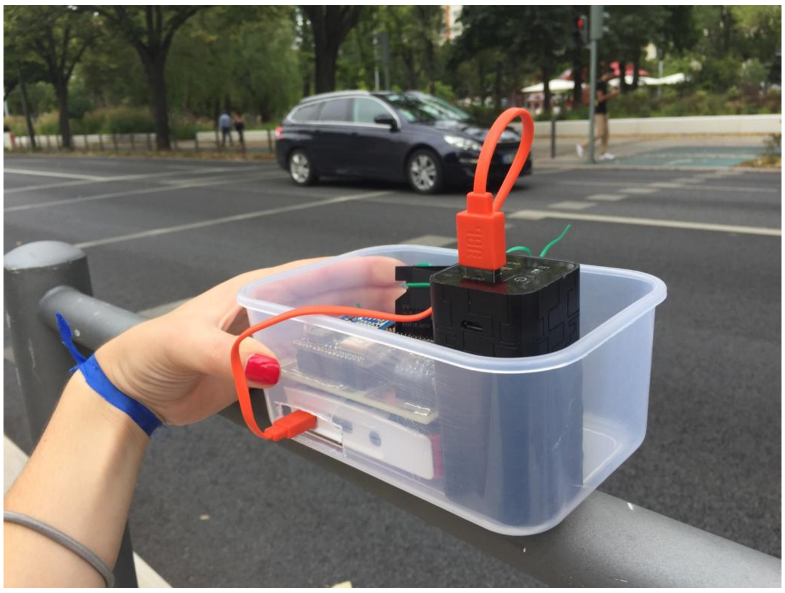


FIGURE 5.1: Fixed-position test

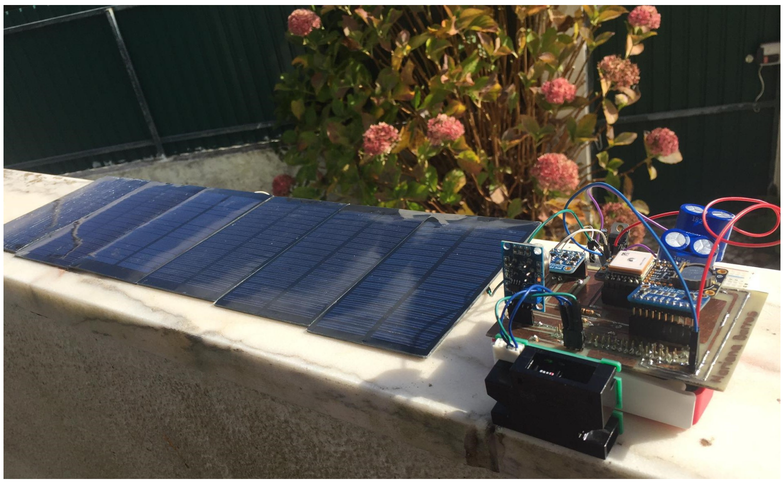


FIGURE 5.2: Stopped Test with Photovoltaic Panels

At a later stage, the tests were made in motion, with the system mounted on a bicycle riding the city of Lisbon, as can be seen in figure 5.3. In this burst of tests, there are changes in all parameters tested, including GPS. For this test, the system was attached to the bike using a plastic case and zip ties.

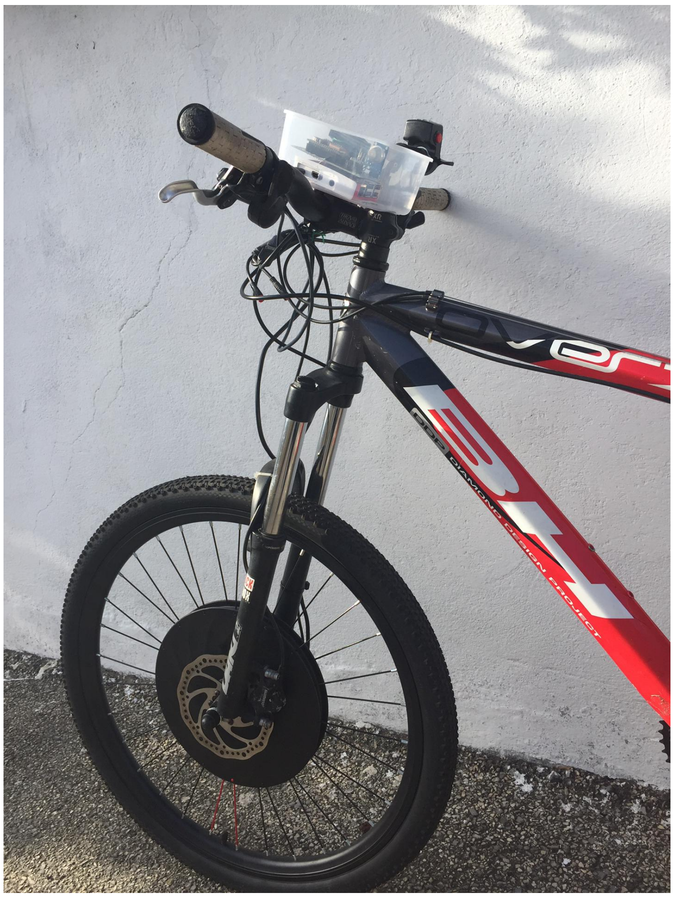


FIGURE 5.3: System Attached to the Bike

With this method, we did tests using the power bank as the power supply of the system because, as mentioned above, we could not fit the panels to the bike. However, only a few tests were done using the power bank as power supply. Already with the idea of the prototype being scalable to several vehicles/places, occurred the idea of taking advantage of the fact that the bike used for the tests was electric and had a USB port that allowed the charging of devices. We took advantage of this to power the system. In figure 5.4 and 5.5, can see the Raspberry Pi connected to the battery of the bicycle wheel.

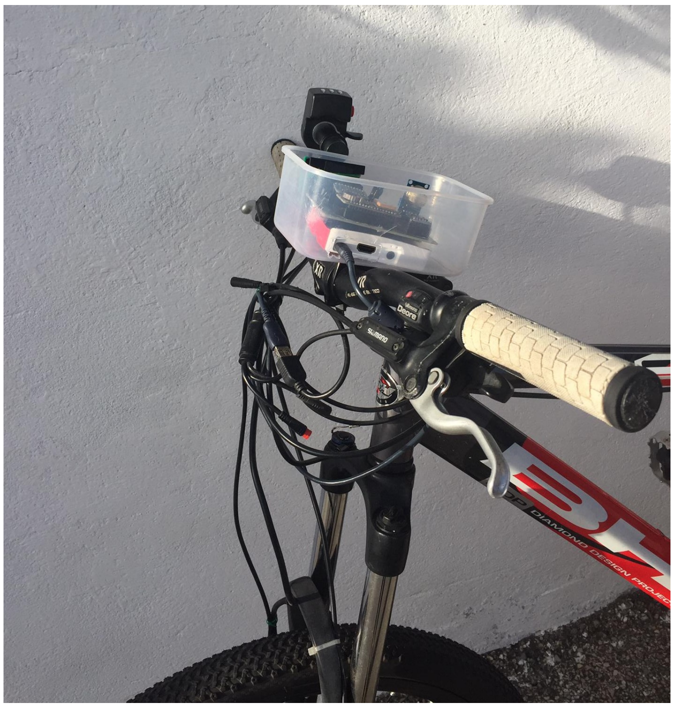


FIGURE 5.4: System Attached to the Bike Side View

FIGURE 5.5: System Powered by Bike Battery

5.1.1 Results

The tests were made using several different materials to test the various peripherals. Beyond this, all tests made with the bike were monitored using the Strava application to record the course, duration, and speed of the ride.

