UCLouvain

École polytechnique de Louvain

Using LoRaWAN and Wi-Fi for smart city monitoring in Louvain-la-Neuve

Authors: **François De Keersmaeker, Nicolas van de Walle**
Supervisors: **Yves Deville, Ramin Sadre**
Readers: **Yves Deville, Sébastien Lugan, Lionel Metongnon, Ramin Sadre**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Acknowledgements

First, we would like to thank our supervisors, Prof. Yves Deville and Ramin Sadre, for having supervised and helped us throughout the year, and for their relevant comments during the proof-reading of the thesis. We also thank the readers, Sébastien Lugan and Lionel Metongnon, for having accepted the task of reading and grading our master's thesis.

We would like to thank our friends who allowed us to use their balconies for the deployment of our system: Lola and Laureline, and Matis and Martin.

Finally, we would like to thank our parents, who allowed us to pursue our studies, that come to an end with this work.

Contents

Chapter 1

Introduction

Current computing technologies, ranging from small and everyday connected objects to enormous centers containing thousands of servers, are continuously growing in popularity and usage. On the one hand, the Internet of Things (IoT) allows every object to become "smart", by giving it autonomy and a network connection. On the other hand, any user can leverage the power of the cloud to meet the needs of any of their applications. Combining these two concepts offers seemingly unlimited possibilities.

One environment that greatly benefits from those technologies are urban areas. Multiple initiatives have been launched around the globe, to transform cities into digital, or "smart", cities [1], by deploying various applications based on the Internet of Things and cloud computing, with the common goal of improving the everyday life of the citizens, and facilitate the management and monitoring of the city.

Nevertheless, every municipality does not have the same requirements. Some want to optimize their traffic, others to monitor environmental data or even add a layer of security for the inhabitants by deploying security cameras. Those requirements will, in practice, influence the choice of the technology and the networks used for communication. With this master's thesis, we will apply the concepts of a smart city to Louvain-la-Neuve, by developing and evaluating a smart city application for this city, namely the smart monitoring of environmental data.

1.1 Goal of the thesis

This thesis focuses on Louvain-la-Neuve and its environment. The goal is to show how the LoRaWAN network, combined with the traditional Wi-Fi network, can be used for smart city monitoring in this city, by deploying an end-to-end system

allowing the monitoring of environmental data such as the temperature, humidity, air quality or noise level. This goal was chosen by analyzing the situation and population of the city: it is mainly a pedestrian city, the city center being restricted for cars, that is home to a lot of students. To this end, sensor devices have been deployed in various locations of the city, to collect data, which are then gathered in one place for monitoring.

This system comes with several requirements. The first one is that the devices should be standard and off-the-shelf, to minimize the overall cost and facilitate their configuration. Specific, ultra-low-power sensor devices exist that would be more efficient, but they would be more costly and difficult to use, and their deployment would become expensive as multiple devices will be used, which is an issue since this is a research project and not a production-level system. Moreover, this accessibility involves that any other citizen could participate to the network with their own devices, and make the network grow.

Other requirements are that the devices should be easy to install in one location, and left functioning in autonomy. To ease this installation, the devices must be powered on with batteries, and use wireless networks for data transmission. We chose to use the LoRaWAN [2] network, which is a low-power network specifically designed for power-constrained devices, and which can be freely used by any user for its own application, provided the network restrictions are respected. For reliability reasons, an objective for the system is the use of hybrid networks interfaces, by being able to transmit data using LoRaWAN or Wi-Fi, depending on the context: in general, the low-power LoRaWAN network is used; if the data is urgent, i.e. if it has exceeded a threshold, the power-hungry but more reliable Wi-Fi network is used. Furthermore, the additional infrastructure needed for our solution to function, i.e. new gateways or access points, is required to be minimal, to allow our system to be installed easily. Those two requirements also enable the system to be easily extended by new users.

The devices have been deployed across the city and gather frequent data, allowing the detection of small variations. The objective of the system is to be able to monitor those data on a dashboard. The dashboard has been made publicly accessible online to anyone, such that any citizen can benefit from the collected data. The data is displayed in an understandable and user-friendly way, by using graphs, maps, gauges, etc.

As the devices are powered with batteries, the main challenge is to minimize their energy consumption, such that they can be left in autonomy for the longest

possible time before having to recharge them. This involves many factors: the choice of a low-power network, a light but comprehensive data transmission protocol, and the most energy efficient configuration. To ensure this, their energy consumption has been evaluated, by extensive measurements in multiple different configurations. The most energy efficient configuration has then been applied to the devices, and their battery life has been estimated. Alternatively, a battery monitoring system has been attached to the devices, such that their battery can be monitored along with all the other environmental data they retrieve.

Beyond the developed system, the goal of this thesis is to understand what are the different components of a smart city application, and to select the most suited parameters depending on the environment. To conclude this thesis, we discuss the advantages and drawbacks of our system, and explore possible improvements such that it can lean towards a real, production-level system.

1.2 Structure

This report is organized following the structure outlined hereafter.

