# Politecnico di Torino

Master's Degree in Computer Engineering


Master's Degree Thesis

# Reliable and efficient solutions for a participative air pollution monitoring system

Supervisors

**Filippo Gandino**

**Edoardo Giusto**

**Pietro Chiavassa**

Candidate

**Alessandro Ricciuto**

**Academic Year 2021 - 2022**

# Contents

# Chapter 1

# Introduction

## 1.1 Problem of air pollution

Air pollution is one of the leading causes of mortality worldwide.

The World Health Organization (WHO) identifies air pollution as one of the major causes of the increasing hospital admissions for cardiovascular and respiratory diseases and for mortality in many European cities and other continents (as shown in Figure 1.1) [1].

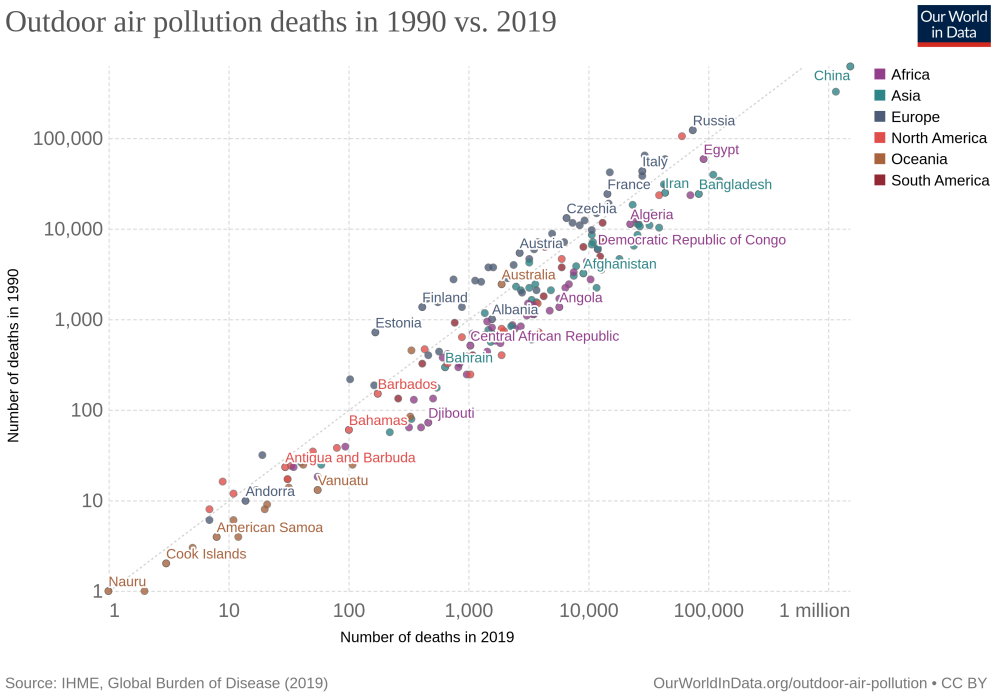


**Figure 1.1:** Number of deaths from outdoor air pollution, 1990 vs. 2019 [2]

While some causes of pollution are strictly correlated to natural events –

sudden changes in temperature, seasonal changes, or regular cycles – others are the effects of human activities [3]. One of the main pollution factors is given by particulate matter, microscopic particles, mostly generated by fossil-fuel combustion and waste incineration. According to their diameter size, they are classified into two categories [4]:

- **$PM_{10}$**, inhalable coarse particles with a diameter of 10 micrometers (µm) or less;

- **$PM_{2.5}$**, fine particles with a diameter of 2.5 µm or less;

Alongside the long-term problems that inhaling these particles can cause, like stroke, heart disease, lung cancer, and so on, the smallest particles are the most dangerous: they can travel deeply into the respiratory tract, reaching the lungs and causing short-term health effects such as eye, nose, throat and lung irritation, coughing and shortness of breath [5].

## 1.2 Air quality monitoring

Many countries in the world, in particular the U.S. and Europe, have developed environmental monitoring systems and taken corrective actions to keep air quality under control. That was made especially through the definition of standards, legal directives and regulations to which every region and agglomeration has to stick.

The PM monitoring in the EU is described by the Directive 2008/50/CE [6] and is performed using high-precision techniques based on $\beta$-attenuation monitoring and gravimetric detection. The sensors are positioned on fixed stations and the spatial coverage is limited due to the high cost of the instrumentation, which is between 50K - 100K US $ [7].

Considering the monitoring network of the Metropolitan Area of Turin, Italy, managed by the environmental agency ARPA (Agenzia Regionale Protezione Ambientale), Figure 1.2 shows how the stations are positioned.

The concentration of particulate matter in the air can be widely variable and requires a monitoring system that detects changes in the shortest time possible. It is closely linked to the climatic conditions of a specific area of the territory and it is the reason why a dense network of stations is needed to have a better representation of the environmental situation. On windy days or in case of wildfires, for example, the measures of particulate concentration vary in less than one hour and could be dangerous to know it only hours later, or even the day after. In some cases, making people aware of the situation as soon as possible could be a vital issue.

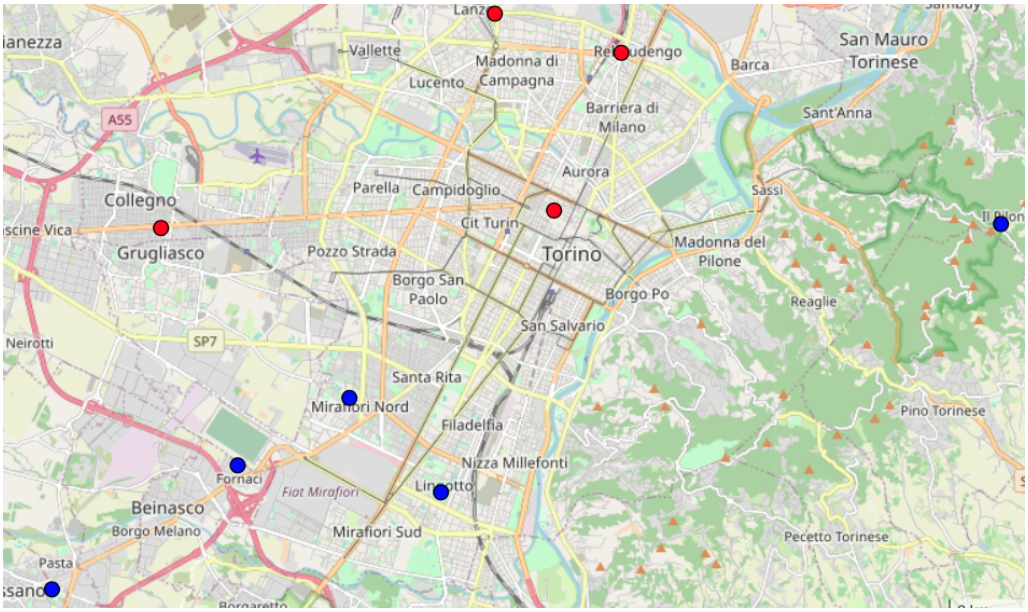


**Figure 1.2:** Network of the ARPA monitoring stations in the Metropolitan Area of Turin [8]. Red markers are high-traffic locations, blue markers are low-traffic locations.

To interpret data in a way that satisfies this need, many algorithms were proposed. One of the most used, until 2013, was the Conroy method. Its result produces an Air Quality Index (AQI) based on the last 24 hours of measurements. However, it was shown to be too slow to respond to rapid changes in air conditions. Based on this, the NowCast algorithm, known also as the Reff method, was proposed. It relies on the hourly averages from the prior 12 hours, unlike Conroy's one which is based on the latest 24. Moreover, when air quality variation is considerable, the AQI is computed on an average of the 3 most recent hours [9].

## 1.3 Goal of the project

This project aims to explore and develop an innovative possibility represented by a participative environmental monitoring system that implies the involvement of the citizens in the transmission and collection of environmental conditions data.

### 1.3.1 Participative approach

In the context of this thesis, citizens will not only be able to keep constantly updated on the air quality of the area they are in, but they will indirectly

participate in the monitoring process. Since the users are an active part of this system, sharing their hardware and software tools for collecting data, the monitoring system will be referred to with the term of *participative*. The participation consists of receiving the measurements from nearby monitoring devices and sending them in turn to a remote server for storing and processing purposes. Measurements are received on citizens' smartphones that can be associated to gateways working in a telecommunications network, allowing the data flow from a source to a destination. Since the concept of this project is based on creating a wider monitoring network through the use of low-cost sensors, the participative approach can reduce the overall cost of the system, increasing also portability and scalability: data trasmission to the server takes place via the internet connection provided by the users, avoiding of equipping each sensor with it.

A similar approach was used for the implementation of an European project, PULSE (Participatory Urban Living for Sustainable Environments) [10]. Through the use of gamification, a research group, composed of major European universities, developed a mobile application where users, represented by avatars, have to gain points to be rewarded. Rewards are assigned as a recompense for the achievement of an objective through a series of actions proposed, to promote or to follow healthy and green habits. Data collection comes from heterogeneous sources and it is mostly in charge of the user through the use of wearable devices [11].

### 1.3.2 Status of the project

This project is intended as a continuation of a previous work [12]. In fact, at the beginning of this thesis work, some architectural and technological decisions had been already taken.

The IoT device, which is in charge to collect and store environmental data, is a microcontroller that mounts a set of sensors for measuring $PM_{10}$ and $PM_{2.5}$ concentration, relative humidity, temperature, air pressure and GPS position. Since the main objective of the research group is the portability and scalability of the system, PM sensors are much smaller and cheaper than the ones used by environmental agencies. Moreover, the research group wants to experiment with a solution in which the system is not connected to the Internet due to the considerable total cost of the service for all the sensors. Following this approach brings advantages and disadvantages:

- Main advantages:
  - portability of the system. It can be easily moved from one location to another.

  - given the low-cost of the sensors, it is possible to create a larger monitoring network on the territory.
  - local governments can take corrective actions to targeted areas.
  - since no mobile network subscription is needed for the participative nature of the system, the cost of the implementation is reduced.

- Main disadvantages:

  - due to the small dimension of the sensors, the precision of the measurements is affected and break events could happen more frequently.
  - since the transmission of the data from the microcontroller to the remote server takes place through users, there is no guarantee that measures are continuously transmitted.

The participative system was implemented through the development of a mobile application whose purpose is to act as a gateway between the monitoring system and the remote server. The UI of the app was completely implemented while the transmission of the data, using Bluetooth Low Energy technology, was not complete and it still had some aspects to improve in performance and reliability.

The remote web server, developed by the research group, was able to manage and store the measurements through its APIs. However, the research group decided to improve and renovate it to adjust it for the experimentation phase.

### 1.3.3 Implementation

The contributions given by this thesis to the project are the following:

- redesign the data management on the microcontroller, implementing a system of queues able to make the operations of load and store more efficiently;
- implements the communication protocol using Bluetooth Low Energy (BLE) to transmit data from the microcontroller to the mobile application;
- implements a BLE client agent on the mobile application able to automatically connect to nearby microcontrollers;
- implement and redesign some APIs on the web server;

### 1.3.4 Thesis organization

The document is structured in 7 chapters, including this introduction. The second chapter is meant to provide a general overview of the architecture of the system, describing all the components, how they interact and the technologies used for this project. Chapter 3 is focused on the IoT device to which the microcontroller belongs. Chapter 4 is about mobile application development, the framework used and its interaction with the board and the server. Chapter 5 is meant to provide an evaluation of the system by presenting the relevant data. The last chapter, the seventh, contains some final considerations and introduces possible future works on the project.

# Chapter 2

# Architecture of the system

The idea of this project stems from the need to make air quality monitoring more effective. In Italy, the official monitoring systems are managed by public entities and are represented by large, fixed and expensive stations. Since they cannot be placed in the city center for logistic and economic reasons, they produce data that are not suitable for taking localized countermeasures.

The solution proposed by this project is represented by the use of tiny low-cost monitoring devices, spread around the city. They are more portable and can sample environmental conditions data every second, producing a considerable amount of data. Moreover, since connecting each of these devices to the Internet would result in a general increase in costs, the system was designed to follow a participative approach. For these reasons, to properly handle and collect the data, the system architecture, shown in Figure 2.1, was adopted. The main components represented in the figure are the IoT device which is in charge of collecting and transferring the data, citizens' smartphones which act as gateways to the server and the server which is composed of a web application, a data analysis module and a database. Alternative technologies, which will be described later in the chapter, are still under study and development, such as the use LoRa technology, the MQTT protocol and a dashboard to configure the system and visualize the data.

The purpose of the next sections in this chapter is to describe in detail the technological components of this system and how they interface with each other.

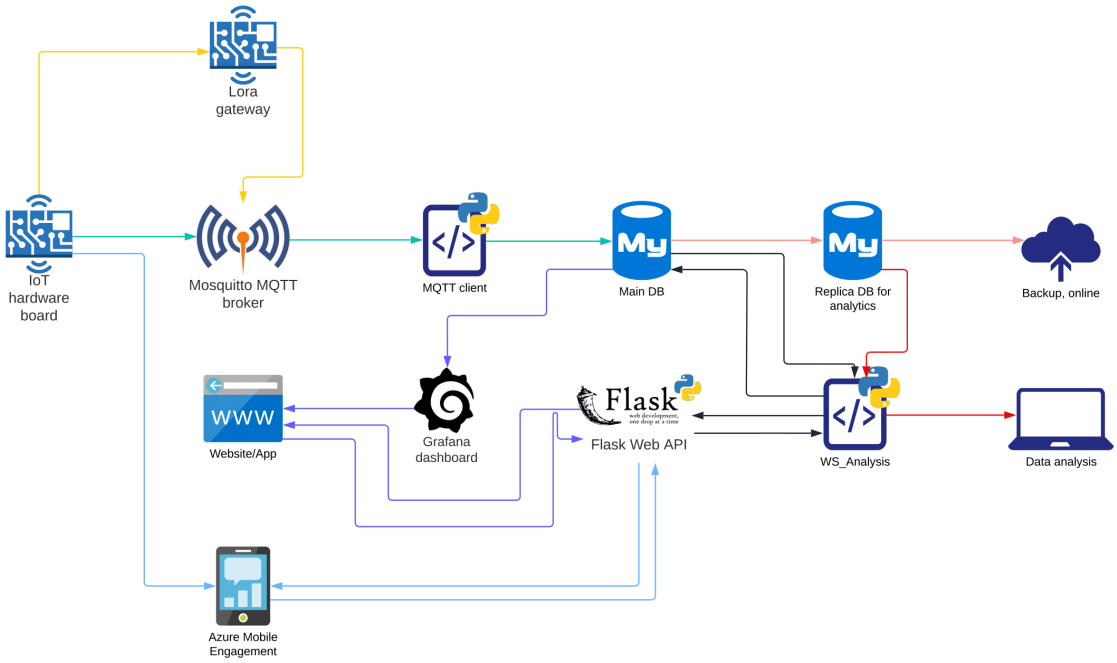


**Figure 2.1:** System architecture schema

## 2.1 Components

This section aims to describe the technologies that compose the parts of the system. The system architecture of this project is placed in the context of the Internet of Things and requires advanced connectivity of devices, systems and services that interact using a variety of protocols, domains and applications.

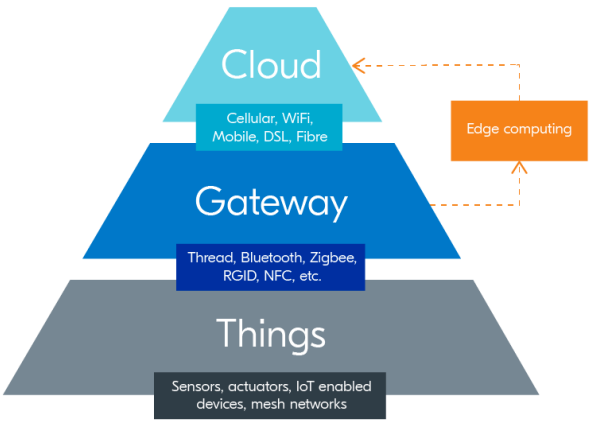


**Figure 2.2:** IoT system architecture [13]

The Internet of Things is often associated with the concept of distributed systems and, even in the context of this project, some similarities could be found. In contrast with the cloud infrastructure characterizing the distributed systems, here the system is based on a centralized remote server with its database. However, the interconnection of the physical items geographically far from each other makes the system decentralized.

The edge nodes of the system are represented by the monitoring devices

while the users' smartphones are the mobile agents that act as the gateways responsible for the data transmission to the server.