5.1.2 Photovoltaic Panels Test

We characterize the effectiveness of the solar panels according to the various possible levels of sun exposure to evaluate the performance of the system during a day experimentally.

To test the panels, we use a Keithley 2000 multimeter to measure the current consumed by the system when the power supply source is the photovoltaic panels. In figure 5.6 is illustrated the resulting graph. It can be seen that the panels can provide about 600mA that the system needs to run the peripherals. On the day this test was done, the sky was cloudy, so it is possible to see in the graph of figure 5.6 that a little before one in the afternoon, that panels reduced the generation of energy to 300mA. This is because a cloud passed and prevented the sun from reaching so intensively the panels, thus the supercapacitors prevent system shutdown in the event of low solar radiation. The consumption of Raspberry Pi was relatively constant, and the photovoltaic panels were able to keep the system running without using other energy sources.

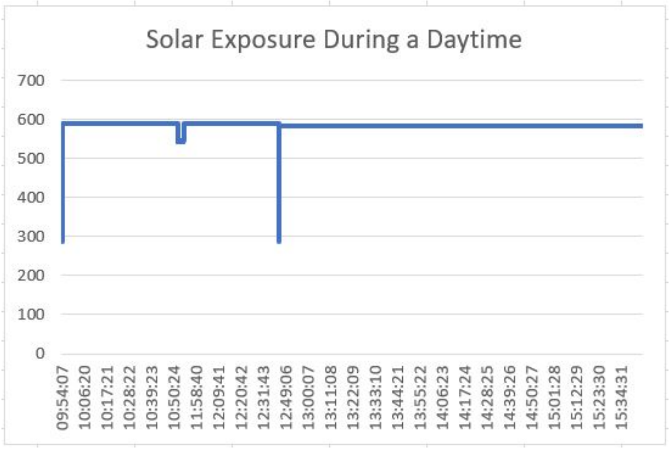


FIGURE 5.6: Solar Exposure During a Daytime

5.1.2.1 GPS Test

In the initial tests, as the system was static, the coordinates were also static. To confirm the GPS position, the sensor gave us used tools such as Google Maps or any other GPS tool via a website or mobile application, which was enough given the situation. The samples were taken every five seconds after the GPS has a fix.

Later, when the tests became in constant motion, the solution adapted before, for when the tests were static, now did not work. Since the system was driven by a bicycle, it was decided to use the Strava application that monitors the entire route. Since GPS samples are taken every 5 seconds, the higher the ride speed, the fewer samples will be taken.

Then the csv files that resulted from the rides were treated. The other sensors and timestamp information was deleted, leaving only the GPS information, latitude, and longitude. With the help of a web tool, MyGeoData, we inserted the csv already treated only with the latitude and longitude. This website converts into a map the GPS positions of several points. In figure 5.7 is the map extracted from the Strava application with a test ride that was made. In figure 5.8, the result of the MyGeoData tool conversion, we see only half of that ride because, in a ride, we generated two csv files, only the first file generated in that ride was introduced in MyGeoData. We can then observe that all GPS points removed by the system correspond to real points. This tool was used to test our GPS system, which proved to work correctly.

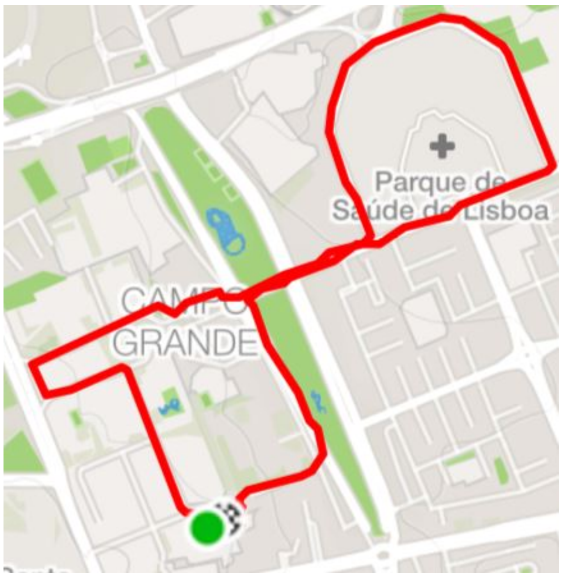


FIGURE 5.7: Strava Ride

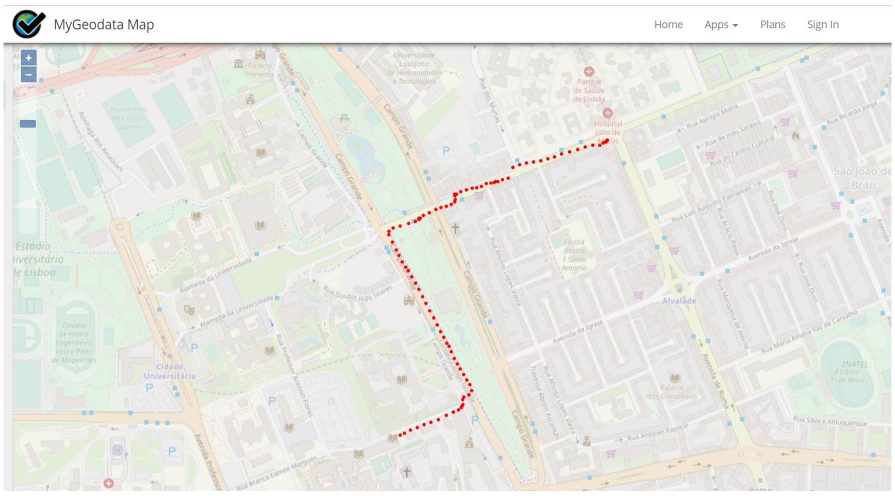


FIGURE 5.8: GPS Test

5.1.2.2 Gas Test

The gas sensor initially went through a simple calibration process before being laboratory tested. With the use of a gas torch, gas was flushed near the sensor as can be seen in figure 5.9, the values immediately started to rise.

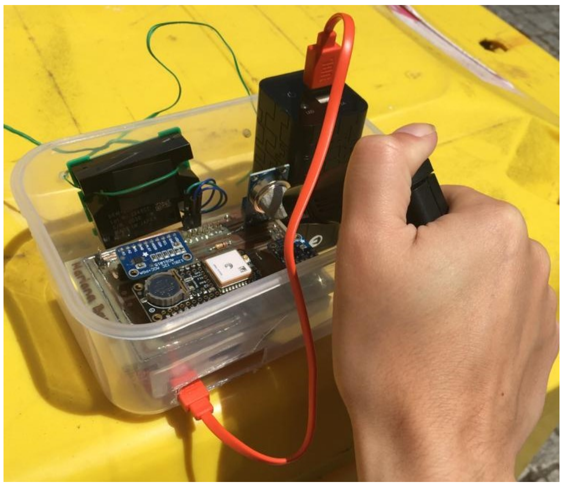


FIGURE 5.9: Gas Test

To calibrate this sensor without resorting to expensive laboratory material, a gas test chamber was created out of a plastic container with a hole in the middle to allow solvent dispensing access. The hole is isolated with a piece of electrical tape, as can bee seen in figure 5.10. To test the sensor, draw a small amount of solvent into the syringe and dispense it into our test chamber. The sensor responds after the solvent is dispensed into the chamber. Using sequential injections can see how the sensor reacts to higher concentrations of volatile organic compounds (VOC).