Chapter 2: Theoretical framework A definition of what the Internet of Things (IoT) and a smart city are will first be given, along with examples of smart city applications. Afterwards, Low-Power Wide-Area Networks, which are networks specifically intended for power-constrained IoT devices, and LoRaWAN in particular, will be discussed. The theoretical aspects and regulations of this network will be detailed, followed by a presentation of The Things Network, which is the backbone network used to receive and forward LoRaWAN transmissions. Finally, the technologies used to handle, store and monitor the collected data will be listed and discussed. Related work will also be presented for reference.

Chapter 3: Technical description of tour solution The motivation underlying our application will first be given, taking the environment of Louvain-la-Neuve into account. The big picture of the end-to-end solution will be presented, then each step will be detailed, following the path taken by the data, from the end devices to the backbone network, to the monitoring platform. The hardware, software and protocol used for the data transmission to the backend are first detailed, before explaining how these data are handled by servers to be displayed on a monitoring interface.

Chapter 4: Experiments To design and evaluate the system, and the assumptions we made while designing it, we carried out numerous experiments, which

will be described in that section. Those experiments consist of: a first prototype of our system, the characterization of the system to monitor the battery of the end devices, extensive energy consumption measurements to estimate the most energy efficient parameters and evaluate the battery life of the end devices, a brief discussion of transmission reliability, and the real-world deployment of our system in the city of Louvain-la-Neuve.

Chapter 5: Discussion and improvements In this last section before the conclusion, the advantages and drawbacks of our system will be discussed, on several aspects such as security and scaling, and potential further improvements will be presented, related to further battery optimization for the end devices, and old data management, among others. As our system involves various components, i.e. microcontrollers, sensors, and cloud facilities, their cost will also be discussed.

Chapter 2

Theoretical framework

In this second chapter, we will present the theoretical framework underlying our system, i.e. a description of smart cities, as well as the technologies and tools that we have selected to deploy our system into the city of Louvain-la-Neuve. Related work will also be presented for reference.

2.1 Smart Cities

The term "smart city" has become trending in the recent years, in parallel with the outbreak of Internet of Things devices in multiple sectors, i.e. personal (smart home), industrial (smart factory appliances), and in cities, as it will be seen in this section. The concept of the Internet of Things will first be explained, then some of its uses in smart cities will be described, along with application examples.

2.1.1 Internet of Things

The "Internet of Things" (IoT) refers to the concept of having all sorts of so-called "smart" or connected objects, or "things", in diverse environments, to help their users by collecting data over their own operation or by enabling them to be more autonomous or remotely controllable [3,4]. The denomination "smart" is quite wide, and can encompass a variety of devices, from small connected sensor devices, e.g. a connected thermometer, to everyday house appliances, e.g. a light bulb, a car, or a fridge, to which new features have been added thanks to digital controllers. Generally, a device is considered "smart" when it provides some sort of autonomous behaviour, along with the following features:

- Sensors are attached to the device, such that it can collect data about its current state and its surroundings. For a smart fridge, the most obvious would be thermometers, to measure the inside and outside temperatures.

- A network connection is enabled on the device, often wirelessly, to provide interaction with the users, e.g. remote control of the device, or communication with another device (machine-to-machine communication), e.g. transmission of measured values, to enable further data processing.

The IoT is a recent concept, but that underwent a tremendous blow-up in the last years. The number of connected objects was estimated to 30 billion in 2020, with a market valued at 248 billion dollars. It is predicted to grow up to 75 billion devices, and 1.6 trillion dollars in 2025 [5].

2.1.2 Smart city applications

One possible use of IoT devices is to provide new capabilities to cities, to form what is called "smart cities" [6]. In general, the name "smart city" is given to any city in which the technology of IoT has been leveraged to provide new applications, e.g. for monitoring by the authority, to help inhabitants with their daily life, or to entice tourists with interactive sites [7]. The best way to present smart cities, and their wide array of possibilities, is to describe some examples of applications:

- Environment monitoring: allows the monitoring of different environmental data across the city (air quality, temperature, pressure, etc.). The state of Singapore has implemented such a system in the city with the "Smart Nation" initiative, where IoT sensors installed all around the city collect real-time data that is openly available for the citizens [8]. This is the application we implemented into the city of Louvain-la-Neuve with this thesis.
- Smart waste bins: uses the IoT to measure the fill-level of the waste bins of the city to only empty them at the right time, saving money and manpower, and contributing to the cleanliness of the city. Such solutions for smart monitoring of waste bins have been proposed by Folianto *et al.* [9] and by Aazam *et al.* [10].
- Smart parking: monitors the parking places of the city, allowing to have an overview of the available spaces across the city, and possibly to book parking places using a specific application. Khanna and Anand [11] have proposed such a system. The city of Copenhagen, Denmark, implemented a system for smart parking, than can predict with a good accuracy which parking places will be free, based on old and real-time data [12].
- Smart traffic: such systems have for aim to better handle the traffic in smart cities. This can take several forms: Rizwan *et al.* [13] designed a system to monitor the traffic in real time and indicating it to the drivers; Kanungo *et al.* [14] proposed a system to dynamically adapt traffic lights, by leveraging video monitoring systems installed at crossroads, with the future aim of interconnecting different crossroads for even better traffic management; Gazal

et al. [15] proposed a similar system, using microcontrollers with infra-red sensors, with the additional intent of helping emergency vehicles stuck in traffic. The city of New York implemented a system for smart traffic, that they claim to have improved traffic speed by 10% [16].