### 2.1.1 IoT device

Each monitoring device is a platform composed of a few boards and a set of sensors. Originally, the core of the prototype was based on a Raspberry Pi, running a Linux operating system capable of collecting the measurements and handling the synchronization tasks. However, due to the high power consumption which can not be modified through the options of the device and the complexity of configuration given the presence of an operating system, the core of the board was replaced by a microcontroller with the following components:

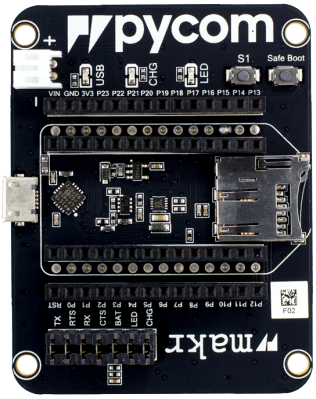


**Figure 2.3:** Pycom Expansion Board 3 [14]

- an expansion board, the Pycom Expansion Board 3 showed in Figure 2.3. It is both USB and LiPo battery-powered, with options for two different charging currents (100mA and 450mA). It is equipped with a MicroSD card slot for memory expansion, several LEDs showing the state of the system and lots of jumpers to enable and disable features. To make development easier, Pycom has developed a plugin, called Pymakr, available for the popular code editors Visual Studio Code and Atom. It facilitates developers in flushing the firmware and debugging the code.

- a development board, the Pycom FiPy Development Board showed in Figure 2.4. It embeds an Expressif ESP32 microcontroller unit (MCU) as CPU which implements different energy modes to reduce power consumption. The MCU is a dual processor and the main processor is entirely free to run the user application. It embeds five communication modules, including WiFi, Bluetooth (classic and low energy), cellular LTE-M (CAT-Ml and NBloT), LoRa and Sigfox. The firmware is programmable in MicroPython, a subset of Python programming language, which permits direct control over the MCU and its peripherals.


**Figure 2.4:** Pycom FiPy Development Board [15]

The hardware architecture is also equipped with the peripherals needed for data acquisition. As shown in the graphical representation of the monitoring device in Figure 2.5, the peripherals are the following:

- 4 PM sensors Honeywell® HPMA115S0-XXX. The quantity of sensors is aimed at introducing data redundancy, allowing the detection of strange events on the single device and avoiding the occurrence of failures;
- 1 DHT22 as temperature and relative humidity sensor;
- 1 Bosch BME280 as pressure sensor;
- 1 PA1010D module as GPS sensor;
- 1 DS3231, a Real Time Clock (RTC) module for the operating system to retrieve the correct time after a sudden power loss;

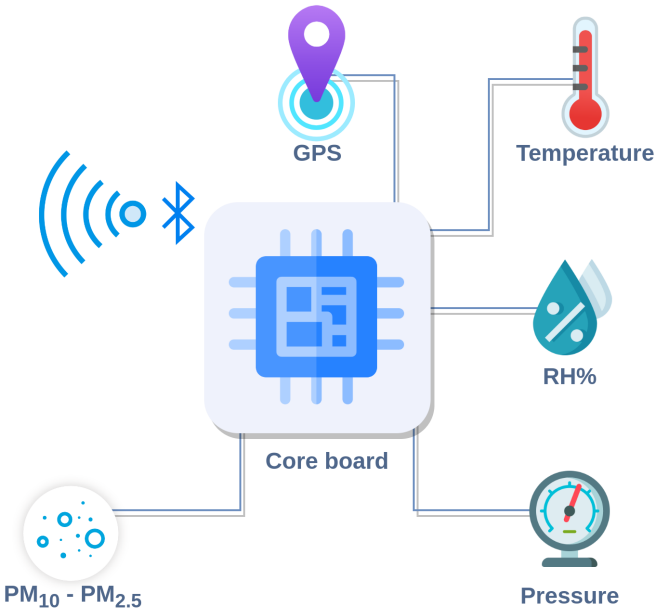


**Figure 2.5:** Monitoring device. Data coming from the sensors are first stored in the MicroSD memory of the board and then transmitted through Bluetooth Low Energy.

### 2.1.2 Mobile Application

The participative way of the project is represented by a mobile application, running on citizens' smartphones. It has been developed using Flutter, an open-source cross-platform framework created by Google, which allows building native interfaces for both of the most popular mobile operating systems,

Android and iOS. This architectural choice was taken to make the application available for most of the smartphones in circulation, without having to develop a native application for each operating system and facilitating the maintenance of the code.

In the context of this project, having a native app is essential to obtain the expected results because using a web-based UI framework to build a hybrid app would lead to performance issues. The reason is the fact that the app of this project heavily uses the hardware components of the device and Flutter, using its rendering engine, eliminates the need for a communication bridge with the operating system.

However, to be able to use some of the built-in platforms on the smartphone, such as Bluetooth Low Energy and GPS location, some restrictions apply to the versions of the operating systems:

- Android 4.4 and later;
- iOS 5 and later;

### 2.1.3 Server

The system architecture is based on a decentralized network of devices that refer to a central remote web server. The server integrates the Flask framework which allows running a web application, implemented using the Python programming language, based on the REST paradigm. Flask was created with the idea of being easy to use, following the principle of minimalism with its lightweight and modular design, but the developer can build a complex web application through the integration of plugins and extensions.

Data are stored in a MySQL database that, for this experimentation phase, is located on the same server of the web application. However, the research group has already faced the performance matter of having a fast response to SQL queries and decided to use a dedicated server, a Dell computer with 20 cores, 384 GB of internal memory, and some TB of hard disk [16]. Moreover, the decision was taken considering also the need to create different layers between the business logic and the data storage, given the very fast growth of the database. In the end, the project also includes a replica of the data which could be useful for analytics and backup purposes.

Since the project is still in the experimentation phase, the central configuration chosen for the server, known also as *star* network topology (figure 2.6), was chosen for its quickness and simplicity in the implementation. However, a single master server represents a single point of failure on the network and cannot scale, limiting the computational power and the requests that can be processed concurrently. Instead, implementing a distributed system, known

also as *mesh* network topology (figure 2.7), could result in a fault-tolerant, highly scalable, faster and secure system. The problem would be related to the orchestration and management of the whole system, considering also the synchronization issues that could arise.

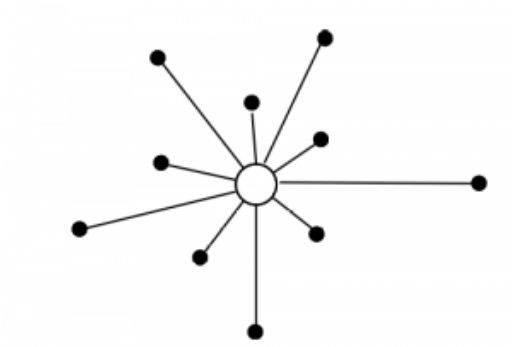

**Figure 2.6:** Centralized network management [17]

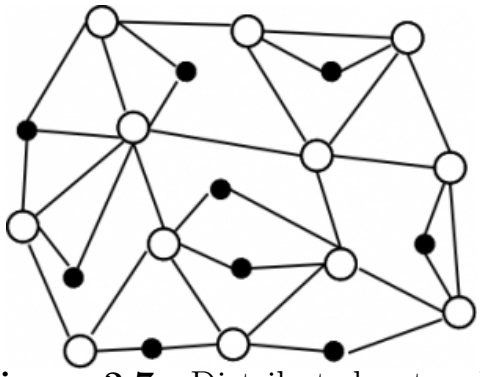

**Figure 2.7:** Distributed network management [17]

## 2.2 Data communication

The main communication channel, on which this work is based, is the one that links the monitoring device to the mobile application. Considering that the objective of the project is to build a dense low-cost network of monitoring devices spread around the territory, each monitoring device could not be equipped with a mobile Internet connection due to the considerable whole cost that it would imply. Therefore, this work aims to analyze the use of the Bluetooth Low Energy technology in data transmission which, as explained in section 3.2, is becoming the preferred and one of the most compatible options for the Internet of Things. The main reason is due to the energy efficiency that supports the connectivity of the IoT devices for longer periods than other technologies (e.g. WiFi or classic Bluetooth), especially when the device is battery-powered.

Bluetooth Low Energy has a range of about 10 meters, limiting its use to confined and small areas. However, it supports multiple topologies options, depending on what the system architecture requires:

- Point-to-Point, used for establishing one-to-one communications. It is optimized for data transfer and is well suited for connected device products, such as fitness trackers, health monitors, PC peripherals and accessories [18];
- Broadcast, used for establishing one-to-many device communications. It is optimized for localized information sharing and is ideal for location

services such as retail point-of-interest information, indoor navigation and wayfinding, as well as the item and asset tracking [18];

- Mesh Networking, used for establishing many-to-many device communications. It allows the creation of large-scale device networks and is ideally suited for control, monitoring, and automation systems where the devices need to reliably and securely communicate with others [18];

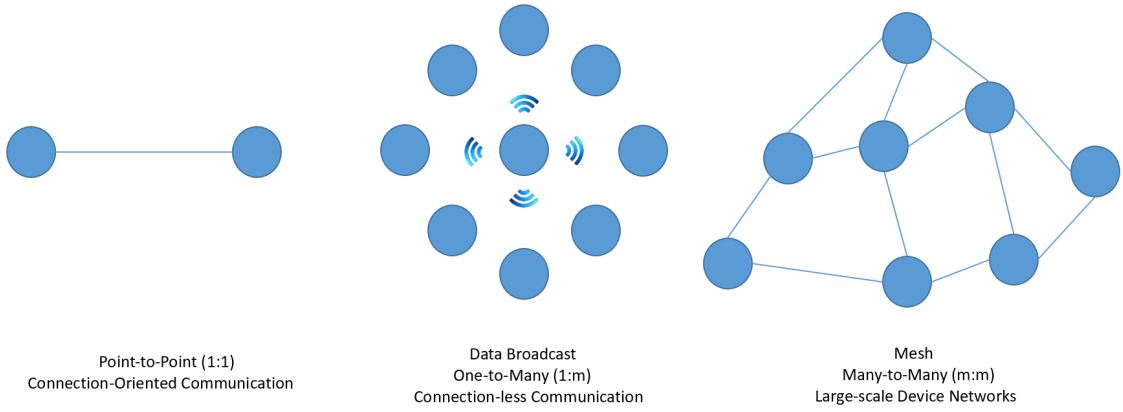


**Figure 2.8:** Bluetooth Low Energy topologies [19]

## 2.3 Alternative technologies

Other solutions to implement the data transmission across the system can be considered.

One of them is represented by WiFi technology. It transmits at frequencies of 2.4 GHz or 5 GHz which are much higher than the frequencies used for cellular transmission, allowing the signals to carry more data. However, all forms of wireless communication represent a trade-off between power consumption, range, and bandwidth. So in exchange for high data rates, WiFi consumes a lot of power and does not have a lot of range [20]. Moreover, to send data from a monitoring device to the remote server, two options are available. The first consists of the presence of a WiFi access point near each monitoring device, which would lead to a considerable increase in costs. Furthermore, even if the Internet connection were provided by local authorities, the organizational complexity would increase and the flexibility in the positioning of the stations would be reduced as in some places there is no network availability. With the second option, each IoT device becomes a WiFi access point, to which gateways will have to connect to receive data. However, previous work on this

project has deeply analyzed the solution discarding it for the long time needed to allocate the socket and build the connection between the devices [12].

Another solution is represented by putting in direct communication each monitoring device with the remote server using LoRa, a low-frequency modulation technology. It is commonly embedded in IoT sensors and devices to achieve long-range communication with low battery consumption. The research group has detected that the communication range can reach 600 m and each packet, for which a maximum payload is 51 bytes, could be sent at an interval of 5 minutes from each other [21]. With the LoRa technology, the peripheral devices have to communicate with a LoRa gateway which is in charge of receiving the signals, eventually converting and sending them to the cloud through the use of other technologies, such as WiFi or optical fiber. The infrastructure of an IoT system based on LoRa could exploit the MQTT protocol to send messages from the LoRa gateways to the network. It is a lightweight open IoT messaging transport protocol that is based on a publish/subscribe pattern of communication. Meaning that, instead of communicating with a server, client devices and applications publish and subscribe to topics handled by a centralized MQTT broker. MQTT can simplify the transmission of LoRa packets since they have to traverse multiple technology stacks and infrastructures [22] but it is too heavy for LoRa technology and can be used by the peripheral devices only over WiFi or from the LoRa gateway by connecting to an MQTT broker. In the context of this project, given the limited bandwidth of the LoRa technology, it could be exploited once in a while to transmit the status of the board while it is not applicable for the transmission of the measurements.

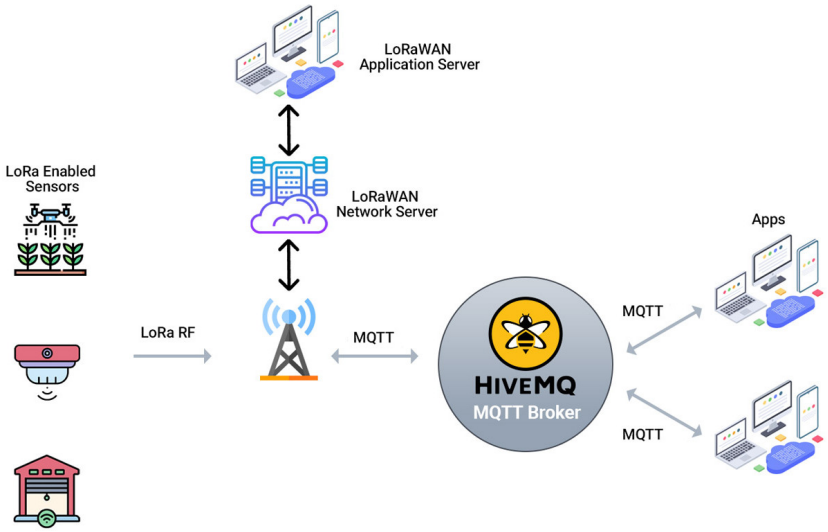


**Figure 2.9:** MQTT Integration with LoRaWan Gateway [22]

# Chapter 3

# IoT Device

This project is placed in the context of smart cities and smart environments that refer to a growing technological field, the Internet of Things. It is intended as the process of connecting to the Internet everyday physical objects, from the most familiar objects used in the home, such as light bulbs, to resources in the health sector, including medical and wearable devices.

IoT stands for any system of physical devices that receive and transfer data mostly over wireless networks, with limited manual intervention. IoT devices are inserted into the real world to monitor and interact with the environment. They can be divided into two main categories of technologies: switches, responsible for sending commands to other objects, and sensors, which acquire data to send them elsewhere [23].

One of the objectives of the IoT is certainly aimed at improving the quality of life of the people and of the environment where they live, through the analysis of the large amount of data collected by these devices.

In the context of this project, air quality monitoring is achieved through the use of an IoT device composed of low-cost PM sensors mounted on a microcontroller that transmits data through Bluetooth Low Energy technology. The core of the device is represented by a Pycom expansion board and a Pycom development board which is programmable with the MicroPython programming language, an efficient implementation of a subset of Python 3.4.

This chapter aims to describe the main functionalities of the device, focusing on those implemented during this work, and the principles of the Bluetooth Low Energy technology used to implement the data exchange protocol, describing also some implementation issues and some future improvements.

## 3.1 Functionalities

The device is in charge of monitoring the surrounding environment. Its primary functions are therefore to collect the data produced by the detection of the sensors with an adequate frequency and to store them in memory. Storage management has to take into account that the memory is limited and an efficient organization of the data should be done. The solution proposed in this work tries to reorganize the data into binary files to reduce memory allocation. Moreover, to avoid wasting CPU clock cycles on I/O operations, data are processed through a system of buffers and queues located in the RAM. Data transmission through Bluetooth Low Energy is another important functionality of the device, trying to make it as efficient as possible but also taking into account a certain level of reliability.

Pycom development board allows to implement the functionalities through many basic libraries inherited from Python, such as the *math* and the *crypto* libraries, but also modules that allow us to communicate with the hardware components of the board, such as the Bluetooth radio or the switches of the LEDs. The deployment of the firmware on the microcontroller is quite simple since Pycom has developed a plugin interface, called Pymakr which is available for Visual Studio Code and Atom IDEs, and allows the developer to upload, download and debug the code on the board.

### 3.1.1 Data sampling

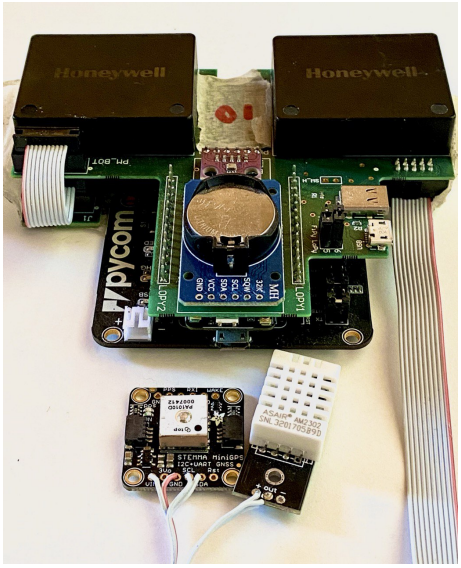


**Figure 3.1:** Picture of the IoT device composition

The data acquisition is made through the sensors mounted on the device (figure 3.1). In addition to PM concentration measurements, other environmental data are needed to detect air quality. In fact, the device is equipped with the following sensing peripherals: 4 PM sensors, each of them able to measure both $PM_{10}$ and $PM_{2.5}$ concentration, attached to the Fipy board UART, 1 sensor connected via one-wire protocol to measure the temperature and the relative humidity, 1 pressure sensor and 1 GPS sensor, both connected via the I2C protocol.