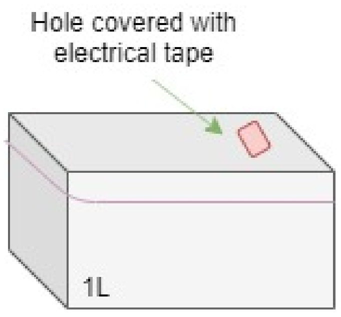


FIGURE 5.10: Gas Test Chamber

The output of the sensor is not in any unit is just a count from 0 to 4095, so to a better perception, we must make calculations to get values in parts per million (ppm). Taking into account that the ADC is 12 bit, so the count goes up to 4095, 2 raised to 12 is 4096, the output value can be converted to Volts through the expression 5.1:

$$V_{out} = \frac{ADCoutput}{4095} \times 5 \tag{5.1}$$

The units were changed to ppm using alcohol because it was very dangerous to use other types of gases such as CO2 without proper laboratory equipment that would protect the user. Using the volume of the test chamber made, and the sensor datasheet, it was calculated that 0.05 ml is the value that must be inserted in the chamber to obtain the Ro. The Rs is obtained from the equation present in 5.2, where Rl is the resistance value of the sensor which, according to the datasheet, is 20 Kohms:

$$Rs = Rl \times (\frac{5}{V_{out}} - 1) \tag{5.2}$$

After obtaining the Ro and Rl, the trend line of the chart in the datasheet is calculated and the equation, present in 5.3, that allows calculating the output in ppm is taken.

$$[ppm] = a \times (\frac{Rs}{Ro})^b \tag{5.3}$$

The value of Rs is calculated in the equation above. Ro is the value of Rs for which the Vout establishes after the test, in graph present in figure 5.11, it is possible to see the values stabilize during the alcohol test. The a and b values correspond to the values of 4,1959 and -0,324, correspondingly taken from the trend line of the calibration values of the datasheet and president on figure 5.12. Then using equation 5.3 to calculate the output of the sensor in ppm during the alcohol test, the value for which the curve stabilizes corresponds to 41 ppm, the sensor is well calibrated since it should be about 50 ppm. The fact that the value is not closer to the ideal is that the pipette used to inject the alcohol into the test chamber is not a precision pipette.

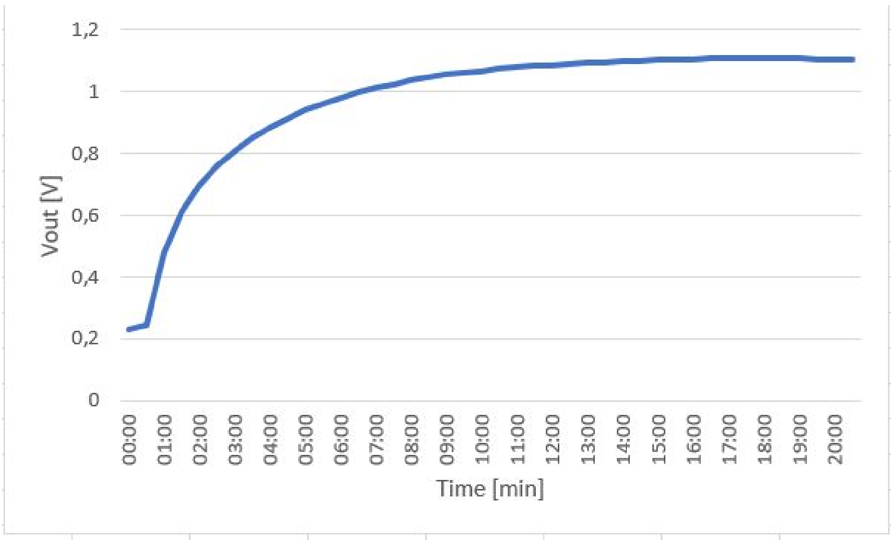


FIGURE 5.11: Alcohol Test Vout Curve

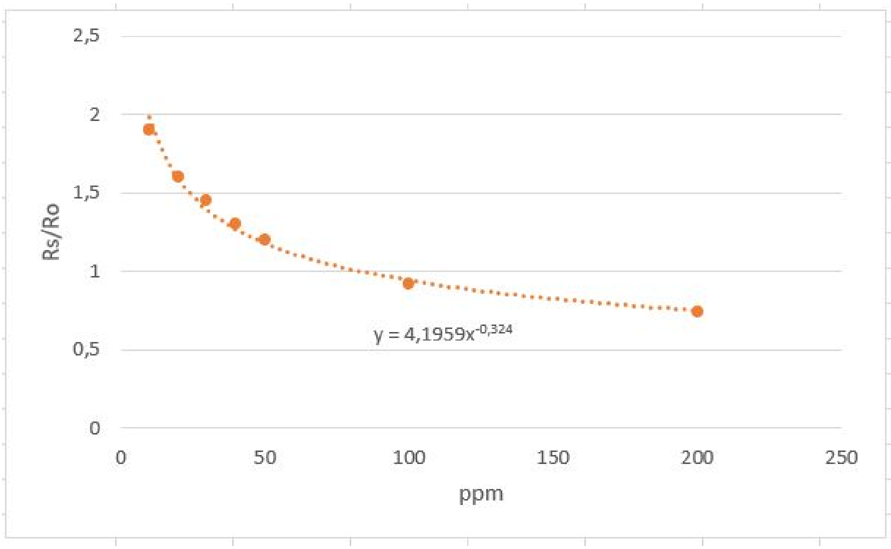


FIGURE 5.12: Alcohol Calibration Trend Line

5.1.2.3 Temperature Test

The temperature test was done indoors using conventional air conditioning. More than one test was done, all following the same model, it took about 30 minutes in which initially the air conditioner was turned off, the outside temperature was 31ºC, the temperature sensor gave 27ºC, turning on the air conditioner gave 26ºC, the extra degree that the sensor was detecting can be because Raspberry Pi was already working for some time and has heated up a bit. After about 10 minutes, the air conditioner was turned on and set to a temperature of 20ºC. After 20 minutes, the air was already at 20ºC, and the sensor measured 21ºC, this degree more is due to the fact already mentioned above that the Raspberry Pi was already working for some time and has warmed up a bit, and the sensor was placed right above.

Another test was done on a less hot day, outside it was 23ºC, it was used the same test model of the previous day, but this time the air conditioner was used to heat the room instead of cooling it down. The sensor measured 21ºC, and the air conditioner gave 20ºC after 10 minutes, the air conditioner was turned on and programmed to heat the environment to 30ºC, 20 minutes after turning on, the environment was 30ºC, and the sensor measured 32ºC, the two degrees more is because the RPi has been on for some time, however, it does not only measure one more degree but two because we are working with higher temperatures and, usually, it tends to heat more.

5.1.2.4 Dust Test

Particle sensor testing was much more difficult. It was necessary to opt for a much more archaic solution. As mentioned earlier, this sensor does not have any libraries available, so the tests made to this sensor were simply with household dust. This means that we did not need a bit of dust for the sensor input, and with the sensor script running on Raspberry Pi, we observed the values increase significantly. Another way that was also used to test the sensor's functioning was with a strong blow near the sensor. This way, we could also observe the values increasing but a less accentuated increase than when you were using the dust, where it can be seen a more significant increase in the sensor values.