- Smart cars: related to smart traffic are the smart cars, that will, in a near future, be able to communicate with other cars and with traffic appliances, to become self-driving [17]. While smart cars themselves are not properly speaking a smart city applications, they are certainly related to it, as they are present in a city ecosystem, and can communicate with smart city applications, e.g. smart traffic lights.
- Smart tourism: IoT technologies can be used to ease and improve tourism, e.g. by using Near Field Communication (NFC) [18] devices that a tourist can simply connect to with their smartphone to obtain more information about a touristic site. Borrego-Jaraba *et al.* [19] proposed such a system, by using smart NFC posters, containing mobile phones, that give information about the city interest points when scanned. Alternatively, Basili *et al.* [20] designed a mobile application intended for smart tourists, used, among others, to scan those NFC posters and make payments with NFC. Boes *et al.* [21] showed that tourists were enthusiastic to the idea of using NFC posters for smart tourism.

2.2 Low-Power Wide-Area Networks

For IoT applications that use resource-constrained devices, e.g. devices that run on battery, that must communicate over a long range, regular networks like Wi-Fi are not suited. In this case, a solution is to use a Low-Power Wide-Area Network, or LP-WAN. An LP-WAN is a wireless network specifically designed to be used by resource-constrained devices, such as IoT devices, in wide environments like smart cities [22]. LP-WANs are usually compared to more classical wireless networks, such as Wi-Fi, on 3 aspects, which are the communication range, the power consumption of devices using this method of communication, and the data rate. Those aspects are summarized in Table 2.1.

	LP-WAN	Wi-Fi
Range	1 km - 40 km	15 m - 100 m
Power consumption	Low	High
Data rate	100 bps - 200 kbps	Over 1 Mbps

Table 2.1: Comparison between LP-WANs and Wi-Fi [23, 24]

The most suited network for device communication therefore depends on the application. For applications that necessitate a long communication range, where devices must consume as less power as possible, e.g. to keep battery-powered devices running for a long time, and where a low data rate is not an issue, LP-WANs are the best choice over Wi-Fi.

One of the most prominent LP-WANs for IoT applications is the LoRaWAN network [25], which is the network we chose for our system, and which will be described in detail in the remaining of this section. Afterwards, two other popular LP-WANs, Sigfox and NB-IoT, will be presented, and compared with LoRaWAN.

2.2.1 LoRaWAN

The LoRaWAN protocol is a Low-Power Wide-Area Network intended to be used for IoT applications, that is maintained by the LoRa alliance [26], an open and non-profit association composed of more than 500 companies handling its worldwide development. In the OSI network layers model [27], shown in Figure 2.1, it is located at level 2 [2], the data-link layer. This protocol is open, which means that any user can implement it on its devices, and is able to use it for wireless transmission. However, the radio interfaces needed to implement the LoRaWAN protocol, i.e. the level 1 network layer, called the LoRa physical layer, are a property of Semtech [28, 29], which means that a user must purchase Semtech's chips, or devices that contain a Semtech chip, to be able to use LoRaWAN.

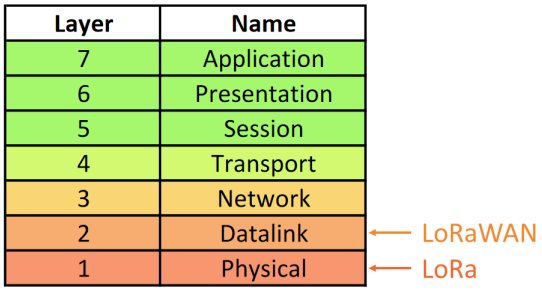


Figure 2.1: OSI model of network layers, with LoRa and LoRaWAN

Structure of the network

When using a LoRa-enabled device and LoRaWAN as data-link layer, the interaction between the end devices and the Internet involves a specific network structure [2, 30], which is depicted on Figure 2.2.