In the boot phase of the device, each sensor connection is established via software through the configuration of the pins described by the configuration file located on the board. Even data sampling is initialized in this phase and it is made through the definition of periodic timers, one for each sensor. Therefore, each sensor has its sampling frequency and when a timer expires, a routine will be activated to retrieve the measurement value produced by the sensor (Figure 3.2). The routine is also in charge of writing the measure in a buffer stored in RAM that, as described later, will be flushed into a file of the memory.

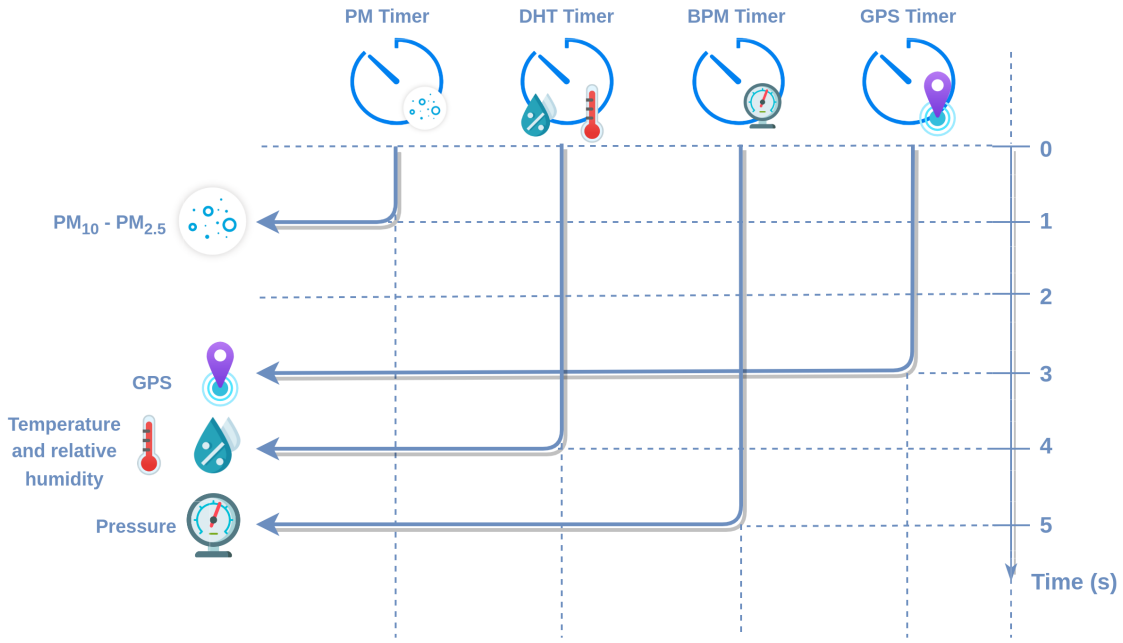


**Figure 3.2:** Data sampling with timers. When a timer expires, a routine retrieves the measurement value produced by the sensor.

### 3.1.2 Data storing

The device is equipped with a RAM of 4MB and built-in flash memory of 8MB on the FiPy board which is inadequate for storing the sensing data and is used to store the firmware code. However, the expansion board has a MicroSD card slot that is accessed via SPI protocol. In the context of this project, a MicroSD of 32GB has been chosen to store the data and the file system of the device.

One of the objectives of this work was to improve storage management in order to reduce the allocated memory and reduce the size of the files containing the measurements produced by the device.

Until now, data were archived in the file system using text files formatted

according to CSV format. Each file contained the measurements of an entire day of detection, bringing its size to around 18MB. However, this file size was incompatible with the operations that the microcontroller can do, both regarding the I/O operations and the transmission over a wireless network. Since Pycom does not provide a module to compress files, what has been decided to do is to change data archiving into binary files. Furthermore, the temporal window of the measurements contained in each file has been reduced from one day to one minute and the main reason, as will be explained in paragraph 3.3, is due to the high transmission time of one-day files, which is incompatible with the time in which a user will be near to a station. These approaches reduced the files dimension by about 65%, improving the I/O operations with the memory, saving computational time and reducing also the data transfer time through BLE.

Each stored measure contains 3 main pieces of information: the acquisition timestamp of the measure, the sensor identifier which collected the data and the value of the measure. Therefore, using the CSV format, the file was structured as in the following example:

```
...

2022-09-19 12:30:05,49,12
2022-09-19 12:30:06,129,35.6
2022-09-19 12:30:07,15,25.8564
2022-09-19 12:30:08,46,9956.45

...
```

The conversion of the data into the relative binary codification has been done through the *struct* library provided by MicroPython. Each piece of information has been converted in bytes following the big-endian representation, so from the most significant byte to the least one. According to their value size, the applied conversion has been done as follows:

- the timestamp has been converted from the string representation of the ISO 8601 standard to the 10-digit representation of the UNIX format. Then, it has been converted to the binary representation of a signed integer, which size is 4 bytes (e.g. 2022-09-19 12:30:05 $\rightarrow$ 1663583405);

- the sensor identifier is an integer and its size could be variable. In order to be as flexible as possible, the binary representation has been chosen to be on 4 bytes integer;

- the value of the measure could be an integer, as in the case of PM measurement, or a float, as in all other cases. Since having multiple sizes

would result in adding information to correctly interpret the data, all the measurement values will be treated as a float represented on 4 bytes;

The resulting conversion of the data in their binary representation is shown in the example in Figure 3.3, where each byte is in its hexadecimal representation in order to be more readable.

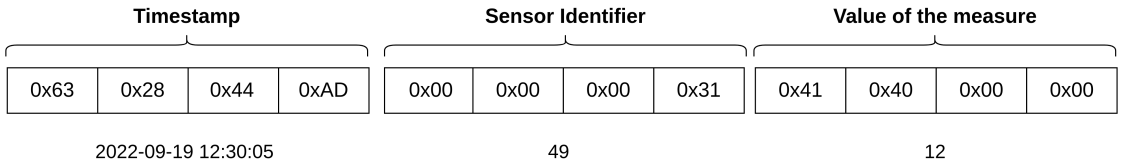


**Figure 3.3:** Example of the binary representation of the data

### 3.1.3 Data processing

Another aspect that this work wants to address is the security associated with the integrity of the data transmitted as they may have been altered during the transmission by unknown actors. Therefore, giving the possibility to the recipient of verifying the origin of the data is important to guarantee secure communication.

A solution proposed by this work is the introduction of the digital signature for the measurement files. A digital signature assures that the person generating the message is who they claim to be and that the content of the message is not altered by anyone during the transmission. It makes use of asymmetric cryptography given by a private and public key. They are used to encrypt and decrypt messages on the basis that every public key matches only one private key. In order to create a digital signature, the sender has to digitally sign the message that wants to send, encoding the message with a hashing algorithm, such as SHA256 or MD5, and then encrypt it with his private key using the RSA algorithm. While the private key is only known by the sender, the public key is accessed by everyone and it is the way through which the recipient validates the signature. The signature validation is done with the reverse approach: the recipient computes the hash of the message, decrypts the digital signature using the sender's public key and finally compares the equality of the 2 hashes.

On the monitoring device, the digital signature has been implemented through the *ucypto* library, where the function *generate_rsa_signature* accepts the string message to encode it with the SHA256 algorithm and the private key in order to encrypt it. Since the message, in this case, is a *bytearray*, containing the measurements, it had to be turned into a string through a Base64 encoding.

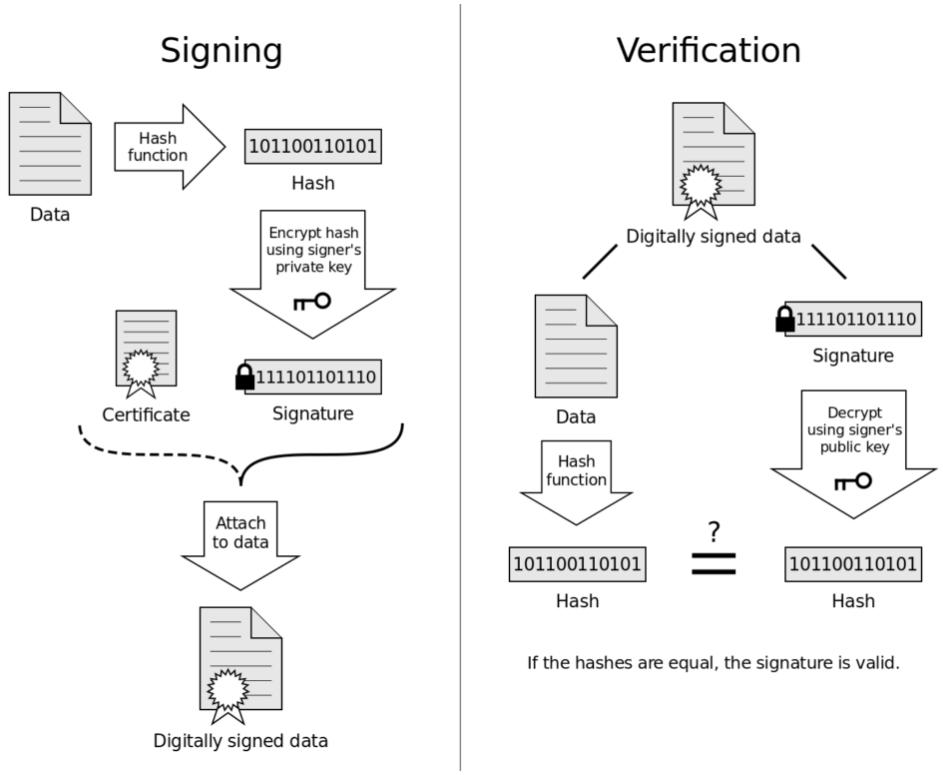


**Figure 3.4:** Digital signature process [24]

However, creating a digital signature is an expensive operation and for every file, it takes about 2 seconds of computations to be performed, during which time the data coming from the sensors is lost. This operation, together with the I/O operations, could compromise data transmission. Since the transmission phase is the one with the highest priority because it could be done only when a smartphone is nearby, what is wanted to avoid is wasting clock cycles in doing expensive operations. In order to meet this need, data are processed in a dedicated module placed between the business logic and the file system. It handles a buffer queue system that aims to simplify and solve a few synchronization and performance issues, giving a priority to each operation (shown in Figure 3.5). Since continuously accessing the SD memory slows down the system, the module contains a series of queues where a copy of some measurements and signatures files located on the file system are stored in buffers in RAM.

It is composed of the following elements:

- a queue of the latest created files, both measurement and signature files, allowing the files to be immediately ready to be sent when a smartphone connects to the device, instead of searching and reading them from the memory;

- a queue of the latest created measurement files which are ready to be signed. Since the operation is computationally expensive, it is handled through this priority queue that is processed one minute at a time only if the device is not already involved in the transmission process;
- a queue of the files already transmitted but waiting for the confirmation of receipt, as explained in paragraph 3.3;
- a single buffer containing the data that refer to the temporal window of the current minute. When the minute will change, the data will be flushed in a file in the file system and it is moved to the top of the queue of the files to be signed. Having this buffer is useful in order to detach the logic of the sensing module from the file system management.

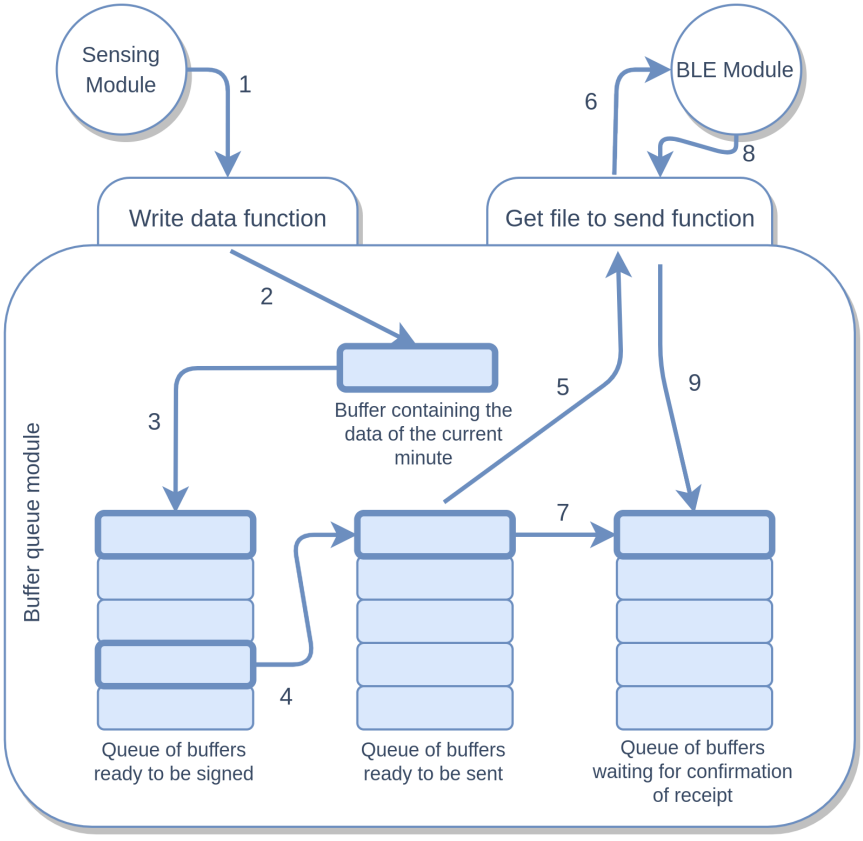


**Figure 3.5:** Buffer queue system

### 3.1.4 Data transmission

In the context of this project, the monitoring device is also in charge of transferring the data that have been stored in its memory to nearby smartphones.

The transmission is made through the Bluetooth Low Energy technology that is based, as explained in section 3.2, on the *GATT* and *ATT* protocol.

The IoT device acts in the role of the *GATT Server*, accepting requests from the *GATT Clients*, represented by the users' smartphones. The connection is based on a Point-to-Point topology and only one connection at a time can be established.

The BLE module, which has been implemented for the data transmission on the board, is contained in the *lib/ble.py* file and it is initialized during the boot phase of the device. The initialization includes 4 main steps:

- reading the configurations from the configuration file. They include the identifiers that the board has to use in order to be recognized by the gateways. Since the gateways will retrieve the identifiers of the BLE service from the server, having the configuration file aligned with the information on the database is essential to establish a connection between the devices;
- instantiating the *GATT service* that will be advertised by the device in order to allow the connection of the smartphones. During this phase, even the *MTU*, the Maximum Transmission Unit represented by the length of the ATT packets, will be defined and fixed to 185 bytes;
- instantiating the 2 communication channels, called *GATT Characteristics*, on which the transmission will take place. One of them will be used for the measurement and signature files transmission and it is configured in order to notify the client of value changes while the other, configured in order to be written by the client, will act as a control channel;
- starting the advertising of the service in order to be discoverable by nearby smartphones and to allow new connections.

The following code snipped shows the BLE initialization:

```
1  class BLE:
2
3      ...
4
5      def __init__(self, bufferHandler):
6          self.bleSettings = config.readJSON('bleConfig.json')
7          self.mtu = 185
8          self.bluetooth = Bluetooth(mtu=self.mtu, secure_connections=False)
9          self.service = self.bluetooth.service(
10             uuid=uuid2bytes(self.bleSettings["ServiceUUID"]),
11             isprimary=True,
```

```
            nbr_chars=2,
            start=True
        )
        self.create_characteristics()
        ...

    def create_characteristics(self):
        self.measure_characteristic = self.service.characteristic(
            uuid=uuid2bytes(self.bleSettings["MeasureCharacteristicUUID"]),
            properties=Bluetooth.PROP_READ | Bluetooth.PROP_NOTIFY
        )
        self.ack_characteristic = self.service.characteristic(
            uuid=uuid2bytes(self.bleSettings["AckCharacteristicUUID"]),
            properties=Bluetooth.PROP_WRITE | Bluetooth.PROP_READ
        )
        ...

    def start_advertisement(self):
        self.bluetooth.set_advertisement(
            name=self.bleSettings["Name"],
            service_uuid=uuid2bytes(self.bleSettings["ServiceUUID"])
        )
        self.bluetooth.advertise(True)
```

The transmission of the files has to take into account that the channel size is regulated by the *MTU* and, as specified in section 4.2.1, its value has been set to 185 bytes for the limitations imposed by the mobile operating systems. Since a file cannot be sent in a single transmission, it must be divided into chunks. The size for the single chunk has been set to 182 bytes because 3 bytes of the ATT packet consists of the *ATT header* and the *ATT handle*, which are respectively 1 and 2 bytes long.