We also opted for a third way to test the sensor, putting it near an exhaust pipe of a car with the car working. In this last approach, it was possible to verify that the sensor's values reached the highest importance that we had ever got. Hence we can conclude that the sensor is working well.

Figure 5.13 presents the graph that resulted from the above test, from the 30 seconds there were some strong blows near the sensor, there is an increase in the number of particles up to 1 minute and a half, then returns to zero. Around 3 minutes and 3 and a half minutes, the hands were clapped with domestic dust. Here, there is already an increase in the number of particles about the breath. Then for some time, there was nothing close to the sensor. The particles' value was zero, at 8 minutes, the system was placed close to a car, and the vehicle was started. The vehicle was accelerated twice at 8.5 minutes and at 9 minutes, where the highest values of the number of particles in suspension were obtained.

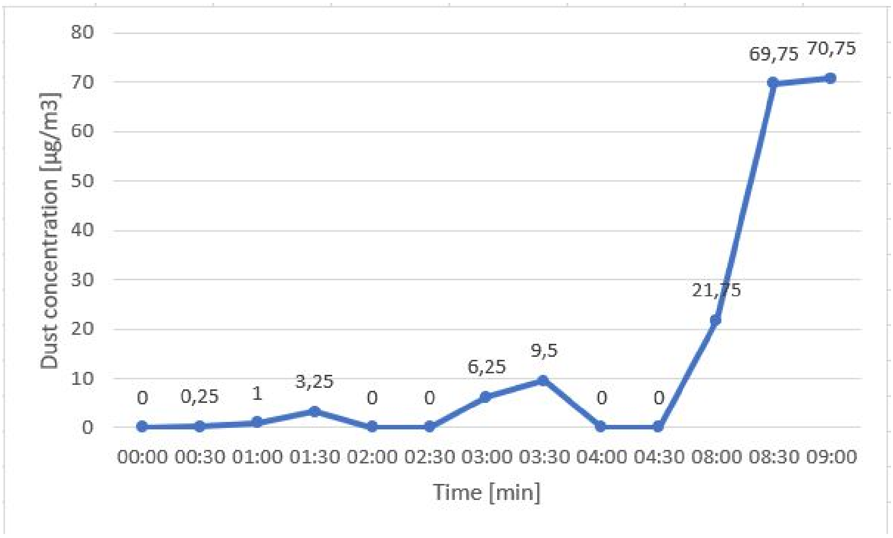


FIGURE 5.13: Dust Test Graph

Chapter 6

Conclusions and Future Work

This final chapter will address the conclusions drawn from all parts of the system development and mention what could be the follow-up of this project as future work.

6.1 Conclusions

In this project, we proposed a scalable mobile system that measures environmental pollution. We produced the system prototype in which all the objectives were achieved except for a watertight box that would protect the system from different weather conditions. However, the box was thought and designed, as mentioned in Chapter 3.

Another part of the project that could have been further developed was the number of tests performed. We could have carried out more tests under different conditions. However, all the tests were done by bike, so it became more difficult to have a more significant number of tests comparing to using the Lisbon bus public network as initially planned. Also, the material for the second prototype already came very close to the deadline for the dissertation's delivery. It was still necessary to assemble every component of the system and put everything to work as a single system and not just a set of components. Only then we can move to the testing phase.

Besides the not-so-well developed things, the whole project has much work involved until the final product arrives. This project had the purpose of being low-cost, which involved hardware and software in a single and complete work. The hardware is composed of pollution monitoring sensors that collect data and send these data via a wireless internet connection to the database. The data is shown in the Web App, that among other features, also allows the user to create a route with a point of origin and point of destination that avoids polluted areas. The final system is only capable of producing this entire work because of the combination of the various software and hardware components that have been used. The sensors and GPS have most of the libraries available for download that are compatible with Raspberry Pi and Python, the language used to encode the hardware components. The fact that the system has various forms of power supply enables alternative applications either by green energy or regular. The wireless communication works with MQTT, the protocol chosen for transmission of messages, sending the data to the database. The database was developed in PostgreSQL. The Web App will get the information from the database showing it on the site and display the heatmap and the routes that prevent pollution.

This project does not yet exist in the city of Lisbon and is a different idea because it allows users to make an informed decision. With the heatmap consultation, it is possible to see that many paths we make almost daily, in the long run, can be harmful to our health, and with the part that creates the less polluted route, it is often possible to choose a slightly longer path but with less pollution. This new path, made years after years, can significantly benefit the health of a person. Our health is very important, it is what allows us to continue to live and create projects like this.