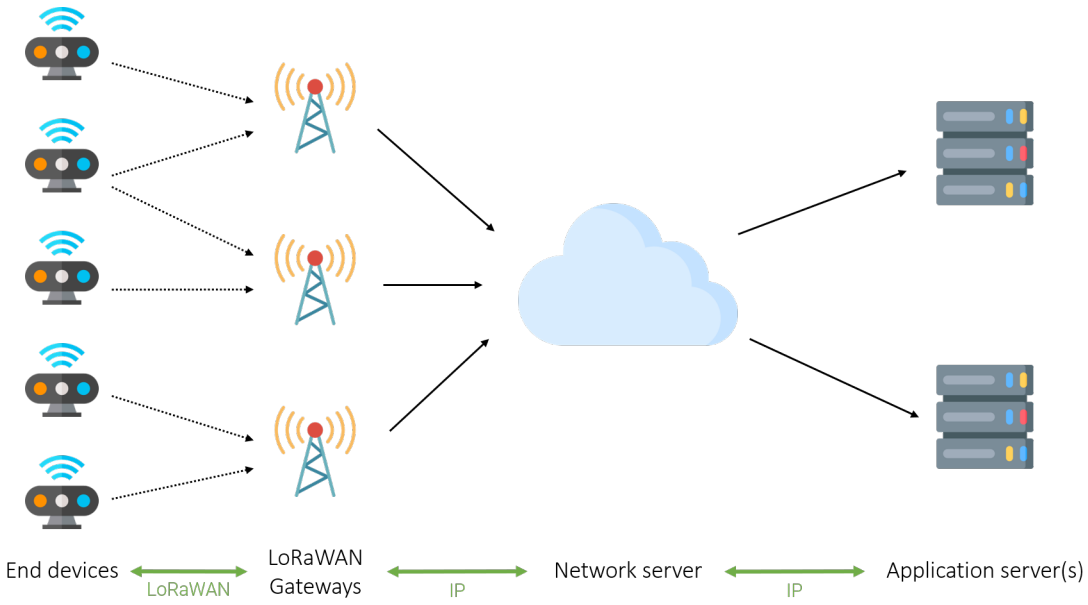


Figure 2.2: LoRaWAN network structure (Icons from `https://flaticon.com`)

The transmission of uplink (from the end devices to the network) LoRaWAN packets takes 4 steps:

1. The end devices send packets, using the LoRa physical layer and the LoRaWAN data-link layer.
2. LoRaWAN gateways, which are devices specifically configured to receive packets from end devices and forwarding them, receive those packets. If they are all in the sending range of the end device, multiple gateways might receive the same packet. They forward all the received packets to the network server, using a traditional IP backhaul network, e.g. wired Ethernet or Wi-Fi.
3. The network server is also specifically intended to handle LoRaWAN traffic. It receives the packets from all the gateways, handles the duplicates ones, and forward them to the application servers specified by the user. Duplicate packets can be interesting for their metadata, i.e. signal strength and air-time, but only one of them must be forwarded. To ensure this, the MD5 hash of the packets' payloads is checked by the network server [31]. An example of a network platform offering this solution is The Things Network, which will be described in more detail in section 2.3. This stage is only intended to forward LoRaWAN traffic to the application servers, and cannot process the transmitted information by itself.
4. The application servers receive the packets from the network server for

further processing. What is done with those packets is application specific, e.g. display the data, send alerts, activate actuators, store the data for later data analysis, etc. This stage can be implemented by using cloud functions with any cloud provider, which will be detailed in section 2.4.

Downlink transmission (from the network to the end devices) is also possible, but is less frequent. In that case, the aforementioned steps are simply taken in the reverse order.

Frequency bands and regulations

LoRaWAN uses free ISM (Industrial, Scientific and Medical) frequency bands, so as to be open and usable by every user without having to pay for a license. In Europe, the unlicensed ISM frequency bands are the 433 MHz and the 868 MHz bands [29, 32], the latter being the most used in the case of LoRaWAN communication.

As potentially anyone can use these unlicensed bands for their own purpose, standards have been established for the sharing of these bands, by regulating the duty cycle of any device using them, to avoid flooding them. In Europe, the duty cycle of any device using the ISM bands is limited to 1% [33], which means that the device may not transmit more than 1% of the time. For example, if a device takes 500 ms to send a message over LoRaWAN, it may not send more frequently than once every 50 seconds.

For similar reasons, LoRaWAN also limits the payload size of messages [32], which can not exceed 250 bytes. However, as metadata must be inserted in the payload to comply with LoRa specifications, the actual application payload size cannot exceed 222 bytes. In practice, the payload size limit depends on the data rate, as explained in more detail in section 2.2.1.

Classes of devices

The protocol accommodates three classes of devices [2, 34], which have different usage policies and can be selected depending on the application. Roughly speaking, the classes offer different three different levels of trade-off between low power consumption and low latency with respect to downlink messages. Those are the following:

- Class A: lowest power consumption, and highest latency. Devices of this class can transmit uplink message mostly whenever they want (without exceeding the duty cycle), but can receive downlink messages only during two short

windows following each uplink transmission. This class is intended for battery-powered sensors, for which a long battery life is important, but downlink latency is not an issue, or for devices that do not need downlink messages.
- Class B: balance between power consumption and latency. Uplink transmission is identical to class A devices. The reception of downlink messages can occur during the two windows following an uplink message as class A devices, as well as during extra scheduled windows, which will be opened upon reception of a beacon from a LoRaWAN gateway. This class is intended for battery-powered actuators, that still need an acceptable battery life as well as a not too high downlink latency.
- Class C: lowest latency, highest power consumption. Uplink transmission is identical as classes A and B, and downlink reception can occur at any time, except during an uplink transmission. This class is intended for actuators that are not powered by batteries, for which power consumption is not an issue, but must have a low downlink latency.

As each class is a strict improvement, regarding downlink latency, above the previous one, all devices must at least accommodate class A to be compliant with LoRaWAN specifications, and thus to be part of a LoRaWAN network.