## 3.2 Bluetooth Low Energy

It is estimated that in 2026 the devices equipped with Bluetooth will reach the quota of 7 billion [25]. One of the reasons is also due to the introduction of Bluetooth Low Energy (or BLE), also called Smart Bluetooth, in the Bluetooth 4.0 specification, as an alternative to Classic Bluetooth. Like its predecessor, BLE technology uses wireless technology based on a radio frequency, in the free band of 2.4 GHz in order to connect nearby devices.

The main difference with Classic Bluetooth, as can be easily deduced from the name, is the reduced energy consumption. In fact, for BLE, the bit rate is 1 Mbit/s (with an option of 2 Mbit/s in Bluetooth 5) and the maximum transmission power is 10 mW (100 mW in Bluetooth 5). This means that the power used is less than half of what it used to be.

Bluetooth Low Energy was first unveiled in 2004, following a Nokia research project, and it was released for the first time with Bluetooth 4.0. The first smartphone equipped with Bluetooth 4.0 was the iPhone 4S, in October 2011, which was followed by many others. Today, all new smartphones are equipped with this Bluetooth version or higher.

BLE is one of the main technologies that make the Internet of Things possible. For example, many internet-connected devices, used for personal health care, fitness, sports, entertainment and tracking, now use Bluetooth Low Energy to communicate with smartphones and tablets, including iPhones, Android phones, Windows and BlackBerry.

BLE is attractive to consumer electronics and internet-connected machine manufacturers due to its low cost, long battery life, and ease of implementation. From thermometers and heart rate monitors, from smartwatches to proximity sensors, Bluetooth Low Energy facilitates short-range wireless data transmission between devices, powered by a simple watch battery.

It introduces two new protocols to the standard:

- *GATT* (Generic Attribute Profile) is the layer at which the application has to interface and defines the data transfer protocol between two devices. As shown in Figure 3.6, it is based on a set of hierarchical entities:
  - *Attribute*, which identifies the application data;
  - *Characteristic*, composed by a group of Attributes. It adds additional properties such as permissions and rules of interaction for a unique set of data. Apart from the Characteristic value, there may be other attributes within each Characteristic, called *Descriptors*. For example, in order to identify the temperature unit of a thermometer, a Descriptor containing the information could be added to the Characteristic;
  - *Service*, composed by a group of Characteristics. It adds information for a given feature or functionality.

  For Example, Battery Service includes a Battery Level Characteristic that describes through an Attribute the battery level of a given device.

- *ATT* (Attribute Protocol) defines the protocol of transferring the attribute data and includes GATT-related functionality such as Write Request, Write Response, Notification and Read Response. Therefore, GATT defines and creates appropriate attributes for a given application while ATT creates outgoing packets and parses incoming ones [27];

BLE devices are divided into two categories: central and peripheral. A central device is commonly powered by mains or by a large battery, so the

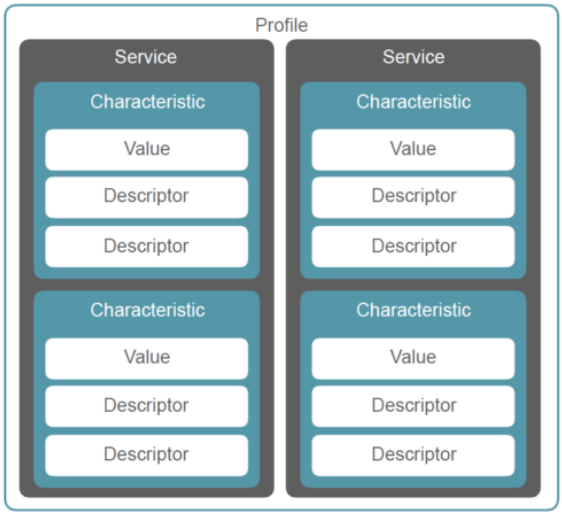


**Figure 3.6:** GATT hierarchy schema [26]

energy consumption caused by the Bluetooth radio is not a problem. Examples of this category of devices are represented by smartphones or computers. A peripheral device, instead, has energy constraints because its batteries are expected to last for years and Bluetooth should be used only when strictly necessary. Typically, the GATT server is represented by a peripheral device while the GATT client by a central one.

The possible ways of communication provided by the Bluetooth Low Energy technologies can be summarized by the following commands:

- *write*, when client or server sends some bytes to the other through a characteristic or descriptor in order to be processed and get a response;
- *read*, when client or server reads the value of a characteristic or descriptor and interprets it based on a protocol that has been established beforehand;
- *notify/indicate*, when a client subscribes to a characteristic for notifications or indications, and it is notified by the server when the value of the characteristic changes;

Notifications and Indications are the ways that allow the server to send a message to the client, avoiding the client from continuously requesting new data from the server. Indeed, if the Characteristic is correctly configured to accept subscriptions, the client can subscribe in order to receive updates on the data. The difference between the 2 sending mechanisms could be associated with the same reason that distinguishes the TCP from the UDP protocol, even if their technological stacks are completely different. While Indications require that the client sends an acknowledgment message to the server every time a

message is received, Notifications do not, increasing the transmission rate and decreasing the reliability.

## 3.3 Data exchange protocol

In order to make data transmission reliable and efficient, a new data exchange protocol has been defined. It establishes the communication rules that have to be applied in the BLE data transmission between the IoT device, which acts as a GATT server, and the citizens' smartphones, which run a GATT client represented by the mobile application.

On the server side, the protocol is based on the instantiation of a GATT Service with 2 GATT Characteristics, which will play the role of transmission channels:

- *measure_characteristic*, responsible for sending each chunk of a file along with its signature. It is configured in Notification mode such that a connected client can subscribe to value changes;
- *ack_characteristic*, used as a control channel. The client will notify the reception of an entire file, communicating also the outcome of the signature verification in order to assure that the file has been received without errors.

As shown in Figure 3.7, the communication between the devices follows these steps:

1. the mobile application detects the presence of a nearby station, connecting to the BLE service active on the IoT device;
2. through the acknowledgment channel, the app sends a command indicating to the server that transmission can begin;
3. one file at a time with its signature is taken from the buffer queue of the server and broken into chucks of maximum size 182 bytes;
4. as shown in step 7 of Figure 3.5, when the single file has been sent, it enters another queue waiting for confirmation of receipt;
5. the app verifies the signature of the file and sends back to the server the outcome through the acknowledgment characteristic;
6. if the outcome is positive, the file is removed from the queue. If not or the confirmation does not come back to the server, the file is reinserted in the transmission queue;

7. if communication errors happen, the connection is reset;

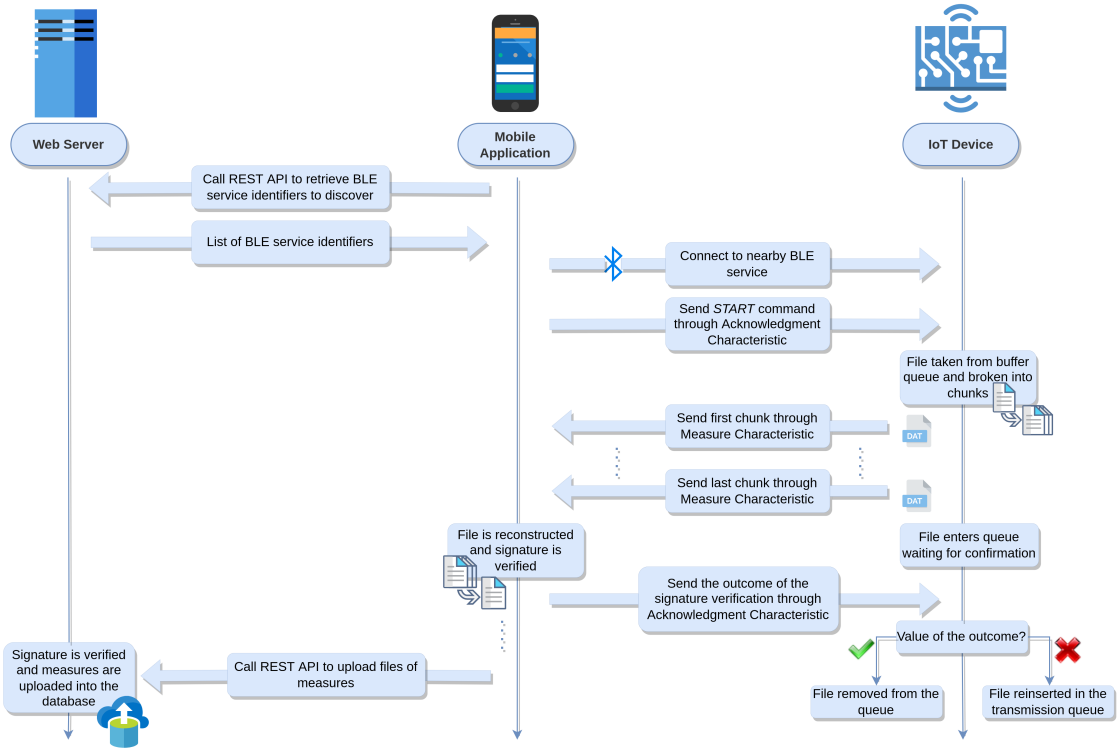


**Figure 3.7:** Data exchange protocol

The entire process for a single file containing measures of a temporal window of one minute takes place in less than 500 ms, improving the previous protocol where the same amount of data in CSV format was transmitted in 4-5 seconds. Reducing the file size, and therefore the transmission time increases the probability that the file arrives at the destination successfully because having fewer data to send, in terms of quantity of bytes, means a minor probability of transmission errors. Moreover, the decision of reducing the temporal window of the files from one day to one minute of measurements also derives from several factors:

- in case of transmission errors, only one minute of data would be lost and not an hour or even a day;
- since the concentration of particulate matter in the air can be widely variable, receiving data from one hour or one day ago could be useless for taking corrective actions;
- users could be close to the station only for a limited amount of time and, since BLE can work well only in a range of 10 meters, having an

efficient transmission is fundamental. Moreover, large files may not finish the transmission before the person leaves and be discarded by spending resources unnecessarily;

## 3.4 Implementation issues

The decision of migrating from a Raspberry PI to a microcontroller based on Pycom technologies as the core of the board has for sure brought benefits to the whole system, increasing the performance and reducing the power consumption that it is one of the most important objectives for an IoT system. However, one of the disadvantages introduced is related to the use of proprietary software represented by the modules that Pycom offers. During the development of this work, some issues were encountered but, since the implementation of these libraries must be taken as a black box, a real solution was not always found.

One of the problems was encountered during the development of the buffer queue system, the module implemented for interfacing with the file system. Initially, the file system was organized in a way such that all the measurement files were in a single folder. However, after a few days of data collection, I/O operations time started to increase drastically, from a few milliseconds to seconds. The reason for this deterioration was due to the number of files stored in the folder which had reached the order of thousands, making read and write operations slowly. The solution implemented was a renovation of the organization of the file system, trying to limit the number of files in a folder to a fixed number. As shown in Figure 3.8, a tree structure has been adopted where subfolders, named with the dates of the days in which the measurements are taken, are inserted in the root folder. Then, in each date folder, a maximum of 24 subfolders are created, one for each hour of the day, which can contain a maximum of 60 files inside, one for each minute of the hour. This approach limited the read-and-write operation time to less than 150 ms per file but, on the other hand, has made the logic and the file system more complicated.

Other issues were related to the signature generation of the measurement files. Pycom board has one physical processor available for user application and, since the signature process requires 2 seconds of computation, in that time interval acquiring data from the sensors is not possible, thus losing 2 seconds of data for each minute. Moreover, the execution of the function *generate_rsa_signature* requires a considerable amount of RAM that causes sometimes a *MemoryError* exception. Since there are no indications on the Pycom documentation about the RAM needed for the execution and how the function has been implemented, a workaround has been applied in order to increase the probability of not having error occurrences. The solution

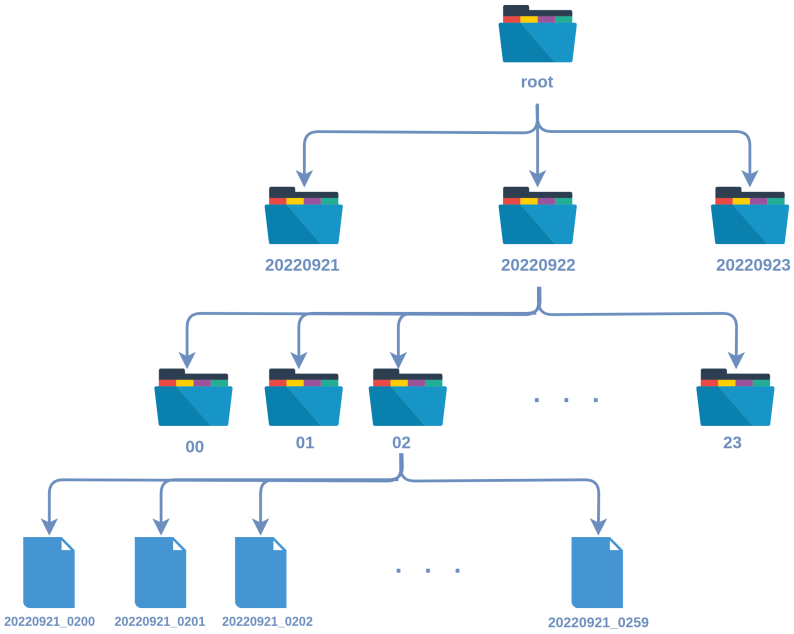


**Figure 3.8:** File system structure

proposed is represented by the use of the garbage collector in order to free unused portions of the memory that will be used by the function for its computations.

```
measuresBase64 = b2a_base64(buf.content)
try:
    buf.signature = crypto.generate_rsa_signature(
        measuresBase64,
        self.privateKey
    )
except MemoryError as e:
    gc.collect()
    buf.signature = crypto.generate_rsa_signature(
        measuresBase64,
        self.privateKey
    )
```

Finally, at the beginning of this work, one of the main issues was related to data transmission through BLE because many files were corrupted during the transmission and they were not understandable for the client. This problem was partially resolved by the introduction of the signature because it facilitated the clients for the verification of the integrity of a measurement file, allowing them to accept or discard it. However, the outcome of the verification had to

go back to the server in order to retry the transmission later or consider it successfully sent. This last step has been implemented through the use of the acknowledgment characteristic, where the client inform the server about the transmission and verification outcome.

## 3.5 Future improvements

This section aims to explore three of the main improvements that could be applied to the system.

One of them is represented by the file storage on the IoT device. When a file is sent to the gateway and the response of the gateway is positive about the reception and the outcome of the signature verification, the file is not removed by the system because there is no assurance that the file has reached the remote server. Since the memory, represented by the MicroSD peripheral, has a limited storage amount, a way to remove these files has to be found.

One solution can be represented by using a further Characteristic, instantiated on the GATT server, in order to allow the gateway to notify the board on which file has been correctly sent to the remote server. Otherwise, by making use of the LoRa antenna on the device and by building a more complex infrastructure by adding LoRa gateways, it is possible to create direct communication between the device and the remote server in low bandwidth. In this way, the device could query the server on the files that have been correctly received in order to remove them from the device storage.

A further improvement could be applied to the management of the private keys used for the signature process. Since a single pair of private and public keys has been generated and stored on the device, a security issue could occur: a device could be violated and the private key could be stolen, making all the data sent by all the other monitoring devices not verifiable.

In order to solve this problem, one solution is represented by having a different pair of keys for each monitoring device. In this way, even if one of them is violated, the others will not be affected. However, public keys should be accordingly diffused in a way such that each gateway can retrieve them. Therefore, public key could be added as an attribute of the device configurations on server side.

Finally, it is necessary to implement a security system that guarantees smartphones connected via BLE to be in communication with an official station and not with a fictitious one created by an attacker.

# Chapter 4

# Mobile Application

One of the architectural features that the research group wants to experiment with is the absence of an Internet mobile connection on the monitoring devices, thus avoiding direct communication between the devices and the remote server. A participative approach has been adopted in order to ensure that the data produced by the sensors reach the server, involving the citizens' smartphones in the transmission process acting as a gateway.

The business logic of the participative approach was implemented through a mobile application. It was developed using the cross-platform framework Flutter which allows the possibility of building a unique version of the code in order to deploy it on multiple platforms (e.g. Android and iOS operating systems).