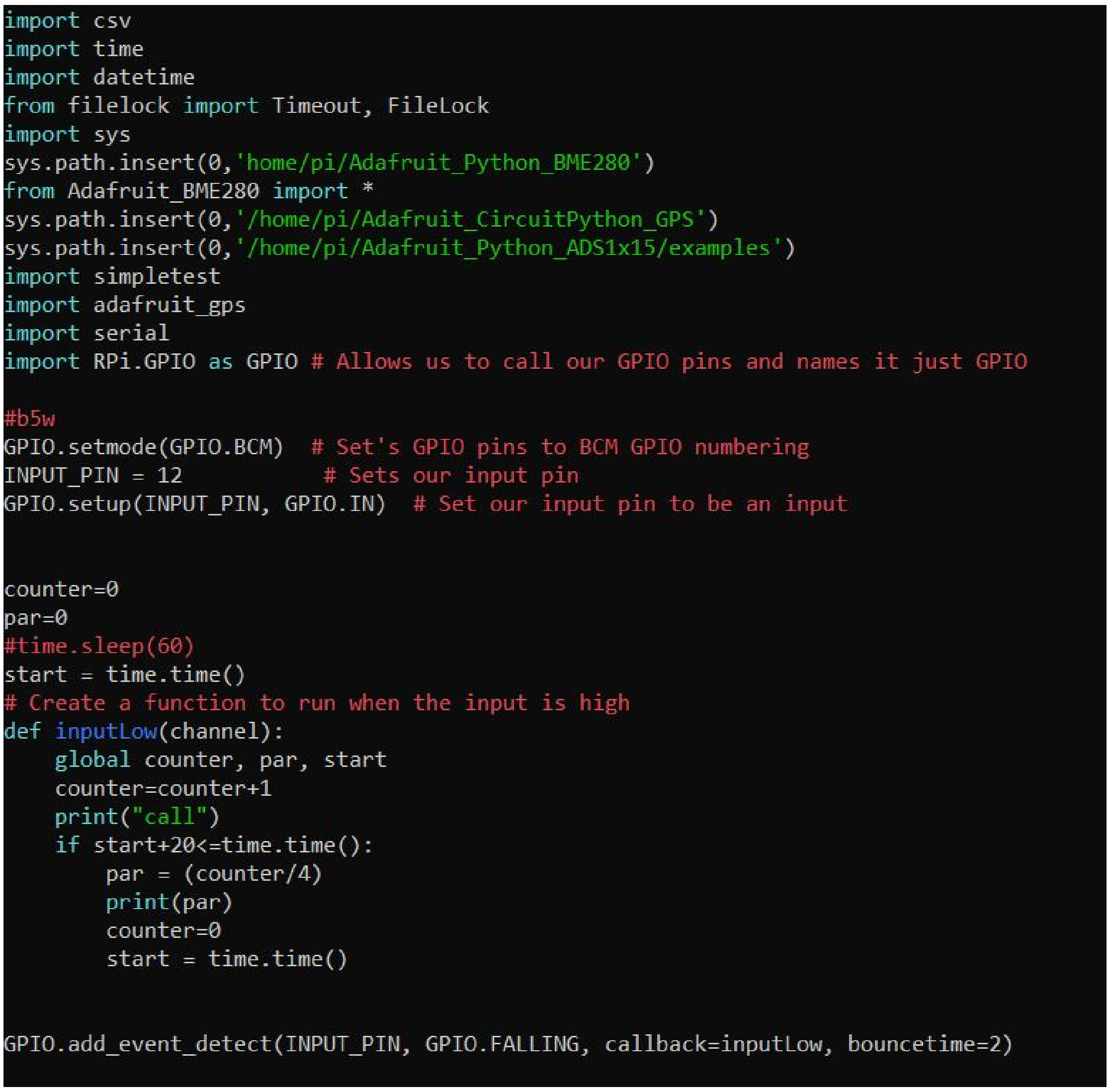
6.2 Future Work

As a follow-up to this project, the system could be operated with only photovoltaic panels and batteries on electric bicycles. More and more users of this transport method in Lisbon, such as Jump bikes or Gira, rent a bike and could while traveling, collect gather measurements from where it passes. The system would only be activated when the bike was rented, and the battery could be charged by the photovoltaic panel or when the bike was at their parking being charged. In this way, there would be even more results for the web app. The bigger the data acquired, the better the route planner algorithm will work and provide better route alternatives avoiding highly polluted areas to the end-user of the web app. This takes us to the next to improve an aerodynamic box, where the air could come in and come out of the box, but rain does not come in not to damage the electronic components.

Appendices

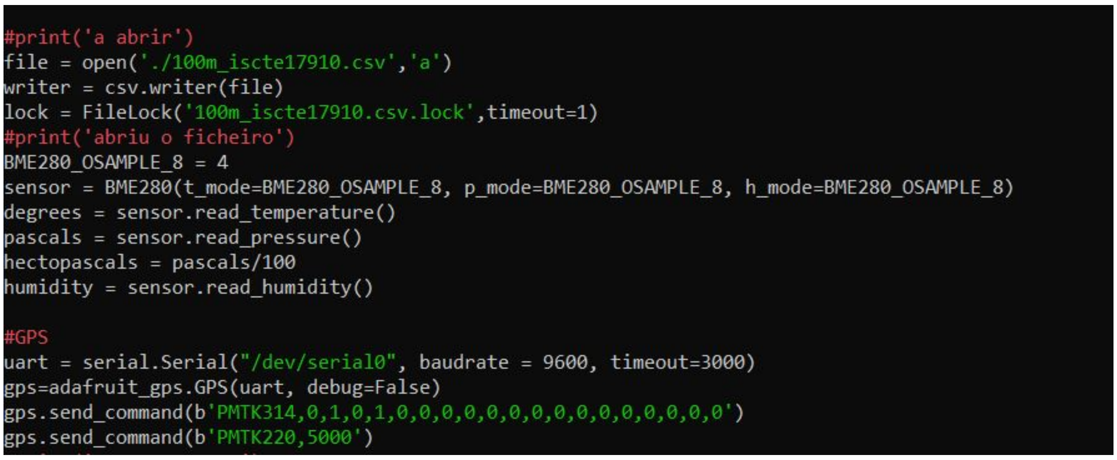
Appendix A

Python Scripts



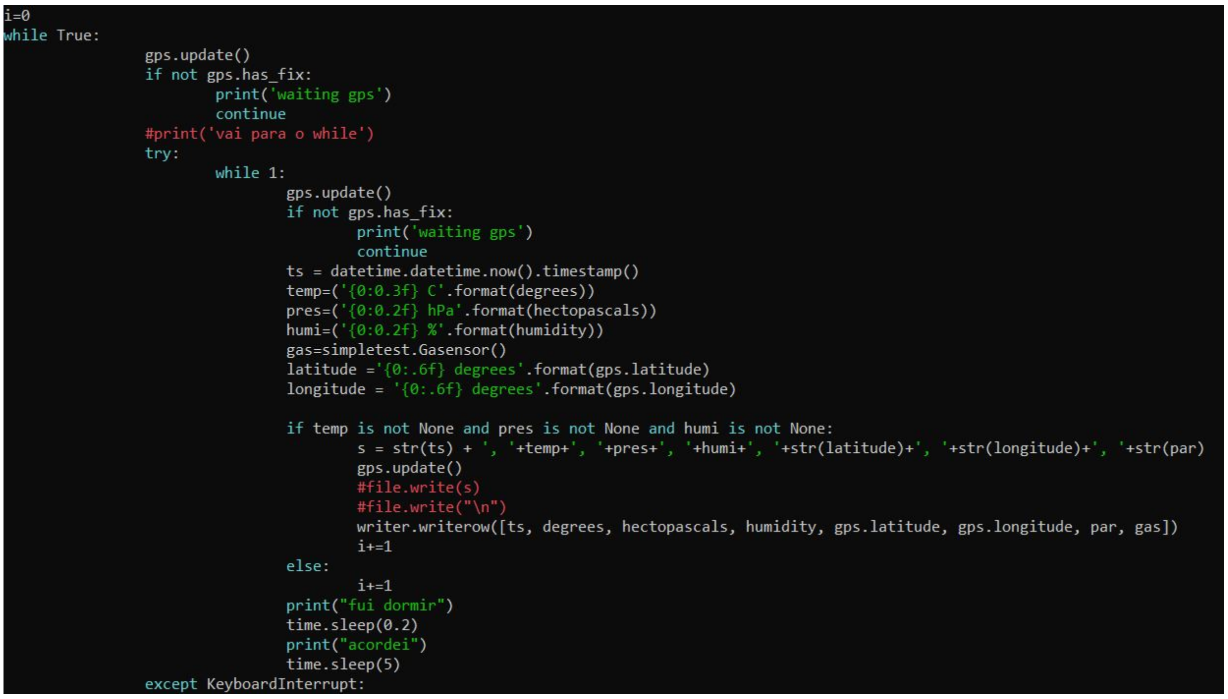
```
import csv
import time
import datetime
from filelock import Timeout, FileLock
import sys
sys.path.insert(0,'home/pi/Adafruit_Python_BME280')
from Adafruit_BME280 import *
sys.path.insert(0,'/home/pi/Adafruit_CircuitPython_GPS')
sys.path.insert(0,'/home/pi/Adafruit_Python_ADS1x15/examples')
import simpletest
import adafruit_gps
import serial
import RPi.GPIO as GPIO # Allows us to call our GPIO pins and names it just GPIO

#b5w
GPIO.setmode(GPIO.BCM)  # Set's GPIO pins to BCM GPIO numbering
INPUT_PIN = 12           # Sets our input pin
GPIO.setup(INPUT_PIN, GPIO.IN)  # Set our input pin to be an input


counter=0
par=0
#time.sleep(60)
start = time.time()
# Create a function to run when the input is high
def inputLow(channel):
    global counter, par, start
    counter=counter+1
    print("call")
    if start+20<=time.time():
        par = (counter/4)
        print(par)
        counter=0
        start = time.time()


GPIO.add_event_detect(INPUT_PIN, GPIO.FALLING, callback=inputLow, bouncetime=2)
```