Data Rate & Spreading Factor

LoRa-enabled devices can be configured to transmit data using seven different Data Rates (DR) [32]. Data Rates provide a compromise between fast data transmission, and resilience to interferences and errors, which increases the range of the transmitted messages. They go from DR6, which is the fastest but least resilient, to DR0, which is the slowest but most resilient one. In practice, one DR specifies the bandwidth used for transmission, and the Spreading Factor (SF) used to encode the message. In short, the Spreading Factor modifies the modulation used to encode the data on the carrier waves. The higher the SF, the more space the data takes on the wave, which makes it more resilient to interferences, but also takes more time to transmit. As a consequence, a lower DR, i.e. a higher SF, involves a higher energy consumption, as the transmission takes more time. Different SFs are orthogonal with each other, meaning interferences between signals encoded with different SFs cannot occur. More information on the Spreading Factor can be found in [35]. To restrict the air-time of LoRa-encoded packets, the payload size is limited, depending on the DR used; the lower the DR, the lower the maximum payload size. Table 2.2 shows the parameters for each of the Data Rates.

Data Rate	Bandwidth [kHz]	Spreading Factor	Max app payload size [bytes]
DR6	250	SF7	222
DR5	125	SF7	222
DR4	125	SF8	222
DR3	125	SF9	115
DR2	125	SF10	51
DR1	125	SF11	51
DR0	125	SF12	51

Table 2.2: LoRa parameters, depending on the Data Rate

Adaptive Data Rate

In some cases, the statically configured data rate of a device is not the most optimal for the device's behaviour or position. For instance, a device that uses DR0 and is located close to a gateway is wasting air-time and power, as it does not need a high spreading factor to reach the gateway without errors. To settle this issue, a mechanism called Adaptive Data Rate (ADR) can be enabled by LoRa devices. This mechanism allows the device to dynamically update its data rate, based on the quality of the link between the device and the nearest gateway [36, 37], as follows. The network server gathers measurements, e.g. the data rate, air-time, and signal-to-noise ratio, from the uplink packets. When it has collected enough measurements, it can schedule a downlink ADR request asking the device to adapt its data rate, to optimize the usage of the wireless link. This request is usually included into an application downlink packet or acknowledgement. If the application never sends downlink packets, an end device can also ask explicitly for a downlink ADR request, which will be sent as a standalone packet, by setting the `ADRAckReq` bit in an uplink packet [37].

ADR can simply be enabled by setting the `ADR` bit in the uplink packets. It should only be used when the link conditions are stable, i.e. when the device is static, because increasing the data rate means decreasing the range of the device, which can become a problem for moving devices, if they move too far from the gateway.

2.2.2 Other LP-WANs

LoRaWAN is not the only LP-WAN present on the market. In this section, we will review two other popular protocols, namely Sigfox and NB-IoT, and present their differences with the LoRaWAN protocol, as well as the arguments for the choice of LoRaWAN for our application.

Sigfox

Sigfox is an LP-WAN which shares some characteristics with LoRaWAN. It has the same star network structure, shown on Figure 2.2 for LoRaWAN, with end devices wirelessly transmitting data that is received by gateways, which then forward it over an IP backhaul to the network server, and finally to the external application server [38, 39]. It also uses the same ISM frequency bands, i.e. 433 MHz and 838 MHz in Europe. The main technical difference between the two resides in the bandwidth used for transmission: whereas LoRa modulation uses 125 or 250 kHz bandwidth, Sigfox uses a bandwidth as small as 100 Hz [22, 24, 38]. This induces the following characteristics:

- The data rate cannot exceed 100 bps, so the packets use a lot of air-time.
- As such, the number of messages is limited to 140 uplink, and 4 downlink, per day.
- The payload size is also limited to 12 bytes for uplink messages and 8 bytes for downlink messages, to not monopolize the network.
- Since the bandwidth is very small, the available frequency bands can be divided into a lot of different channels, e.g. 400 on the 838 MHz European band, which are all orthogonal to each other. Nearly all these channels can be used for simultaneous transmission.
- This results in a natively good resilience against interferences, without needing spreading factors like LoRaWAN. As such, the range of Sigfox is increased, as measured by Lauridsen *et al.* [40].

Regarding power consumption, Sigfox devices are similar to class A LoRa devices [24]. They are in sleep mode most of the time, and are woken up when data must be sent. Sigfox devices can only receive downlink data directly after an uplink transmission, which leads to a high downlink delay. Combined with the limited number of downlink messages and thus of acknowledgements, this involves a quite poor quality of service, that is partly solved by multiple uplink transmissions for each message.

On an economical point of view, Sigfox uses a somewhat inverse model as LoRa [41]. The LoRa radio interfaces can only be manufactured by Semtech, but the network is open and every developer can use it freely, provided they purchases the LoRa radio interfaces. On the contrary, Sigfox licenses its radio interfaces to several different manufacturers, which means that a developer has the ability to choose between different manufacturers to purchase hardware. However, the network is not open, and a user must pay a subscription to be able to use it, or use devices that come along a prepaid Sigfox subscription. The subscription is provided by Sigfox itself, thanks to partnerships with network providers in every

active country (e.g. Engie M2M in Belgium), and starts at 6€ per device per year [42]. As such, it is more expensive to develop research or proof-of-concept IoT applications using Sigfox.