The main functionalities offered by the app were implemented in previous work on this project [12] and they are included, as showed in Figure 4.1, in the following user interfaces of the app:

- authentication, offered by multiple login procedures: traditional one with email and password, Facebook login and Google login;
- home, which is dedicated to data visualization. It exploits visual components such as charts, graphs and maps according to the position of the user. Here, the concentration of $PM_{10}$ and $PM_{2.5}$ are used to compute the air quality index (AQI) according to the NowCast algorithm, giving the perception of the air quality to the user;
- map, which shows the current position of the user and where the monitoring devices are located around the city. In this section, the user can search for a destination to reach and the map will show, according to the AQI, the best path to follow;
- settings, where the user can select their preferences about the language and the measurement unit;

- profile, where a section is dedicated to the favorite stations of the user and another one to the account information, with the possibility to log out from the app.

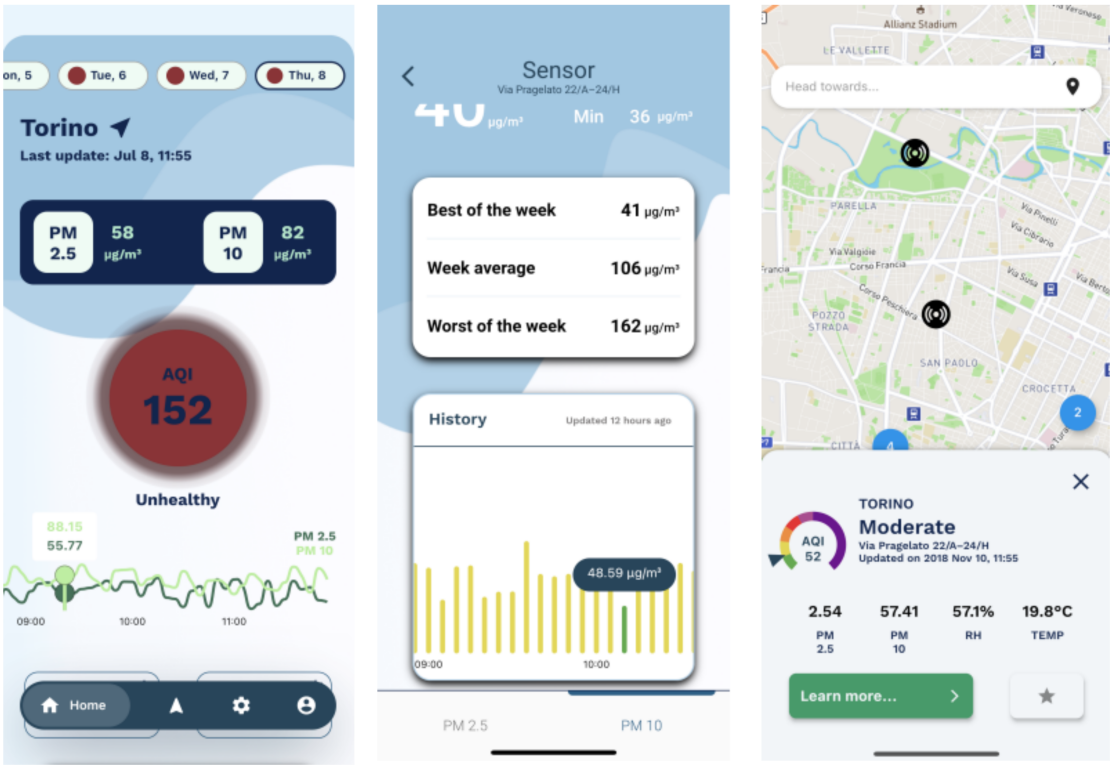


**Figure 4.1:** App screenshots

In the context of this work, the participative system was revised in order to adapt it to the new communication protocol between the mobile application and the monitoring device, implementing a background process that can connect and receive data without requiring any action from the user.

## 4.1 Flutter framework

The continuous evolution of mobile devices has pushed developers to understand in depth the mechanisms and the guidelines of the main operating systems, iOS and Android, in order to reach as many users as possible through the various markets. Although the evolution of these OS and the introduction of new languages and approaches have simplified and speeded up the development of native applications, it is still difficult for a single developer to able to create and maintain a native app for the different platforms over time.

The main reason lies in the absence of a shared codebase between the apps for the different mobile platforms.

In order to solve this problem, several frameworks have been proposed for creating cross-platform or hybrid apps in HTML5 and Javascript, which exploit the typical approaches of native apps and web apps. Some examples are Apache Cordova and Ionic which have allowed many developers to greatly reduce the development time of their apps, simplifying also the maintenance of a single codebase over time.

In 2018, a new framework developed by Google entered the landscape of cross-platform app development which allowed its developers to create their native apps for Android, iOS and Windows with native interfaces and shared code based on the Dart programming language. Flutter is a free open-source project for building high-quality native apps on iOS and Android quickly and with native interface support. Its goal is to create new apps through:

- a rapid development phase with features such as hot reload, which does not require you to recompile the code;
- expressive and flexible user interfaces with a set of modular widgets, libraries for animations and a layered and extensible architecture;
- performances very close to native ones;
- a unique codebase for Android and iOS applications;

Flutter's "everything is a widget" strategy applies object-oriented programming to everything, including the user interface: an app interface is thus composed of various widgets, which can be nested within each other. Each button and displayed text is a widget containing several features that can be changed. Widgets can influence each other and react to external state changes via integrated functions. For all the main elements of the user interface, the respective widgets are supplied in such a way that meets the design requirements of Android and iOS and the most common web applications. If necessary, the widgets can be expanded with additional functions or it is possible to create custom widgets that can be perfectly combined with existing ones. Compared to tools in other SDKs, widgets offer much more flexibility but have the disadvantage that they are all located in the program's source code, which is therefore heavily nested and intricate.

Compared to other frameworks, Flutter is built in a completely new way, allowing you to create simple, high-performance applications. To make this possible, Flutter runs on each platform natively using AOT (Ahead-Of-Time) compilation, while during the development phase, JIT (Just-In-Time) compilation is used to make the testing process faster. Each widget is interpreted and

represented on a Canvas managed by the Skia graphic engine. The platform shows the widget thus constructed to the end user and intercepts and forwards the events resulting from the interaction to the app.

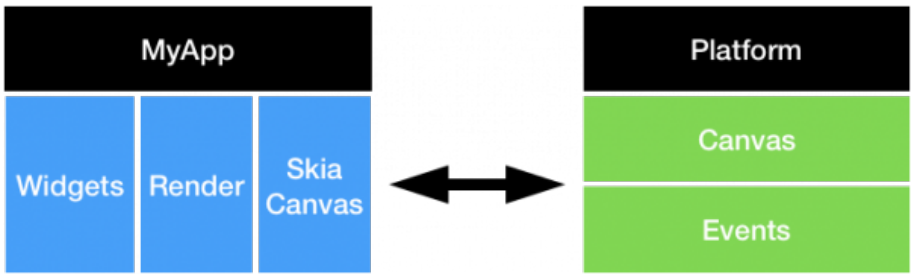


**Figure 4.2:** High-level representation of the communication between a Flutter app and the mobile platform [28]

The Flutter architecture, shown in figure 4.3, consists of three main macro blocks composed in turn of APIs and libraries that characterize each layer:

1. Embedder - Platform Specific.

   It is the lowest level of Flutter's architecture and it is the heart of the Flutter Engine. In this layer, the specific embedders for the platforms are defined, which have the purpose of tying together the rendering to the toolkit of the native screen, the management of input events, etc. In order to do that, the embedders interact with the Engine layer via low-level C / C++ APIs. However, these APIs are only exposed internally. If the developer needs to implement a particular behavior, he can use high-level integration APIs for the Android and iOS platforms.

   This layer also consists of a Shell that hosts the Dart VM. In particular, the Shell is specific to each platform and offers access to the native API of the platform in question. Shells implement specific code, such as communication with Input Method Editors (IMEs) and app lifecycle events based on the operating system of interest;

2. Engine.

   This middle layer is the C / C++ side of Flutter and it is defined in the repository engine. In particular, the Engine includes multiple low-level components, essential for the functioning of the framework and its basic operations. Among these components, we find the Skia graphics engine and the shells that can be accessed through the APIs exposed by the library *dart:ui*. Specifically, it is possible to create an app using the classes defined in this library such as Canvas, PaintandTextBox;

3. Framework.

   It is the most important layer for developers and offers all the libraries and packages necessary for developing an app, such as the layers relating to animations, the definition of gestures, or the creation of widgets. Generally, during the development of an app it will be much more common to work with the layers of this component of the architecture, composing widgets and animations starting from those already provided by Flutter itself or creating their ad-hoc ones.

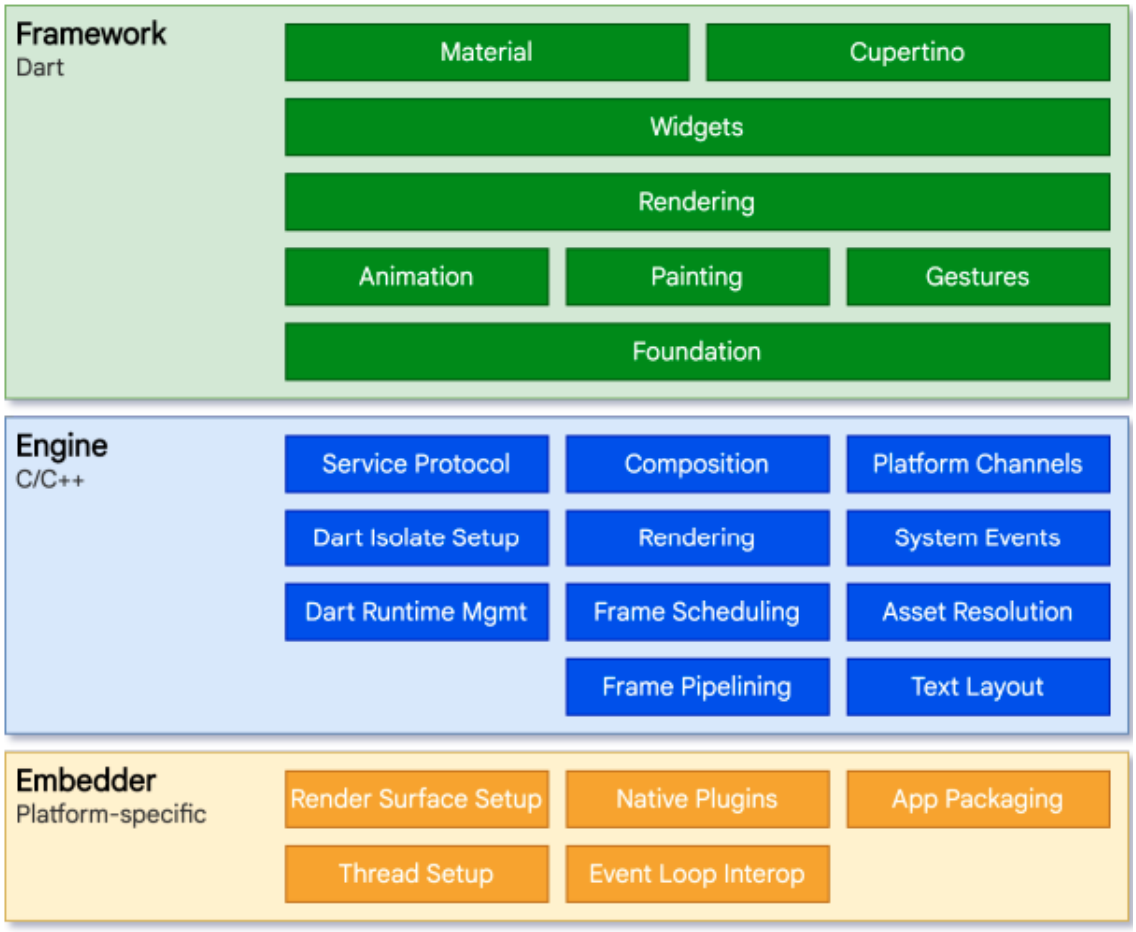


**Figure 4.3:** Flutter architecture [29]

### 4.1.1 Cross-platform frameworks comparison

Mobile application development can follow multiple approaches depending on the technologies used.

The first approach consists of the use of the native platforms of the device, such as iOS and Android SDKs, which represent the most stable choice

for mobile application development, also for the large community and the available tutorials. As shown in the diagram in Figure 4.4, the app is in direct communication with the system, choosing this framework as the best one in terms of functionalities and performance. However, a drawback is represented by the fact that there is no unique codebase to develop and maintain and the programming languages are platform-specific: Kotlin or Java for Android, Obj-C or Swift for iOS. Therefore, in order to build an app with the same functionalities on all the platforms, all the platform-specific languages must be used, complicating even the process of modifications of the code that should be duplicated across the platforms. This kind of framework is not a good choice for a small team or for projects that need speed in the development process.

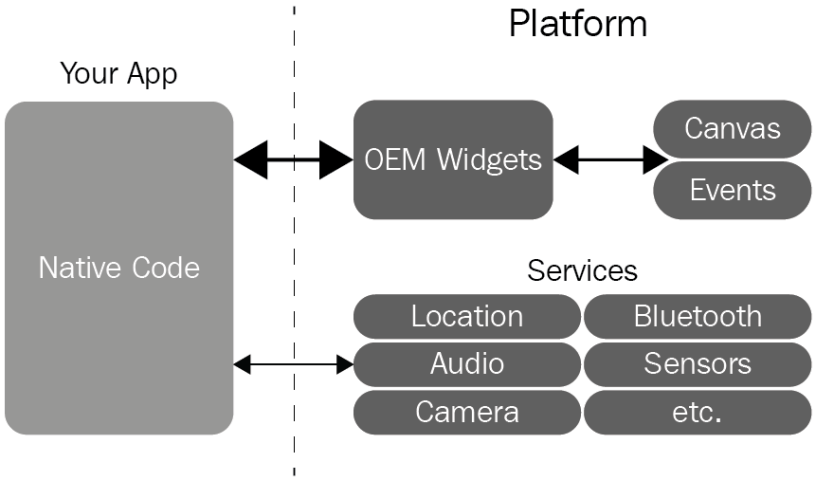


**Figure 4.4:** Native framework architecture [30]

A second approach is represented by the use of a cross-platform framework, which enables the possibility of building platform-specific applications from a single codebase. However, every framework has some drawbacks.

In the early time of mobile computing, several WebView-based frameworks have been proposed such as Cordova, Ionic and PhoneGap. The use of these frameworks has allowed developers to build applications composed of a WebView rendering HTML, which transformed native mobile application development into website development. Despite the considerable simplification introduced, unfortunately, these frameworks do not allow for fully exploiting the typical characteristics of the various mobile platforms, and suffer from greater slowness in execution, and access to local resources and peripherals (e.g. cameras, sensors). The reason is the fact that these kinds of frameworks rely on external components called plugins, which act as a bridge between the

operating system and the application itself. The bridge can switch between JavaScript running on the native web view to the native system. The following diagram in Figure 4.5 shows how a WebView-based framework works:

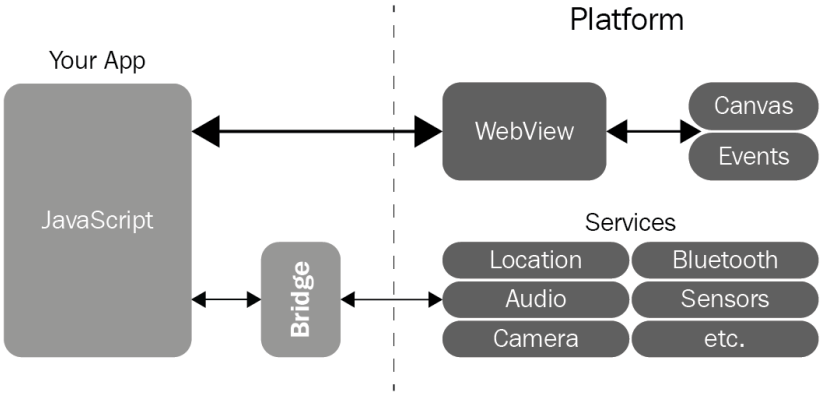


**Figure 4.5:** WebView-based framework architecture [30]

Since Flutter is compiled AOT (Ahead Of Time) instead of JIT (Just In Time) during the development phase, its performances are much better compared to the other solutions. As shown in Figure 4.6, it eliminates the bridge element between the application and the operating system and does not rely on the OEM platform, using Widgets Rendering instead of working with the canvas and events. It uses Platform Channels in order to call the services (e.g. Location, Bluetooth, ...), simplifying the use of platform APIs with an asynchronous messaging system. The framework allows to build, integrate and publish plugins, even developed by other developers, in order to use platform-specific services and functionalities.