FIGURE A.1: Script for Exporting Data to csv File part 1



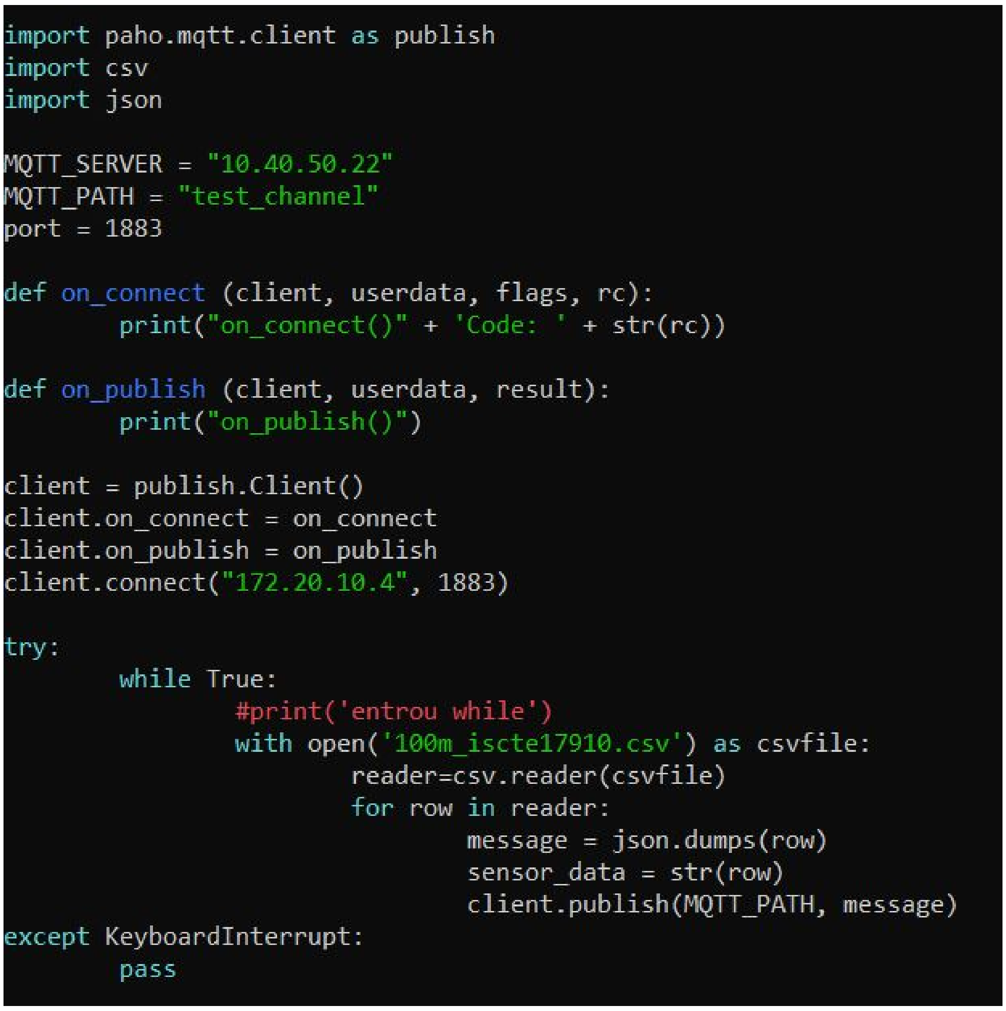
```
#print('a abrir')
file = open('./100m_iscte17910.csv','a')
writer = csv.writer(file)
lock = FileLock('100m_iscte17910.csv.lock',timeout=1)
#print('abriu o ficheiro')
BME280_OSAMPLE_8 = 4
sensor = BME280(t_mode=BME280_OSAMPLE_8, p_mode=BME280_OSAMPLE_8, h_mode=BME280_OSAMPLE_8)
degrees = sensor.read_temperature()
pascals = sensor.read_pressure()
hectopascals = pascals/100
humidity = sensor.read_humidity()

#GPS
uart = serial.Serial("/dev/serial0", baudrate = 9600, timeout=3000)
gps=adafruit_gps.GPS(uart, debug=False)
gps.send_command(b'PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0')
gps.send_command(b'PMTK220,5000')
```

FIGURE A.2: Script for Exporting Data to csv File part 2



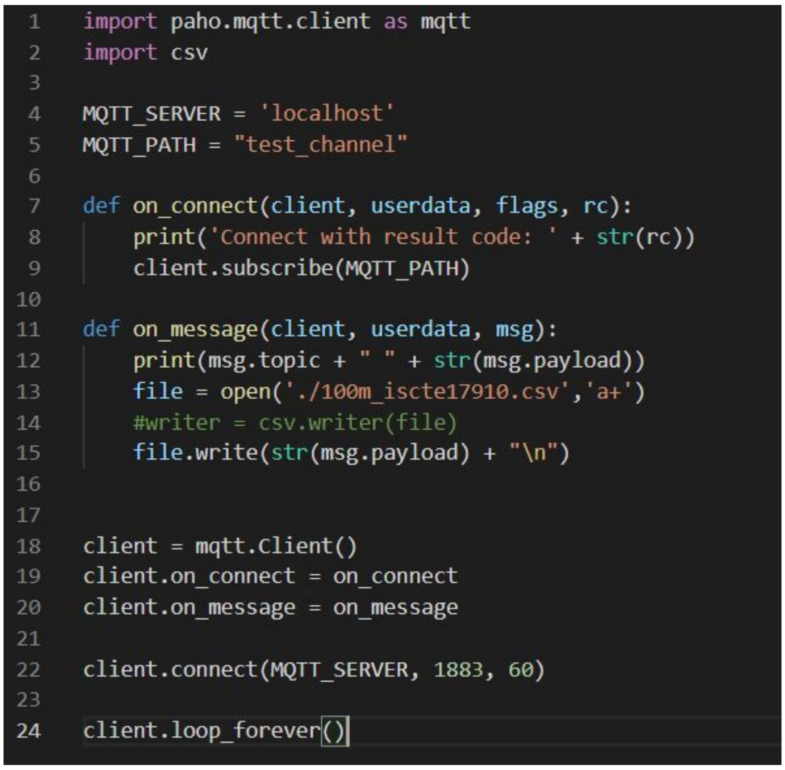
```
i=0
while True:
                gps.update()
                if not gps.has_fix:
                        print('waiting gps')
                        continue
                #print('vai para o while')
                try:
                        while 1:
                                gps.update()
                                if not gps.has_fix:
                                        print('waiting gps')
                                        continue
                                ts = datetime.datetime.now().timestamp()
                                temp=('{0:0.3f} C'.format(degrees))
                                pres=('{0:0.2f} hPa'.format(hectopascals))
                                humi=('{0:0.2f} %'.format(humidity))
                                gas=simpletest.Gasensor()
                                latitude ='{0:.6f} degrees'.format(gps.latitude)
                                longitude = '{0:.6f} degrees'.format(gps.longitude)

                                if temp is not None and pres is not None and humi is not None:
                                        s = str(ts) + ', '+temp+', '+pres+', '+humi+', '+str(latitude)+', '+str(longitude)+', '+str(par)
                                        gps.update()
                                        #file.write(s)
                                        #file.write("\n")
                                        writer.writerow([ts, degrees, hectopascals, humidity, gps.latitude, gps.longitude, par, gas])
                                        i+=1
                                else:
                                        i+=1
                                print("fui dormir")
                                time.sleep(0.2)
                                print("acordei")
                                time.sleep(5)
                except KeyboardInterrupt:
```