Along with the Sigfox network subscription, comes an access to the Sigfox Cloud platform [43], which is the platform that enables the monitoring of Sigfox transmissions coming in and out of the Sigfox network server, similarly to The Things Network for LoRaWAN, which will be described in detail in section 2.3. Contrarily to LoRaWAN, and accordingly to the Sigfox business model, only the Sigfox network server can be used for Sigfox transmission, which means that Sigfox Cloud is the only possible platform to monitor it. To forward the data to the application server, there are two possibilities:

- Callbacks: the devices are configured on Sigfox Cloud such that, when a message is received from a device, the corresponding task is automatically carried out. The tasks linked to the reception of a message can be to forward to an external cloud platform, to send a downlink message, etc.
- REST API: the Sigfox Cloud platform provides a REST API that can be accessed by any external application. When using this possibility, a task is not carried out automatically when a message is received, but the messages are stored for three days in Sigfox Cloud. An external platform can then use HTTPS requests to retrieve the messages and information about the devices, and further process the data.

NB-IoT

NB-IoT (Narrow Band IoT) is an LP-WAN that stands out from the two previous ones, as it puts the accent on performance and quality of service, at the expense of battery life and range. It uses the same frequency bands as 3G and 4G, i.e. the 700, 800 and 900 MHz bands, with a 200 or 180 kHz transmission bandwidth, depending on the operation mode [44]. In practice, NB-IoT uses the actual LTE (4G) network, and optimizes its features for IoT applications such that battery consumption can be kept to a minimum. More precisely, the LTE messages, that contain metadata about the channel quality and carrier usage, that every device in a cell periodically broadcasts to every other device in the cell, are limited to a minimum, in size as well as in frequency.

The fact that the frequency bands are licensed and not used by other networks enables a high performance and quality of service [45], as interferences are way less likely. As such, the data rate can go up to 200 kbps for uplink messages, and 20 kbps for downlink messages [24]. The payload size can also be bigger than with the

two previous LP-WANs, its maximum being 1600 bytes. The communication range of NB-IoT between devices and gateways is lower than the other two LP-WANs, but this is balanced by the fact that it uses the same antennas as LTE, which cover, depending on the country, the whole territory. NB-IoT devices are also in sleep mode most of the time, but their battery life is usually lower, due to their compliance with the periodic LTE metadata messages explained above, and their physical layer that is more power-hungry [24, 45].

As NB-IoT uses licensed frequency bands, a user must pay to receive the authorization to use them for its application. As licensed bands are as expensive as 500 million euro per MHz, they are reserved to telecommunication companies. The business model to use this technology is therefore similar to mobile phones, by using SIM cards and paying a subscription to a network operator, e.g. Orange [46, 47]. Different subscription plans can be possible, depending on the data rate and number of messages. As such, the cost of using this technology can become high, if a big number of end devices are used, but the enhanced performance and quality of service justify this higher cost.

Since NB-IoT devices are used in conjunction with a network operator, the possibilities for further data processing depend on the operator. For instance, Orange proposes a free prototyping platform, Orange Maker [48, 49], which allows to monitor the data transmitted by the end devices, translate them and store them, such that they can be accessed by external applications via different APIs, namely a REST API, similarly to Sigfox Cloud.

Comparison and choice

Table 2.3 shows a comparison between the three described LP-WANs: LoRaWAN, Sigfox, and NB-IoT. With insight about the characteristics of all three possible LP-WANs, we can argue our choice to use LoRaWAN for our application.

	LoRaWAN	Sigfox	NB-IoT
Frequency band	Unlicensed ISM	Unlicensed ISM	Licensed LTE
Bandwidth [kHz]	125 / 250	0.1	200
Data rate [kbps]	0.3 - 50	0.1	200
Message limit	1% duty cycle	140 up, 4 down	Unlimited
Max payload size [bytes]	222	12 up, 8 down	1600
Range [km]	5 (urban), 20 (rural)	10 (urban), 40 (rural)	1 (urban), 10 (rural)

Table 2.3: Comparison between the 3 LP-WANs [24]

Sigfox has a better range and resilience, and a potentially lower battery consumption due to its lower bandwidth and payload size. However, its payload size is way too short for our application: we could only put three 32-bit values per message, which is too few if we want to monitor the city's environmental data.

NB-IoT, on the contrary, provides more quality of service than we need. Indeed, in our case, packet losses are not such a big deal, for two reasons: if the data is not urgent, i.e. it is not an emergency case, a following packet, containing similar data, will be transmitted shortly afterwards; if it is urgent, the packet will not be transmitted using an LP-WAN, but using Wi-Fi, which is more reliable. Furthermore, the payload size of 1600 bytes is greater than we need for our application. As such, we are not willing to pay for the benefits provided by NB-IoT. Besides that, the battery life of NB-IoT devices is lower than with the other LP-WANs, and we want to optimize it for our application, to minimize devices maintenance costs.