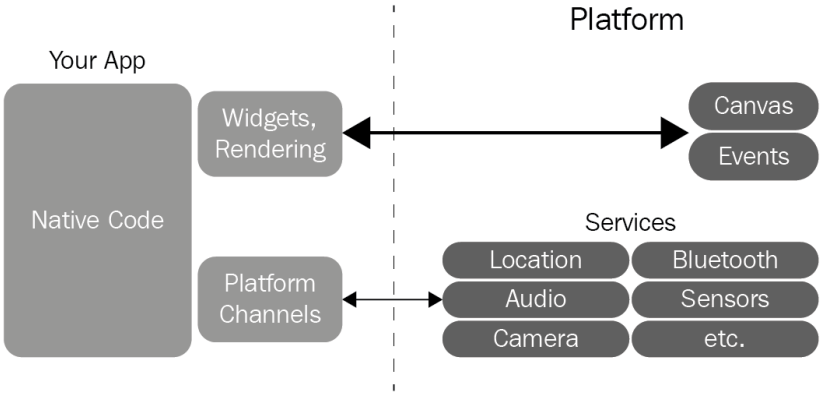


**Figure 4.6:** Cross-platform framework architecture: Flutter [30]

## 4.2 Participative system implementation

During the previous work, a prototype of the participative system on the mobile application has been already developed. However, since the data management and transmission have been changed on the monitoring device adopting a new communication protocol, the prototype has been adapted accordingly. Moreover, the existing prototype suffers from problems that could compromise the experimentation. One of the main issues was related to the limitations imposed by the operating systems in the use of Bluetooth Low Energy, forcing its use only when the app is up and running in order to improve energy consumption. This limitation could damage the participative approach of the system: users should pay attention to being near a station and open the app, increasing the effort and attention required from the user.

For this reason, in the context of this thesis work, there is the aim of making the participative way of the system as transparent as possible for the user in order to discourage the app uninstallation, implementing a background process that can run even if the app is closed. The background process on Flutter has been implemented with the integration of the plugin *flutter_background_service* which allows starting a process detached from the UI thread. It is initialized at the first run of the app and will be notified about the state of the app lifecycle in order to react accordingly. Since the operating systems have restrictive policies on the use of resources by apps that are not running up, they may kill background processes at every moment. In order to keep processes up and running, one possibility is represented by disabling the app from the battery optimization process of the operating system. As shown in the code snippet below and Figure 4.7, through the integration of the *optimize_battery* plugin, the battery optimization settings are opened before the process starts in order to give to the user the possibility of excluding the app from the optimization process.

```
1    static Future<void> _checkIgnoreBatteryOptimization() async {
2      bool ignoreBatteryOptimization = await
3          OptimizeBattery.isIgnoringBatteryOptimizations();
4      if(!ignoreBatteryOptimization){
5        await OptimizeBattery.openBatteryOptimizationSettings();
6      }
7    }
```

However, even if this action solves the problem on Android, on iOS there is no way of having long-running services because, when the app is running in the background, the OS will suspend it soon. Currently, the *flutter_background_service* provides the *onBackground* method, which allows the

process to be executed periodically by *Background Fetch* capability provided by iOS but its execution frequency cannot be faster than 15 minutes and the process will be alive only for about 15-30 seconds.

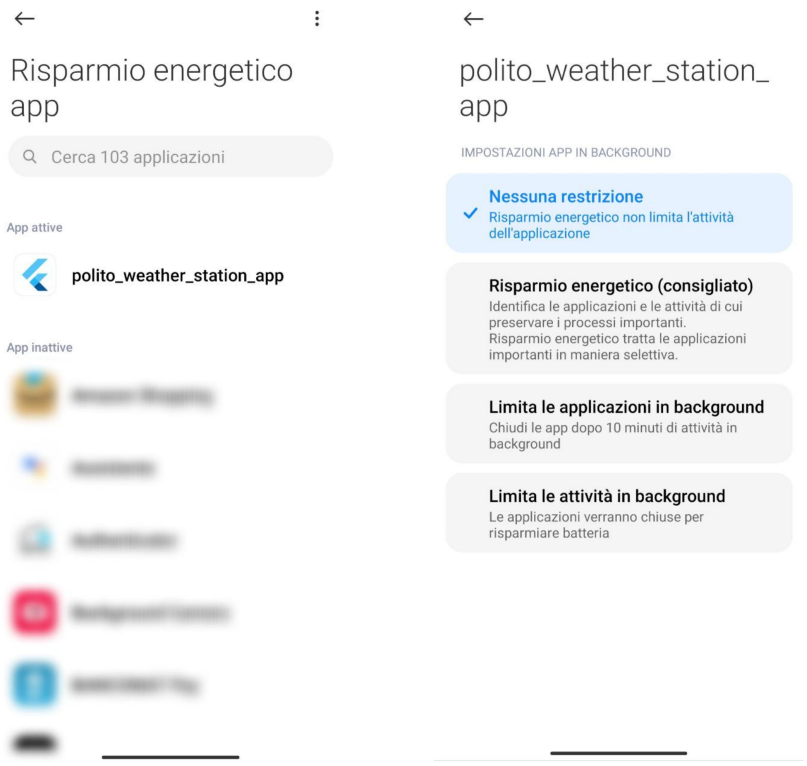


**Figure 4.7:** Battery optimization settings on Android

The main goal of the background process is to exploit the Bluetooth Low Energy in order to scan for nearby monitoring devices, establish a connection, receive and process the data applying the communication protocol and send them to the remote server. The Bluetooth Low Energy functionality is accessed through the integration of a further plugin, *flutter_blue_plus*, which replaced the previous one, *flutter_blue* because it can also be used by a process detached from the UI thread. The BLE scan routine is activated through a periodic timer that triggers the *_listenBLEScanResults* function every 5 seconds. The scan routine is not interrupted when a nearby device is already connected because, in case a closer device is found, the connection should be migrated to the new one. The proximity of a device is provided by the plugin through the RSSI parameter which is used in telecommunications in order to measure the power of a signal. When a connection is established, communication between the devices starts. The app starts to collect the file chunks sent by the monitoring device through the *measureCharacteristic* along with the metadata

needed to recognize the beginning and the end of each file. Then, the integrity and authentication of the file are verified through the signature verification process in order to send back to the monitoring device the outcome of the verification through the *ackCharacteristic*. At the end of the process, the pair, composed of the data file and signature one, is stored in the local storage of the device. The following code snippet shows the main steps just described:

```
Future<bool> _handleCharacteristicValue({
    @required BluetoothDevice device,
    @required BluetoothCharacteristic ackCharacteristic,
    FileInfo fileInfo,
    List<int> values
}) async {
  if (fileInfo.needMetadata) {
    ... // set metadata information
  } else {
    fileInfo.appendData(values);

    if(fileInfo.completed){
      logger.wtf("END RECEIVING DATA AT ${DateTime.now()}");
      Map signature = await fileInfo.verifySignature();
      if(signature['ok']) {
        String pathMeasures = await fileInfo.measures.store();
        String pathSignature = await fileInfo.signature.store();
        if(pathMeasures != null && pathSignature != null){
          await ackCharacteristic.write(
              signature['name'].codeUnits + ',OK'.codeUnits,
              withoutResponse: true
          );
        }else{
          logger.e("Error while saving measures/signature in localStorage");
        }
      } else {
        await ackCharacteristic.write(
          signature['name'].codeUnits + ',KO'.codeUnits,
          withoutResponse: true
        );
      }
    }
  }
  ...
}
```

The background process is configurable for working in 2 different ways. When the app is up and running, BLE is the unique way of interacting with the environment, continuously scanning for nearby devices. Instead, when the app is in background or is closed, the first approach, the default one, is to

keep using BLE even in case. The second one is represented by the use of the GPS sensor on mobile smartphones. Since the position of each monitoring device is known, the app can monitor the position of the user and warn him when the distance from the monitoring device is under 10 meters. It is done through the sending of an in-app notification which should be tapped in order to open the app and start the data transmission through BLE. The user has the ability of choosing how he could be involved in the participative process by choosing in the app settings one of the configurations for the background process. It is also possible to switch off the process in order to use the app only as a data visualizer.

### 4.2.1 BLE limitations

The use of Bluetooth Low Energy and GPS location requires the granting of some permission from the user that should be asked accordingly. The permission have to be added to the project in the following files of the 2 platforms:

- in the android/app/src/main/AndroidManifest.xml:

```
<uses-permission android:name="android.permission.BLUETOOTH" />
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
<uses-permission
    android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

- In the ios/Runner/Info.plist:

```
<dict>
 <key>NSBluetoothAlwaysUsageDescription</key>
 <string>Need BLE permission</string>
 <key>NSBluetoothPeripheralUsageDescription</key>
 <string>Need BLE permission</string>
 <key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
 <string>Need Location permission</string>
 <key>NSLocationAlwaysUsageDescription</key>
 <string>Need Location permission</string>
 <key>NSLocationWhenInUseUsageDescription</key>
 <string>Need Location permission</string>
```

Since the background process is intended to run even if the app is not up and running, permission should be granted not only when the app is in use but always. Even if the process will use only the BLE, location permission is needed because Bluetooth has direct access to the device's MAC address for the

purpose of pairing. If the MAC addresses of WiFi or Bluetooth transmitters are readable, a device could be located. Therefore, the requirement to allow location services to use Bluetooth is about ensuring that someone who disables location keeps their location private.

A limitation imposed by the mobile application is the file chunk size used for the data transmission. This parameter has been influenced by the Maximum Transmission Unit (MTU) value that has been set to 185 bytes. The reason is due to the fact that, even if the MTU value is negotiated across the devices during the pairing process, each device has its maximum value. In this case, the MTU of the monitoring device could be set between 23 and 200 bytes and Android devices could arrive even up to 512 but iOS imposes a maximum value equal to 185 bytes. Therefore, the MTU value has been set for all the devices to 185 bytes and the file chunk size to 182, for the ATT header size of the BLE packet which occupies 3 bytes.

Another problem, mostly on Android, is represented by the use of Bluetooth. Even if the app has been excluded by the battery optimization process, the operating system tends to stop Bluetooth scans after a while because it is not efficient for the battery energy. However, by applying a filter on the scans programmatically, it is possible to make it work for a long time. In particular, the filter could be applied to the GATT service identifiers that we want to discover. Since for this project every station is configured on the server along with the BLE configurations, it is possible to retrieve them through an API in order to insert them in the BLE scan filter. The following code snippet shows how the filter on the scan is applied:

```
1       _flutterBlue.startScan(
2         timeout: Duration(seconds: 4),
3         withServices: boardsConfig["ble_service_uuid"]
4           .values
5           .map((s) => Guid(s))
6           .toList()
7       ).then(...)
```

## 4.3 Interaction with the Server

The mobile application is a client of the remote web server and the communication is made through REST APIs. On Flutter, an useful plugin that implements an http client for the Dart programming language is *dio*, which supports functionalities like interceptors, requests cancellation, cookies management and file uploading and downloading.

Besides the APIs to download the data to be displayed to the user, an

important functionality is represented by the uploading of the files received from the monitoring device to the remote server, since it is the last hop of the data transmission. The following code snippet shows how it has been implemented:

```
Future<...> sendMeasureFiles(List<String> paths, ...) async {
  ...

  FormData formData = FormData.fromMap(files);
  try {
    Response<List<dynamic>> response = await dio
      .post<List<dynamic>>(
        '/api/measures',
        data: formData,
        options: Options(
          receiveTimeout: 30000,
          validateStatus: (int status) => status == 200,
          headers: {"x-access-token": token}
        )
      );