FIGURE A.3: Script for Exporting Data to csv File part 3



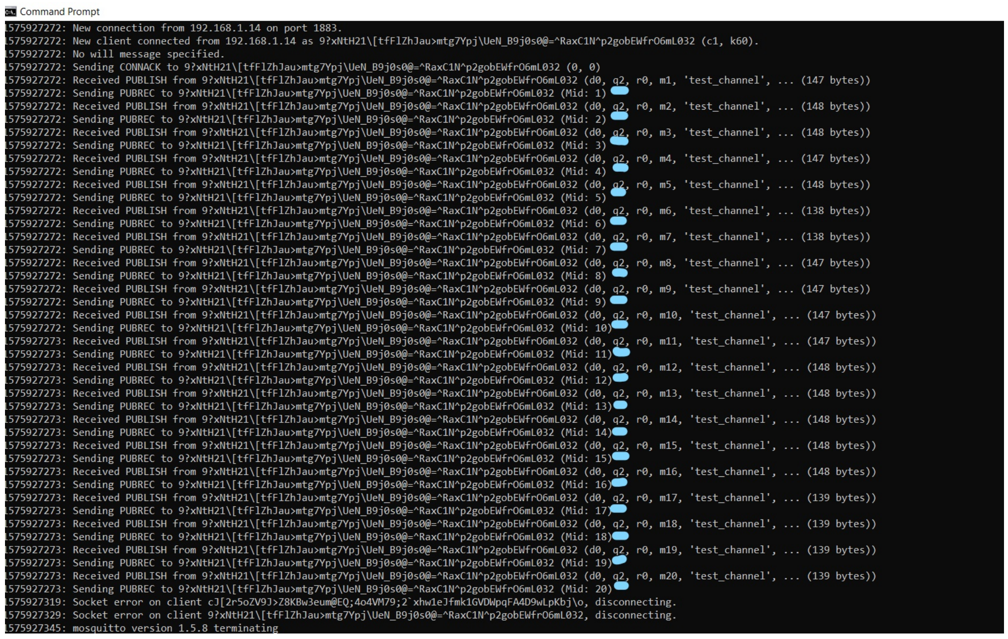
```
import paho.mqtt.client as publish
import csv
import json

MQTT_SERVER = "10.40.50.22"
MQTT_PATH = "test_channel"
port = 1883

def on_connect (client, userdata, flags, rc):
        print("on_connect()" + 'Code: ' + str(rc))

def on_publish (client, userdata, result):
        print("on_publish()")

client = publish.Client()
client.on_connect = on_connect
client.on_publish = on_publish
client.connect("172.20.10.4", 1883)

try:
        while True:
                #print('entrou while')
                with open('100m_iscte17910.csv') as csvfile:
                        reader=csv.reader(csvfile)
                        for row in reader:
                                message = json.dumps(row)
                                sensor_data = str(row)
                                client.publish(MQTT_PATH, message)
except KeyboardInterrupt:
        pass
```

FIGURE A.4: Client Publisher



```python
import paho.mqtt.client as mqtt
import csv

MQTT_SERVER = 'localhost'
MQTT_PATH = "test_channel"

def on_connect(client, userdata, flags, rc):
    print('Connect with result code: ' + str(rc))
    client.subscribe(MQTT_PATH)

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))
    file = open('./100m_iscte17910.csv','a+')
    #writer = csv.writer(file)
    file.write(str(msg.payload) + "\n")


client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(MQTT_SERVER, 1883, 60)

client.loop_forever()
```

FIGURE A.5: Client Subscriber



Command Prompt

```
1575927272: New connection from 192.168.1.14 on port 1883.
1575927272: New client connected from 192.168.1.14 as 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (c1, k60).
1575927272: No will message specified.
1575927272: Sending CONNACK to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (0, 0)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m1, 'test_channel', ... (147 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 1)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m2, 'test_channel', ... (148 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 2)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m3, 'test_channel', ... (148 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 3)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m4, 'test_channel', ... (147 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 4)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m5, 'test_channel', ... (148 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 5)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m6, 'test_channel', ... (138 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 6)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m7, 'test_channel', ... (138 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 7)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m8, 'test_channel', ... (147 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 8)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m9, 'test_channel', ... (147 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 9)
1575927272: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m10, 'test_channel', ... (147 bytes))
1575927272: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 10)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m11, 'test_channel', ... (147 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 11)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m12, 'test_channel', ... (148 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 12)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m13, 'test_channel', ... (148 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 13)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m14, 'test_channel', ... (148 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 14)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m15, 'test_channel', ... (148 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 15)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m16, 'test_channel', ... (148 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 16)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m17, 'test_channel', ... (139 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 17)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m18, 'test_channel', ... (139 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 18)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m19, 'test_channel', ... (139 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 19)
1575927273: Received PUBLISH from 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (d0, q2, r0, m20, 'test_channel', ... (139 bytes))
1575927273: Sending PUBREC to 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032 (Mid: 20)
1575927319: Socket error on client cJ[2r5oZV9J>Z8KBw3eum@EQ;4o4VM79;2`xhw1eJfmk1GVDWpqFA4D9wLpKbj\o, disconnecting.
1575927329: Socket error on client 9?xNtH21\[tfFlZhJau>mtg7Ypj\UeN_B9j0s0@=^RaxC1N^p2gobEWfrO6mL032, disconnecting.
1575927345: mosquitto version 1.5.8 terminating
```

FIGURE A.6: Message Exchanged with QoS=2

Bibliography

[1] R. Arnold, C. Rock, L. Croft, B. L. Gilliam, and D. J. Morgan, “Reply to "therapeutic outcomes of pyogenic vertebral osteomyelitis requiring spinal instrumentation",” *Antimicrobial Agents and Chemotherapy*, vol. 58, no. 11, p. 7022, 2014.

[2] M. A. Feki, F. Kawsar, M. Boussard, and L. Trappeniers, “The Internet of Things: The Next Technological Revolution,” *Computer*, vol. 46, no. 2, pp. 24–25, 2013.

[3] Z. J. Andersen, M. Hvidberg, S. S. Jensen, M. Ketzel, S. Loft, M. Sørensen, A. Tjønneland, K. Overvad, and O. Raaschou-Nielsen, “Chronic obstructive pulmonary disease and long-term exposure to traffic-related air pollution: A cohort study,” *American Journal of Respiratory and Critical Care Medicine*, vol. 183, no. 4, pp. 455–461, 2011.

[4] R. Rushikesh and C. M. R. Sivappagari, “Development of IoT based vehicular pollution monitoring system,” *Proceedings of the 2015 International Conference on Green Computing and Internet of Things, ICGCIoT 2015*, pp. 779–783, 2016.

[5] Y. Wang, J. Li, H. Jing, Q. Zhang, J. Jiang, and P. Biswas, “Laboratory Evaluation and Calibration of Three Low-Cost Particle Sensors for Particulate Matter Measurement,” *Aerosol Science and Technology*, vol. 49, no. 11, pp. 1063–1077, 2015.

[6] A. Arfire, A. Marjovi, and A. Martinoli, “Mitigating Slow Dynamics of Low-Cost Chemical Sensors for Mobile Air Quality Monitoring Sensor Networks,” *Proceeding EWSN ’16 Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*, pp. 159–167, 2016.