The biggest argument in favour of LoRaWAN concerns its business model. With LoRaWAN, we only have to purchase LoRa-enabled devices, then we can use the network freely. This is not the case with the other 2: with Sigfox, we would have to buy devices as well as a subscription to use the network, and with NB-IoT we would have to use different SIM cards for each one of the end devices, as well as a subscription to a network operator. The open aspect of the LoRaWAN network is very important in a project like this master's thesis, as well as in most research projects, as it eases the deployment of the application by only having to pay for the devices. Furthermore, it allows any other user to extend our network with its own devices, without having to pay for the network connection.

2.3 The Things Network

In the LoRaWAN network structure presented on Figure 2.2, we observe that the third step during a transmission is the "network server". The role of this step is to receive the packets forwarded by the LoRaWAN gateways, to collect them all in one place, dropping duplicates from multiple gateways, and forward them again to the last step, the user application servers. The user can either build their own service to undertake these tasks, or rely on existing commercial solutions. For this thesis, we have chosen to use the free solution The Things Network [50] (hereafter referred to as TTN), which will be described in this section.

2.3.1 TTN gateways

The Things Network is an easy to use and free network service intended to be used for any kind of LoRaWAN application. As such, any user can connect LoRaWAN devices to TTN, and start developing its own application. As explained before, end devices interact with gateways to transmit data; those gateways are linked to TTN servers over the Internet [51]. There are around 20,000 active gateways all around the world, and, while they are active, any user can use any gateway for its application. For example, there are three fixed gateways, belonging to UCLouvain and installed on the university buildings, always active in the city, so we can use them for our application. If there are no active gateways in the surroundings of the end devices, a user can also build and deploy its own gateway. They must then register it to The Things Network to use it for its application, which is straightforward and only requires a gateway ID, generated upon gateway configuration. In this case, it becomes and active gateway, and can also be used by any other user.

2.3.2 LoRaWAN network server features

This section will present the functionalities that any LoRaWAN network server, which includes The Things Network, must provide. Those features are mainly related to the security of the network, by providing authentication and encryption.

Identification and authentication of end devices

Since any TTN gateway can receive and forward any LoRaWAN payload, authentication is needed to ensure only intended devices can send packets to an application. For that, each end device is associated with two values, the `DevAddr` (Device Address) and the `NwkSKey` (Network Session Key) [25, 52], that are valid for the whole session, i.e. from device start-up to shut-down. The `DevAddr` is a 32-bit address used to identify the device inside the network. The `NwkSKey` is a 128-bit key used to sign messages, by using AES-CMAC authentication. The signature is then checked by the server, which uses the same key; if the signature is not valid, it means that the message has been sent by a non-authorized device, it is thus dropped.

Encryption

Besides end device authentication, The Things Network provides encryption from the end devices to the external application, split in two parts: encryption of the LoRaWAN payload between end devices and TTN, then encryption of the HTTPS request between TTN and the application server. In our case, the encryption is

not fully end-to-end, because the LoRaWAN payload has to be decrypted to be decoded by the TTN payload decoder, described in section 2.3.3. Both encryptions use a 128-bit key, the `AppSKey` (Application Server Key): the former uses AES to encrypt the LoRaWAN payload sent by end devices with the key [25, 52], and the latter is a generic HTTPS encryption, using TLS with the `AppSKey`. As the `DevAddr` and the `NwkSKey`, the `AppSKey` is valid for a whole session.

Device activation

The generation of the aforementioned address and keys depends on the procedure of device activation, which configures the necessary values for an end device to be able to transmit data to the server. Two alternatives are possible: Activation By Personalization (ABP) or Over-The-Air Activation (OTAA) [25, 53].

With ABP, the necessary session values are directly hard-coded into the device and the server. The advantage is that, unlike OTAA, the device does not have to follow a join procedure to negotiate the keys with the server, and can start to send LoRaWAN payloads right away, by specifying and using the hard-coded keys for its transmissions. This saves a small amount of time and bandwidth. However, this alternative is less secure: if an attacker manages to retrieve the keys, the only way to change them is to manually update the configuration of all devices and of the TTN application, which is the online tool to manage the end devices configuration, described in section 2.3.3.

With OTAA, the `DevAddr`, `NwkSKey` and `AppSKey` are not hard-coded, but are dynamically generated for each session. The `DevAddr` is generated based on the device's `DevEUI`, which is simply a unique 64-bit address for the device, that can be provided by the manufacturer, or generated by TTN. The `NwkSKey` and `AppSKey` are based on the `AppKey`, a 128-bit key hard-coded in the end devices and TTN applications, which is unique for each pair of end device with an application. To negotiate those keys, every OTAA end device must send a join request to the LoRaWAN network, containing this key, before being able to transmit payloads. To identify the TTN application that must process the join request from an end device, another 64-bit address is used, the `AppEUI`. If the request was successful, the server responds to the end device with the newly created session keys. In practice, this procedure introduces an overhead when the device starts, but makes the transmission more secure, as the keys are not known in advance and change at each new session, which means that a simple reboot of the devices updates the keys.