    ...
  } catch (error) {
    rethrow;
  }
}
```

As mentioned in the previous section, when the files arrive to the mobile application through BLE and the signature is verified, they are stored in the local storage of the device. In order to send them to the server, the background process has been implemented to have also a procedure in charge of monitoring the directory where the files are stored. It can be considered as a file watcher routine that is triggered every 20 seconds by a timer and it checks if the folder contains new files to send. The first action of the routine is to check if there is an Internet connection in order to send the request to the server. Then, it looks for the pair of data and signature files associated by their names that should be equal. Since the API of the server has been developed in order to accept more than one file per request, the routine looks for a maximum of 20 files and, once the request has been sent, it waits for the response. If successful, the response contains a list of the file names just sent, with the outcome of the processing for each of them. Each file that has a value of the outcome equal to *VALID* is removed from the local storage.

# Chapter 5

# Server

This chapter is in charge of describing the core of the system, the server. It is the unique interface to the database and offers the functionalities to apply the CRUD (create, retrieve, update, delete) operations on its tables, applying the business logic needed to guarantee a certain level of service.

At the beginning of this thesis, a server has already been implemented but the research group decided to renovate and improve it. It acts as a web server, offering web services via the Internet to the mobile application and a web dashboard used by the administrators of the system, which is still in the development phase. The development is based on the Python programming language and uses some frameworks that facilitate the implementation of the functionalities, such as Flask, SQLAlchemy and Marshmallow.

Web services, also called web application program interfaces (API), consist of public endpoints that accept requests in order to produce responses, formatted according to the JSON format. The APIs follow the REST paradigm that is based on the HTTP protocol.

The APIs make extensive use of the database, due to the considerable amount of data that is processed. One of the renovations introduced with this work is the use of the ORM paradigm in order to simplify the way to access and operate the database. Moreover, the introduction of the new communication protocol made necessary the adaptation and improvement of existing APIs and procedures.

The database is a MySQL database and it is structured in order to contain the measurements, the sensors and the boards' information, the configurations and some other entities.

## 5.1 REST paradigm

REST, or REpresentational State Transfer, is an architectural paradigm to provide a way to facilitate communication between the systems on the web. It is characterized by a separation of concerns between client and server where the systems try to be as stateless as possible.

The separation between client and server means that the implementation of the two systems is independent from each other and changes on one system will not affect the operation of the other one, considering also the possibility to evolve independently. Moreover, knowing only the interface of a system and the way to interact with increases modularity, flexibility and scalability.

In the REST architecture, clients send a request to the server in order to receive a response. The aim of the request could be for retrieving resources or modifying them on the server. A request generally consists of:

- HTTP method, which defines the operation to execute;
- headers, allowing the client to add information about the request;
- a path to a resource;
- an optional body, which contains a content or a message that the client wants to transmit;

Since the REST paradigm is based on the HTTP protocol, there are 4 basic operations to interact with the resources of a REST system:

1. GET, used to retrieve specific resources;
2. POST, used to create a new resource;
3. PUT, used to update a specific resource;
4. DELETE, used to remove a specific resource;

In the HTTP protocol, the GET (read), POST (create), PUT (update) and DELETE (remove) methods are considered CRUD operations as they have precise semantics relating to the management of data storage. In fact, they allow the client to directly manipulate the status of the web resource of interest.

Each request from the client corresponds to a response from the server that contains the status code of the request, alerting the client if it has been successful or not. The status code is a number of 3 digits and the first one represents the category of the outcome. The possible values are:

- 1xx is an informational response indicating that the request has been received and the processing can continue (e.g. 100 Continue);
- 2xx indicates that the request has been successfully received and processed (e.g. 200 OK, 201 Created);
- 3xx indicates that the client must take further action to fulfill the request (e.g. 304 Not Modified);
- 4xx indicates that the request is syntactically incorrect or cannot be satisfied (e.g. 400 Bad Request, 401 Unauthorized, 404 Not Found);
- 5xx indicates that the server failed to fulfill a valid request (e.g. 500 Internal Server Error, 502 Bad Gateway);

## 5.2 ORM paradigm

Object-Relational Mapping (ORM) is a technique that allows to query and manipulate data from a database using an object-oriented paradigm, which is implemented in several libraries of various programming languages.

Each ORM library encapsulates the code needed to manipulate the data, interacting directly with the object in the language that has been chosen and avoiding using SQL. It creates a virtual object database. However, many popular database products such as SQL database management systems (DBMS) are not object-oriented and they are often not able to manipulate and store complex types, treating the data as scalars, such as integers and strings, and organizing them into tables. The conversion of the object values into groups of simpler values is required in order to store them in the database. In the same way, a conversion back to complex types on retrieval is needed in order to avoid using simple scalars in the program. This is what Object-relational mapping is intended to implement and, in case both conversions are implemented, the objects are said to be persistent, abstracting the implementation characteristics of the specific DBMS used.

The main advantages of using this technique are the following:

- the overcoming (more or less complete) of the fundamental incompatibility between the object-oriented paradigm and the relational model on which most of the current DBMS used are based;
- high portability concerning the DBMS technology used: when changing DBMS the routines that implement the persistence layer do not have to be rewritten; generally, it is enough to change a few lines in the product configuration for the ORM used;

- drastic reduction in the amount of source code to be written. Behind simple commands, ORM masks the complex activities of creating, collecting, updating and deleting data (CRUD). Such activities usually take up a good percentage of the overall development, testing, and maintenance time. Furthermore, they are by their nature very repetitive and, therefore, favor the possibility that mistakes are made during the development of the code that implements them;

- it suggests the implementation of the architecture of a software system based on a layered approach, thus tending to isolate the logic of data persistence in a single level, with the aim of increasing the modularity of the system;

The currently most popular ORM products often offer natively features that would otherwise have to be created manually by the programmer:

- automatic loading of the graph of the objects according to the association bonds defined at the language level. The loading of an instance of a class could automatically cause the loading of the linked data. In addition, this loading can only take place if the data are requested by the program, otherwise, it is avoided (a technique known as *lazy-initialization*);

- concurrency management in data access during transactions. Conflicts during the modification of data by several users at the same time can be automatically detected by the ORM system;

- data caching mechanisms. For example, if the same data are fetched several times from the DBMS, the ORM system can automatically provide caching support that improves the performance of the application and reduces the load on the DBMS system;

- management of a conversation using the Unit of Work design pattern, which delays all data updating actions when the transaction is closed; in this way, the requests sent to the DBMS are those strictly indispensable (e.g. only the last of a series of updates is performed on the same data, or a series of updates is not performed at all on a data that is subsequently deleted); moreover, the dialogue with the DBMS takes place by composing multiple queries in a single statement, thus limiting the number of round-trip-times required and, consequently, the application response times to a minimum;

## 5.3 Frameworks used

The development of the server and its REST APIs were made through the framework Flask. It is not considered a full web development framework because its light installation offers minimal and basic functionalities, making it in fact a micro-framework. The main features of Flask are the following:

- a built-in development server and debugger;
- integrated support for unit testing;
- RESTful request dispatching;
- uses Jinja templating, allowing the creation of HTML, XML, or other markup files, which are returned to the user via an HTTP response;
- support for secure cookies;
- error handling, for aborting a request with an error status code;
- static routing, to trigger a specific function corresponding to a certain URL;
- dynamic routing, to trigger a specific function corresponding to a set of URLs, each of one containing parameters that will be used by the function;

Flask uses the concept of *Blueprints* to create sub-components of the application sharing the same configurations and the same patterns. Blueprints can simplify the management of a large application, making it more modular. Each Flask Blueprint is an object that, similarly to a Flask application, can have resources, such as static files, templates, and functions that are associated with routes. However, it is not an application and it needs to be registered in a Flask application before being run by calling *Flask.register_blueprint()*. When a Flask Blueprint is registered in an application, it extends the application with its contents. It is a great modular way to organize the application, defining also a dynamic URL prefix that will be applied to all routes in a blueprint.

Moreover, Flask can be extended with third-party libraries. In the context of this project, one of the libraries that have been integrated is *SQLAlchemy*, an Object Relational Mapper (ORM) tool that translates Python classes to tables on relational databases and automatically converts function calls to SQL statements. It is able to provide a standard interface that allows developers to create database-agnostic code to communicate with a wide variety of database engines. In order to interact with a database, SQLAlchemy

needs to instantiate an *Engine*, responsible of managing two components: *Pools* and *Dialects* [31]. The library supports the most common database management systems available on the market, such as PostgreSQL, MySQL, Oracle, Microsoft SQL Server, and SQLite. The Connection Pools are based on the *object pool pattern* where, instead of spending time creating objects that are frequently used, the program fetches an existing one from the pool. In the database connections context, this pattern allows easier management of the number of connections that an application might use simultaneously [31]. The Dialects are the proprietary variations introduced by the various DBMS in the SQL standard. SQLAlchemy enables Python developers to create applications that communicate to different database engines through the same APIs, creating a separated layer between the Python code and the database engine used [31].

Another library integrated into this project is Marshmallow. It converts complex data types to and from Python data types. It is a powerful tool for both validating and converting data for serialization and deserialization. It is based on the concept of *Schema* that defines the structure of the data and also the validation to apply and provides functions to serialize and deserialize to and from JSON format.

## 5.4 Database structure

At the beginning of this thesis, the structure of the database, a MySQL relational database, was already provided by the research group. The basic entities are represented by the tables of measures, sensors and boards, together with all those necessary for the enrichment of the information and for describing the relationships. A representation of database relational schema is shown in Figure 5.1.

Each measurement inserted in the *measure_table* has the information about the sensor that produced the data. In particular, this information is a foreign key that references the *logical_sensor_table* which in turn references the *unit_of_measure_table*, containing the unit of measure of the data produced by a sensor.

A logical sensor is linked to a physical sensor, represented by the table *physical_sensor_table*, with a relationship many to one. The concept of the logical sensor has been introduced for distinguishing the sensor physically mounted on a board from the measurement that it produces. In order to better clarify this concept, one example is represented by the PM sensor: even if a unique physical PM sensor is mounted on the board, it produces two different measurements, $PM_{10}$ and $PM_{2.5}$, as they were produced by two different logical sensors. Each physical sensor and each board, represented by the *board_table*,

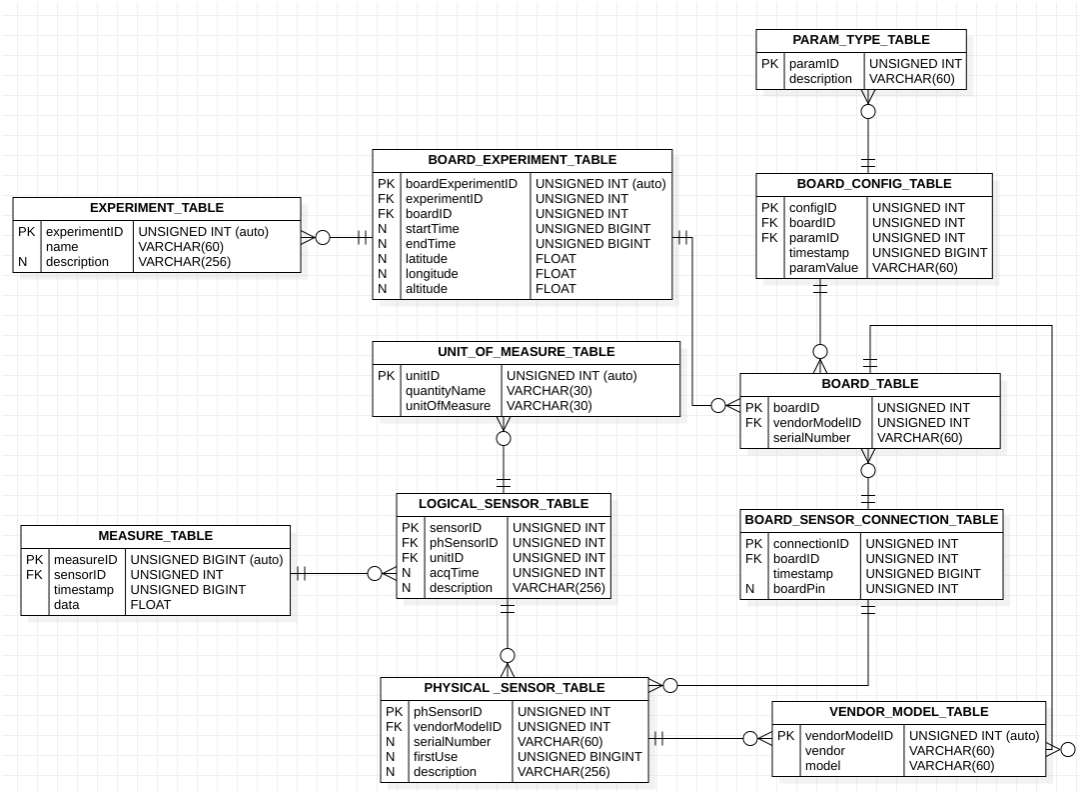


**Figure 5.1:** Database relational schema

have their vendor and model that are described in the *vendor_model_table*. The connection between a physical sensor and a board is described in the *board_sensor_connection_table*.

In the context of this work, in order to make the participative system work, each board should be associated to one or more experiments, each of one contained in the *experiment_table*, through the *board_experiment_table*. The association is made through a manual configuration from a web dashboard, setting also the location of the board for the specific experiment, represented by the properties *latitude*, *longitude* and *latitude*.

Moreover, each board could have a set of configurations that are inserted in the *board_config_table* and all the possible configuration types are described in the *param_type_table*. Indeed, all the configurations needed for the BLE access point of a board are described here. In particular, the *param_type_table* describes the generic configuration names (shown in Table 5.1), such as the name of the access point and the UUIDs for the service and characteristics, and the *board_config_table* describes their values for a specific board (shown in Table 5.2).

There is also a *user_table* which is in charge of storing the user's information handled by the mobile application to guarantee the authentication functionality.

| ParamId | Description |
|---|---|
| 1 | ble_ap_name |
| 2 | ble_service_uuid |
| 3 | ble_measure_char_uuid |
| 4 | ble_ack_char_uuid |

**Table 5.1:** BLE configurations in *param_type_table*

| BoardId | ParamId | ParamValue |
|---|---|---|
| 1 | 1 | weather-station-1 |
| 1 | 2 | a7dd77c8-1088-48ba-b0da-8ddefc069aa9 |
| 1 | 3 | 00054326-0000-1000-8000-00805f9b34fb |
| 1 | 4 | b12ef43c-3f2c-4c3c-a52a-524d758f854c |

**Table 5.2:** BLE configurations in *board_config_table*

## 5.5 Code structure

The code implemented for the server is structured in a way such that the Flask application is started in the file *app.py*. Here, during the boot phase of the application server, the application registers the Blueprints and initializes the singleton components such as the mail server and the database session through SQLAlchemy.

The core of the application is represented by the business logic which is contained in 4 main folders: *routes*, *services*, *models* and *schemas*.

The *routes* folder contains all the controllers of the application. Each Blueprint defines a route that is linked to a controller, representing the interface exposed outside of the application and contains the APIs. The aim of the functions defined in a controller is to retrieve eventual parameters or messages within a client request and return a response containing the HTTP status code along with a response body or an error description. The response is generated starting from the input supplied through the call to a function defined in another module, the service. Below is an example of the API defined in *measures_controller.py* to upload the file containing the measurements:

```
1  @measures_controller.route('', methods=['POST'])
2  @token_required
3  @doc(tags=["measures"])
4  def upload_measures(current_user):
5      try:
```

```
6           results = MeasureService.upload_measures(request.files)
7           return make_response(jsonify(results), 200)
8       except FileNotFoundError:
9           return make_response(
10              jsonify({"message": "No file was uploaded"}),
11              404
12          )
13      except Exception as e:
14          return make_response(jsonify({"message": str(e)}), 500)
```

The *services* folder contains the implementation of the APIs and here there is the business logic of the application. Each function receives eventual parameters that are first validated according to the specifications and then used to produce an output, involving also other components of the application. The following code snippet shows the implementation of the API to retrieve all the boards with the optional parameter *modelId* provided as input:

```
1   @staticmethod
2   def get_all(modelId=None) -> List[Board]:
3     if modelId is None:
4       return Board.query.all()
5     else:
6       return Board.query.filter(Board.vendorModelId == modelId)
```

The *models* folder contains the classes used to implement the ORM paradigm. Each class represents an entity table of the database declaring all the columns with their data type, constraints and external relationships. A model extends the *Model* class of the SQLAlchemy library, responsible for providing the native functionalities to manage the connection, access the database and retrieve and store the data in the table. The following example shows the *Measure* model:

```
1   class Measure(db.Model):
2       measureId = db.Column(db.BigInteger(), primary_key=True)
3       timestamp = db.Column(db.BigInteger())
4       data = db.Column(db.Float())
5       sensorId = db.Column(
6           db.BigInteger,
7           db.ForeignKey('logical_sensor.sensorId'),
8           nullable=False
9       )
```

Last module, represented by the *schemas* folder, contains the classes with the rules for validating, serializing and deserializing the data from the object

model to the JSON format. Each class extends the *SQLAlchemyAutoSchema* class of the *Marshmallow* library that provides two main decorators, *pre_load* to clean input data and *post_load* to envelope the response data, and two functions for converting an object, *load* to validate and deserialize the input data to the model object and *dump* to serialize the model data to the output data. The example below shows the *BoardSchema* to convert to and from the *Board* model:

```
1  class BoardSchema(ma.SQLAlchemyAutoSchema):
2      class Meta:
3          model = Board
4
5      boardId = ma.auto_field(required=False)
6      serialNumber = ma.auto_field()
7      connections = fields.Nested(BoardConnectionSchema, many=True)
8      vendorModel = fields.Nested(VendorModelSchema)
9
10     @post_load
11     def make_board(self, data: Any, **kwargs) -> Board:
12         return Board(**data)
```

## 5.6 Implementation

The aim of this section is to describe the implementations and the changes applied to the web application during this work. It focuses on the application of the ORM paradigm with a renovation and adaptation of some new and already existing APIs, in particular the ones regarding the management of the users, boards and measurements, and procedures needed to aggregate the measurements.

At the boot phase of the server, besides the definition of the Blueprints along with their controllers, a background scheduler is instantiated to run two background jobs periodically. Given the considerable amount of data that are stored in the *measure_table*, the purpose of these jobs is to aggregate them in order to have data averaged on a temporal window. In particular, the jobs are the following:

1. *insert_data_five*, which aggregates the measurements by sensor every 300 seconds in a temporal window of 5 minutes and inserts them in the table *five_min_avg_measure*;

2. *insert_data_hour*, which aggregates the measurements by sensor every 61 minutes in a temporal window of 1 hour and inserts them in the table *hour_avg_measure*;

They work in a very similar way and the main steps are:

- retrieve the last aggregated measure produced by a specific sensor from the aggregated table;
- if there is no aggregated data for that sensor, retrieve, if exists, the first measure in the *measure_table*;
- get the acquisition timestamp from the result in order to compute the belonging temporal window;
- query the *measure_table* applying the temporal window filter on the acquisition timestamp;
- aggregate all the results computing the average, minimum and maximum value of the measurements and insert them into the aggregated table;

The two aggregated tables offer the possibility of having fast data retrieving when a client sends a request because their size is smaller than the *measure_table* and they allow to compute statistics in a easier way.

The main APIs that were implemented are included in the following controllers:

- *user_controller*, which handles the registration, login and password recovery functionalities to provide the authentication on the mobile application and session management through a JWT token. In fact, most of the APIs require a valid token to be executed successfully. It has to be inserted in the *x-access-token* field of the request headers and it is validated through the *token_required* decorator shown below:

```
1  def token_required(f):
2      @wraps(f)
3      def decorated(*args, **kwargs):
4          ...
5          token = request.headers['x-access-token']
6          if not token:
7              return make_response({"message": "Unauthorized"}, 401)
8          data = jwt.decode(token, ...)
9          current_user = User.query
10             .filter(User.user_id == data['sub'])
11             .first()
12         ...
13         return f(current_user, *args, **kwargs)
14
15     return decorated
```

- *measures_controller*, which offers the functionalities to upload the measurement files and to retrieve the aggregated data according to specific filters. The upload API is in charge of verifying the signature of each measurement file and converting the bytes content to the measurement object model in order to insert it into the database, as specified in the communication protocol. Its implementation is shown here:

```
@staticmethod
def upload_measures(files):
    ...
    for f in files.keys():
        out = {"filename": f, "status": MeasureStatus.VALID.name}
        try:
            verify_signature(key, files.get(f), files.get(signFileName))
        except (ValueError, TypeError):
            out["status"] = MeasureStatus.SIGN_NOT_VALID.name
        else:
            measures = bytes_to_measures(files.get(f))
            db.session.add_all(measures)
            db.session.commit()
        outcomes.append(out)
    return outcomes
```

- *boards_controller* and *board_config_controller*, which offers the APIs to create, manage, retrieve and delete the boards and their configurations. In particular, an API to retrieve the boards, according to a location sent in the request, has been implemented for the participative system. It returns all the board in a range of a maximum distance specified in the request. Its implementation in *BoardService* is shown here:

```
@staticmethod
def get_all_by_location(latitude: float, longitude: float, maxDistanceKm):
    boardIds = BoardService.get_all_ids()
    boardExperiments = BoardExperimentsService
        .get_board_experiments_between(boardIds, int(time.time()))
    nearbyBoards = []
    for exp in boardExperiments:
        distanceKm = geopy.distance.geodesic(
            (latitude, longitude),
            (exp.latitude, exp.longitude)
        ).km
        if distanceKm <= maxDistanceKm:
            nearbyBoards.append(exp)

    return nearbyBoards
```

# Chapter 6

# Evaluation

The aim of this section is to present the results obtained with the contribution given by this work.

The first paragraph is intended to show the general performance of the system and the various parts that were affected by the changes.

The second one focuses on the data transmission through the Bluetooth Low Energy technology and the performances given by the new communication protocol to collect and process the data.

Finally, the last paragraph is intended to highlight some open issues that have to be reviewed in order to start the experimentation phase.

## 6.1 Performances of the system

The contribution given by this work was intended to increase the performance of the whole system, in particular regarding the participative aspect of the project.

### 6.1.1 I/O and transmission results

One of the changes has involved the storage management on the IoT device, migrating from the CSV format to a binary representation of the data in the files in order to reduce their size and improve the I/O operations and the transmission time. Moreover, the temporal window of the data contained in a single file has been reduced from one day to one minute.

File system management has also been changed due to increased I/O operation time when the amount of files in the root folder becomes large. The solution implemented tried to fix the amount of files in a folder, limiting in turn read-and-write operation time. Indeed, file system structure has been

refactored into sub-folders, divided by date and time, limiting the amount of files for each folder to 60, one for each minute of the hour.

Moreover, a new communication protocol has been introduced in order to transmit the data from the monitoring device to the mobile application. Its aim was to improve the performance and reliability of the transmission.

Table 6.1 shows some of the results obtained with a comparison between the old version of the project and the new one and a focus on the increasing performance percentages in the last column. The results were derived from tests conducted using both textual measurement files, shown in the first column, and binary ones, shown in the second column. Both kinds of files cover a temporal window of one minute, containing the same amount of measurements that, as explained in section 6.1.2, is about 560. The values shown in the table were computed by averaging all the results obtained by the tests, indicating the mean values for a single file.

| | **Before work changes** | **After work changes** | **Improvement rate** |
|---|---|---|---|
| **File size** | ~13 KB | ~4.5 KB | 65 % |
| **I/O operations time** | $\geq 500ms$ | ~150 ms | $\geq 70\%$ |
| **Data transmission time** | ~4500 ms | ~500 ms | 88 % |

**Table 6.1:** Data storage and I/O operations statistics

### 6.1.2 Data loss

As shown in the previous sections, a signature process has been introduced on the IoT device in order to let the clients verify the integrity of the data. However, this process takes 2 seconds of computation, during which the device is unable to handle the interrupts coming from the sensor timers. Since they will be handled when the thread is free, signature process results in data loss every minute. Furthermore, since data sampling is different for each sensor, the amount of data lost during the process changes accordingly, resulting in a minimum and maximum percentage of data loss. In order to compute these values, the following table shows for each sensor how often it is sampled and how many measurements the sample contains:

| **Sensor type** | **Sampling time** | **N° of measurements produced for each sample** | **Measurements produced** |
|---|---|---|---|
| PM | 1 s | $2 * 4 = 8$ | $PM_{10}$, $PM_{2.5}$ |
| GPS | 3 s | 2 | latitude, longitude |
| DHT | 4 s | 2 | temperature, relative humidity |
| BPM | 5 s | 1 | pressure |

**Table 6.2:** Data sampling

Given the number of measurements produced in a sample ($m_s$), the sampling time ($t_s$) and the number of each type of sensor mounted on the board ($n_s$), it is possible to compute the maximum amount of measurements contained in a single file with a temporal window of one minute:

$$\begin{aligned} max &= \sum_s m_\mathrm{s} \frac{60}{t_\mathrm{s}} n_\mathrm{s} \\ &= 2\frac{60}{1}4 + 2\frac{60}{3}1 + 2\frac{60}{4}1 + 1\frac{60}{5}1 \\ &= 480 + 40 + 30 + 12 = \mathbf{562} \end{aligned} \tag{6.1}$$

Moreover, an hypothetical minimum value could be computed considering that the last interrupt of the current minute from sensor timers could be kept at the beginning of the next minute. In this way, the minimum value is represented by the maximum value minus the measurements of one sample:

$$\begin{aligned} min &= \sum_s (m_\mathrm{s} \frac{60}{t_\mathrm{s}} n_\mathrm{s} - m_\mathrm{s}) \\ &= (480 - 8) + (40 - 2) + (30 - 2) + (12 - 1) \\ &= 472 + 38 + 28 + 11 = \mathbf{549} \end{aligned} \tag{6.2}$$

In the end, data loss could be computed in the best and worst case for each sensor. The results, presented in table 6.3, show a data loss percentage from 1.4% to 3.8% every minute.

| | **Min** | **Max** |
|---|---|---|
| N° PM measurements lost in 2 seconds | 8 | 16 |
| N° GPS measurements lost in 2 seconds | 0 | 2 |
| N° DHT measurements lost in 2 seconds | 0 | 2 |
| N° BPM measurements lost in 2 seconds | 0 | 1 |
| N° measurements in a file of 1 minute | 549 | 562 |
| Data loss | $\frac{8}{562} \simeq \mathbf{1.4\ \%}$ | $\frac{21}{549} \simeq \mathbf{3.8\ \%}$ |

**Table 6.3:** Data loss statistics

### 6.1.3 API performance

When the files are transmitted from the monitoring device to the mobile application through BLE, they are in turn sent to the server through the provided REST API. Compared to the previous implementation of the API accepting a request body containing a single measurement, the current API is able to accept multiple files containing hundreds of measurements. Most of the time spent to receive a response from the server is due to network latency and to the HTTP protocol operations and not to the elaboration of the request itself. The implemented API is configured to receive a maximum of 20 files, 10 for the measurements and 10 for the signatures, because, as shown in table 6.4, in this way the server is able to respond in less than 1 second. In order to send X measurements to the server, the old API had to be invoked X times and the table shows an approximation of the old API response time to a linear function described by $Xt_0$, where $t_0$ is the response time for a single measurement upload. Even in the case of a single file upload, the performance improvement increased by almost 100%.

| **N° of measurements sent** | **Old API response time** | **New API response time** | **Improvement rate** |
|---|---|---|---|
| 1 | 32 ms | - | - |
| 200 (~1 file) | 6.4 s | 88 ms | 98.6 % |
| 400 (~2 file) | 12.8 s | 102 ms | 99.2 % |
| 600 (~3 file) | 19.2 s | 148 ms | 99.3 % |
| ... | ... | ... | ... |
| 2000 (~10 file) | 64.0 s | 986 ms | 98.5 % |

**Table 6.4:** Upload API statistics

## 6.2 Data transfer statistics

Many strategies of communication were explored in order to transmit the data through BLE from the monitoring device to the mobile application in the fastest way possible and the best one has been chosen through an empirical approach based on the measured performances. Moreover, at the beginning of this work, many files were corrupted during the BLE transmission and they were not understandable for the client. For this reason, signature process was introduced in order let facilitate the verification of the integrity of the file on the client. However, even if it results in performance degradation, the server should be aware of the success of the transmission in order to retransmit or delete a specific file.

One the variable that affects the performance is represented by the combination of commands provided by Bluetooth Low Energy technology. As described in section 3.2, they are the following: *write*, *read*, *notify/indicate*.

Another variable to take into consideration is represented by the buffer queue system which is involved in the data exchange protocol, storing files in RAM and avoiding as much as possible I/O operations with the SD during the transmission phase.

Table 6.5 shows the results in terms of data transmission time for each file of measurements along with the signature one for each combination of the solutions adopted. In order to send the data, the operation needed can be the read or the notify one. Two of the solutions use them in combination with the write operation which is used to acknowledge the server that the file has been received, while the other ones do not wait for a confirmation of receipt. While with the first solution (using the read and write operation) the client acknowledges every chunk of the file, in the solution with the notify and write approach, the client acknowledges only the whole file reception. The introduction of the combination of the two operations is to increase the reliability of the data transmission at the expense of the performance in order to let the monitoring device know about which file has been received and which should be sent again.

The results show that the read operation is in general much slower than the notify one and that acknowledging every single chunk of the file is too expensive from the performance point of view. The best solution should use the notify approach and, since acknowledging the file reception ensures more reliability, the combination with the write operation has been chosen for this project, even if it leads to an increment in transmission time of about 200 ms. Finally, all the results show that using the buffer queue system as an access point to the memory to have a cache of the files in RAM instead of the SD leads to a reduction of the overall transmission time of about 150 ms,

corresponding to the I/O time.

| BLE commands | Without buffer queue system | With buffer queue system |
|---|---|---|
| Read and Write | ~4900 ms | ~4700 ms |
| Read | ~1100 ms | ~950 ms |
| Notify and Write | ~650 ms | **~500 ms** |
| Notify | ~450 ms | ~300 ms |

**Table 6.5:** Data transmission time statistics

The results reported in the table were obtained with experiments carried out by placing the devices at close range, mainly less than two meters from each other. However, further experiments, carried out to verify how the attenuation of the Bluetooth signal impacted the transmission time, did not show evident differences. Indeed, the power of signal is no longer detectable at a distance of about 10 meters and in that case the connection is interrupted.

Moreover, the previous results do not take into account the time needed for establishing a BLE connection between the devices, which is independent from the command used in the communication and it is required only once at the beginning. Its mean value was measured to be around 2 seconds and was measured on the app, starting from the discovery of the board through the Bluetooth scan until the receipt of the first chunk of a file. This time, added to the time required to transfer all the chunks of a file through the notify and write commands, represents the minimum time in which a user should be near to the board in order to perform a complete transmission.

## 6.3 Open issues

The aim of this paragraph is to highlight the main issues that this work didn't address and that could be solved in future works on this project.

### 6.3.1 Data aggregation

One of the problems was encountered during the reorganization of the procedures on the server which are responsible for aggregating the measurement values, computing averaged values on different temporal windows. The problem is related to the logic that retrieves the measurements from the *measure_table*. In fact, when an aggregated measurement already exists for a specific sensor, the logic aggregates only the measurements arrived after that time arrival.

This implementation is a simplification that avoids recomputing the old aggregated values but introduces a strong limitation in the system: it assumes that all the measurements for a sensor arrive in the right chronological order. However, the monitoring device has been designed to send first the latest collected measurements because they have the highest priority and, if older measurements have not been already sent, they are no more considered by the server.

Actually, the procedures run on two background tasks of the web application and it could lead to performance issues when satisfying client requests. Since the operations of the tasks have to aggregate the data on the database, it could be useful to move the aggregation procedures directly to the database, implementing some scheduled *stored procedures* in order to optimize the web application job.

### 6.3.2 Data calibration

Another problem is related to data calibration. The use of tiny low-cost PM sensors leads to inaccurate measurements as compared to the data provided by the official environmental agencies. For this reason, the research group had already studied and developed a calibration technique using machine learning models. Initially, they analyzed both the Multivariate Linear Regression model and the Random Forest one using the data produced by ARPA stations in Turin as training. This work is part of a Python module that has been called *WSAnalysis*. During the development of the old server, it was integrated to calibrate the measurements but, with the new server development, it has not been integrated because the development is still in progress and data calibration was not in the scope of this work.

# Chapter 7

# Conclusions

The goal of the project described in this work is to explore and develop an environmental monitoring system to give citizens a tool to improve their health conditions and therefore their lives. The project wants to propose an alternative low-cost solution to the one provided by the official environmental agencies in order to collect data about the air quality conditions and in particular about Particulate Matter (PM) concentration. Given their considerable dimension and costs, the official monitoring stations cannot be located in the city center and they are not so representative of a specific area. With this solution, the monitoring devices can cover up multiple areas of the territory since they are cheaper and smaller and they allow them to take localized countermeasures.

In order to maintain the overall cost of the system low, some architectural choices have been taken. One of them is represented by the citizens' participation in the system itself: since equipping each monitoring device with an Internet connection would lead to an increase in costs, the collected data of the monitoring devices are transmitted to citizens' smartphones which are in turn responsible of sending them to a centralized remote server. Since the good functioning of the system depends on citizens' participation, the system has been called participative.

The participative approach of the system is highly based on an IoT infrastructure composed of monitoring devices. The transmission of the data to the citizens' smartphones, which run a mobile application that acts as a gateway to the server, is made through Bluetooth Low Energy technology.

The aim of this work was to contribute to the development of the system, in particular by taking care of solutions to improve and fill up some of the performance and reliability gaps during the transmission of the data of the previous prototype of the system. It was made through the implementation of optimized data management which was able to save 65% of the memory on the IoT device and a new data exchange protocol that improved the data transmission time by about 88%.

The results obtained could open up new opportunities for project development and improvement. In fact, with the current performances achieved, the monitoring devices could be installed in places where people use to stop even for a few minutes, such as bus stops. In this way, data collection could also be done by a single close person and the download of an entire hour of data could take place in just about 30 seconds.

However, the success of the project and therefore the good functioning of the system depend on the participation of the users. Indeed, encouraging to download the mobile application and involving users as much as possible is one of the most important next steps of the project.

Several technical experiments on the prototype will have to be conducted in a real context to verify its validity and identify any weaknesses. During this experimentation phase, it could be useful to conduct a series of interviews with a group of citizens, future users of the system, to also verify that their needs are still compatible with the functions offered by the system, or, instead, to discover new possible areas of improvement.

# Bibliography


[1] World Health Organization. URL: https://www.who.int/news-room/fact-sheets/detail/ambient-(outdoor)-air-quality-and-health (cit. on p. 1).

[2] Our World in Data. URL: https://ourworldindata.org/grapher/outdoor-pollution-deaths-1990-2017 (cit. on p. 1).

[3] Phys.org. URL: https://phys.org/news/2016-04-air-pollution.html (cit. on p. 2).

[4] U.S. Environmental Protection Agency. URL: https://www.epa.gov/pm-pollution/particulate-matter-pm-basics#PM (cit. on p. 2).

[5] New York State. URL: https://www.health.ny.gov/environmental/indoors/air/pmq_a.htm (cit. on p. 2).

[6] *EUR-Lex - 02008L0050-20150918 - EN - EUR-Lex*. URL: https://eur-lex.europa.eu/legal-content/IT/TXT/?uri=CELEX%3A02008L0050-20150918 (cit. on p. 2).

[7] *Report: A Guide for Local Authorities Purchasing Air Quality Monitoring Equipment - Defra*. URL: https://uk-air.defra.gov.uk/library/reports?report_id=386 (cit. on p. 2).

[8] Città Metropolitana di Torino. URL: http://www.cittametropolitana.torino.it/cms/ambiente/qualita-aria/rete-monitoraggio/stazioni-monitoraggio (cit. on p. 3).

[9] CRAN. URL: https://cran.r-project.org/web/packages/PWFSLSmoke/vignettes/NowCast.html (cit. on p. 3).

[10] PULSE. URL: http://www.project-pulse.eu/ (cit. on p. 4).

[11] URL: http://www.project-pulse.eu/wp-content/uploads/20190707_Empowering-Citizens-through-Perceptual-Sensing-of-Urban-Environmental-and-Health-Data-Following-a-Participative-Citizen-Science-Approach.pdf (cit. on p. 4).

[12] Gabriele Telesca. «Study and development of a participative air pollution monitoring system». MA thesis. Politecnico di Torino, 2021 (cit. on pp. 4, 14, 31).

[13] Nordic Semiconductor. URL: https://blog.nordicsemi.com/getconnected/iot-wireless-architecture (cit. on p. 8).

[14] Pycom. URL: https://pycom.io/product/expansion-board-3-0/ (cit. on p. 9).

[15] Pycom. URL: https://pycom.io/product/fipy/ (cit. on p. 9).

[16] B. Montrucchio, E. Giusto, M. G. Vakili, S. Quer, R. Ferrero, and C. Fornaro. «A Densely-Deployed, High Sampling Rate, OpenSource Air Pollution Monitoring WSN.» In: *IEEE Transactions on Vehicular Technology* (2020) (cit. on p. 11).

[17] ZPE. URL: https://www.zpesystems.com/centralized-vs-distributed-network-management-zs/ (cit. on p. 12).

[18] Bluetooth. URL: https://www.bluetooth.com/learn-about-bluetooth/topology-options/ (cit. on pp. 12, 13).

[19] Atlas RFID Store. URL: https://www.atlasrfidstore.com/rfid-insider/are-bluetooth-bluetooth-low-energy-ble-forms-of-active-rfid/ (cit. on p. 13).

[20] LEVEREGE. URL: https://www.leverege.com/iot-ebook/iot-wifi (cit. on p. 13).

[21] Amir Nagah Elghonaimy. «LoRaWAN for Air Quality Monitoring System». MA thesis. Politecnico di Torino, 2021 (cit. on p. 14).

[22] HIVE MQ. URL: https://www.hivemq.com/blog/lorawan-and-mqtt-integrations-for-iot-applications-design/ (cit. on p. 14).

[23] RedHat. URL: https://www.redhat.com/it/topics/internet-of-things/what-is-iot (cit. on p. 15).

[24] CyberAngel. URL: https://cybelangel.com/digital-signatures-are-the-cybersecurity-vulnerability-you-need-to-stop-ignoring/ (cit. on p. 20).

[25] Bluetooth. URL: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/ (cit. on p. 23).

[26] Bosch. URL: https://developer.bosch.com/products-and-services/sdks/xdk/develop/c/connectivity/bluetooth-low-energy-ble (cit. on p. 25).

[27] PunchThrough. URL: https://punchthrough.com/maximizing-ble-throughput-part-2-use-larger-att-mtu-2/ (cit. on p. 24).

[28] HTML. URL: https://www.html.it/pag/377313/il-framework-dettagli-tecnici/ (cit. on p. 34).

[29] Flutter. URL: https://flutter.dev/docs/resources/architectural-overview (cit. on p. 35).

[30] Wajahat Karim. URL: https://wajahatkarim.com/2019/11/how-is-flutter-different-from-native-web-view-and-other-cross-platform-frameworks/ (cit. on pp. 36, 37).

[31] Auth0. URL: https://auth0.com/blog/sqlalchemy-orm-tutorial-for-python-developers/ (cit. on p. 49).