[7] I. Lee and K. Lee, “The Internet of Things (IoT): Applications, investments, and challenges for enterprises,” *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.bushor.2015.03.008

[8] H. E. Co, “MG-811 Datasheet.” [Online]. Available: https://sandboxelectronics.com/files/SEN-000007/MG811.pdf

[9] Omron, “B5W-LD0101 Datasheet.” [Online]. Available: https://omronfs.omron.com/en_US/ecb/products/pdf/en_b5w-ld0101-1_2.pdf

[10] T. S. C. LTD, “PPD60PV Datasheet.” [Online]. Available: http://www.snuffle.org/lib/exe/fetch.php?media=uk_shn_ppd60pv-t2_ds.pdf

[11] A. E. C. Ltd, “DHT22 Datasheet.” [Online]. Available: https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf

[12] Alphasense, “CO2 IRC-AT Datasheet.” [Online]. Available: http://www.alphasense.com/WEB1213/wp-content/uploads/2016/11/IRC-AT.pdf

[13] A. P. Almeida, “Using sensors and RaspberryPi for IoT.”

[14] M. Maksimović, V. Vujović, N. Davidović, V. Milošević, and B. Perišić, “Raspberry Pi as Internet of Things hardware : Performances and Constraints,” *Design Issues*, vol. 3, no. JUNE, p. 8, 2014.

[15] J. Geerling, “Raspberry Pi microSD card performance comparison - 2018.” [Online]. Available: https://www.jeffgeerling.com/blogs/jeff-geerling/raspberry-pi-microsd-card

[16] M. Ibrahim, A. Elgamri, S. Babiker, and A. Mohamed, “Internet of things based smart environmental monitoring using the Raspberry-Pi computer,” *2015 5th International Conference on Digital Information Processing and Communications, ICDIPC 2015*, pp. 159–164, 2015.

[17] G. S. N, “Raspberry Pi Based Client-Server Synchronization Using GPRS,” vol. 5, no. 2, pp. 94–98, 2015.

[18] N. M. Kumar, K. Atluri, and S. Palaparthi, “Internet of Things (IoT) in Photovoltaic Systems,” *2018 National Power Engineering Conference, NPEC 2018*, pp. 1–4, 2018.

[19] D. Thangavel, X. Ma, A. Valera, H. X. Tan, and C. K. Y. Tan, “Performance evaluation of MQTT and CoAP via a common middleware,” *IEEE ISSNIP 2014 - 2014 IEEE 9th International Conference on Intelligent Sensors, Sensor Networks and Information Processing, Conference Proceedings*, no. April, pp. 21–24, 2014.

[20] N. De Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali, “Comparison of two lightweight protocols for smartphone-based sensing,” *IEEE SCVT 2013 - Proceedings of 20th IEEE Symposium on Communications and Vehicular Technology in the BeNeLux*, pp. 0–5, 2013.

[21] N. Tantitharanukul, K. Osathanunkul, K. Hantrakul, P. Pramokchon, and P. Khoenkaw, “MQTT-Topics Management System for sharing of Open Data,” *2nd Joint International Conference on Digital Arts, Media and Technology 2017: Digital Economy for Sustainable Growth, ICDAMT 2017*, pp. 62–65, 2017.

[22] K. Grgić, I. Špeh, and I. Hedi, “A web-based IoT solution for monitoring data using MQTT protocol,” *Proceedings of 2016 International Conference on Smart Systems and Technologies, SST 2016*, pp. 249–253, 2016.

[23] WHO, “7 million premature deaths annually linked to air pollution.” [Online]. Available: https://www.who.int/mediacentre/news/releases/2014/air-pollution/en/

[24] D. Hasenfratz, O. Saukh, C. Walser, C. Hueglin, M. Fierz, T. Arn, J. Beutel, and L. Thiele, “Deriving high-resolution urban air pollution maps using mobile sensor nodes,” *Pervasive and Mobile Computing*, vol. 16, no. PB, pp. 268–285, 2015.

[25] W. A. Lahoz, M. Vogt, F. R. Dauge, P. Schneider, A. Bartonova, and N. Castell, “Mapping urban air quality in near real-time using observations from low-cost sensors and model information,” *Environment International*, vol. 106, no. December 2016, pp. 234–247, 2017.

[26] K. Hu, A. Rahman, H. Bhrugubanda, and V. Sivaraman, “HazeEst: Machine Learning Based Metropolitan Air Pollution Estimation from Fixed and Mobile Sensors,” *IEEE Sensors Journal*, vol. 17, no. 11, pp. 3517–3525, 2017.

[27] D. N. Gomes and S. Sim, “Mobile Station for Air Quality Mapping Sensor Network,” Ph.D. dissertation, 2011.

[28] M. Mead, O. Popoola, G. Stewart, P. Landshoff, M. Calleja, M. Hayes, J. Baldovi, M. McLeod, T. Hodgson, J. Dicks, A. Lewis, J. Cohen, R. Baron, J. Saffell, and R. Jones, “The use of electrochemical sensors for monitoring urban air quality in low-cost, high-density networks,” *Atmospheric Environment*, vol. 70, pp. 186–203, may 2013. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S1352231012011284

[29] D. B. Ramsay, “LearnAir : toward Intelligent , Personal Air Quality Monitoring,” Ph.D. dissertation, 2016.

[30] S. M. Biondi, V. Catania, S. Monteleone, and C. Polito, “Bus as a sensor: A mobile sensor nodes network for the air quality monitoring,” *International Conference on Wireless and Mobile Computing, Networking and Communications*, vol. 2017-Octob, pp. 272–277, 2017.

[31] Adafruit, “Adafruit_Python_BME280.” [Online]. Available: https://github.com/adafruit/Adafruit{_}Python{_}BME280

[32] A. ADS, “Adafruit Python ADS1x15.” [Online]. Available: https://github.com/adafruit/Adafruit{_}Python{_}ADS1x15

[33] T. Instruments, “Analog-to-Digital Converter with Internal Reference ADS1013,” 2009.

[34] Adafruit, “GPS FeatherWing.” [Online]. Available: https://www.adafruit.com/product/3133

[35] A. GPS, “Adafruit-Ultimate-GPS-FeatherWing.” [Online]. Available: https://github.com/adafruit/Adafruit-Ultimate-GPS-FeatherWing-PCB

[36] PTISP, "PTISP." [Online]. Available: https://ptisp.pt/

[37] Amen, "Amen Hosting Linux." [Online]. Available: https://www.amen.pt/hosting/linux/

[38] Webtuga, "Alojamento Linux."

[39] M. Stonebraker and G. Kemnitz, *The POSTGRES next generation database management system*, 1991, vol. 34, no. 10.

[40] 2ndQuadrant, "PostgreSQL vs MySQL." [Online]. Available: https://www.2ndquadrant.com/en/postgresql/postgresql-vs-mysql/

[41] Apache, "Apache Friends." [Online]. Available: https://www.apachefriends.org/index.html

[42] EDUCBA, "Java vs Python." [Online]. Available: https://www.educba.com/java-vs-python/

[43] C. Götz and D. Obermaier, "HiveMQ." [Online]. Available: https://www.hivemq.com/blog/are-your-mqtt-applications-resilient-enough/

[44] E. Lindén, J. Wallgren, and P. Jonsson, "A latency comparison of IoT protocols in MES," p. 47, 2017. [Online]. Available: http://www.ep.liu.se/.