2.3.3 TTN applications

When using The Things Network as a network server for LoRaWAN applications, a user must define and configure "applications" on The Things Network [54, 55], which are the tool to configure the end devices parameters on the network server, and determine what will be done with the transmitted LoRaWAN payloads. The TTN online console provides several configuration and monitoring capabilities for applications:

- Monitoring of all the devices linked to the application.
- Live monitoring of the packets sent by and to the end devices.
- Configuration of the translation of packets sent by and to the devices.
- Configuration of the connections with external applications.

End devices monitoring

Before being able to use an end device with a TTN application, the device must be created in the application console [56, 57]. The device creation is a simple process, when the user simply has to provide a name and a unique ID for the device, as well as some LoRaWAN parameters such as the frequency band and the activation method, i.e. OTAA or ABP. The device keys can also be provided if needed, but TTN can generate them automatically. When the device has been correctly configured and added, the console displays all the necessary information to be specified in the device source code.

Besides the information about a device's identity, the console also provides real-time monitoring of the packets sent by any of the application devices. It displays, for each packet received, its payload as well as a handful of metadata. It also shows TTN-internal packets, the packet forwarded to the external application, and the potential downlink packets or acknowledgements. This panel is useful to assess the correct reception and forwarding of the LoRaWAN packets.

Payload formatters

In the context of IoT applications, the end devices are often resource-constrained, and they should therefore consume as little energy as possible. As such, and also to comply with the LoRa restrictions, the LoRaWAN payloads sent by end devices must be small, and contain all the necessary information in the minimum number of bytes. However, this means that the payloads are difficult to handle by the next step in the data journey, i.e. the external application that will use the data, and are even more difficult to read by a human. The Things Network provides a way to translate the binary payloads into a JSON object before being forwarded to the

external application, for uplink messages, and the opposite for downlink messages: the payload formatters [58]. To benefit from this, the user must simply provide a function, called the decoder (resp. encoder) that takes the binary payload (resp. JSON object), and returns the corresponding JSON object (resp. binary payload). The easiest way to write the function is by using JavaScript, but other formats are available as well, as CayenneLPP (Cayenne Low-Power Payload) [59], which is an Arduino class specifically designed to encode and decode binary payloads for low-power devices.

Connection with external applications

The Things Network can call HTTPS webhooks and forward them the received payload, along with TTN-added, as well as potential user-defined, metadata [60]. Those webhooks calls are event-based and respond to multiple events such as uplink messages, join messages, downlink messages, errors, etc. When such events are triggered, TTN sends a POST request with all the linked data and metadata to the corresponding endpoint. The webhooks can simply be registered on the TTN application console, by providing the corresponding HTTPS URL, the webhook format (JSON or Protocol Buffers) and an optional API key [61].

2.3.4 Fair use policy

The Things Network being a free open LoRaWAN network server, any LoRa developer can use it as backbone for its own application. Without proper regulations, this could render the service unusable if every end device continuously sends packets to the server. On top of the LoRaWAN regulations, detailed in section 2.2.1, TTN also implements a fair use policy, to ensure fair sharing of the network amongst all users. This policy limits the uplink air-time to 30 seconds, and the number of downlink messages to 10, per device and per day [62]. As described in section 2.2.1, the higher the spreading factor, the more air-time a single message needs to be transmitted.

To maximize the number of messages an end device can send, its spreading factor must be minimized, which can be done manually on every device, and is also implemented by ADR after having collected a sufficient number of measurements. A LoRaWAN air-time calculator is available online [63], which computes the air-time of a LoRaWAN payload with respect to its size and data rate, and gives the average number of packets that can be sent while respecting the legal duty cycle and TTN's fair access policy. The number of downlink messages must also be minimized, by configuring the uplink messages as unconfirmed as much as possible, such that the server does not reply with unnecessary acknowledgements.

2.4 Cloud functions

A cloud function is a cloud computing service based on the serverless principle. It consists of code or a container which is run whenever an event is triggered. Those can be of many kind, e.g. HTTP request, time interval, database changes, real-time processing, etc.

Major public cloud providers provide such Function-as-a-Service capabilities (e.g. Azure cloud functions [64], AWS lambda [65], Google cloud functions [66], IBM cloud functions [67], etc.) allowing developers to choose their programming language and deploy their code without having to worry about server maintenance.

It works with the "Pay As You Go" principle. Users only pay for the amount of resources they use as well as for the amount of executions of the function.

2.5 Databases and monitoring

Internet of Things applications generate frequent data, which need to be stored and retrieved quickly and conveniently, by using databases hosted in the cloud. A tremendous number of databases offers exist, with different use cases, such as relational databases, document stores, graph databases, etc. This section will present two offers, Firebase and InfluxDB, that we use to store data related to our system. To provide a more human-readable way of visualizing the stored data, one can rely on monitoring solutions, such as Grafana, the monitoring platform we chose, which will also be described in this section.

2.5.1 Firebase Firestore

Firebase [68] is a cloud environment developed by Google as a part of their Google Cloud Platform. It provides services such as a Backend-as-a-Service containing file storage, real-time databases, document stores and cloud functions, as well as analytic features for developers.

Firebase Firestore [69] is one of those services, and consists of a NoSQL document-oriented database. Like all the other services of the platform, it can be accessed using a REST API, or language specific SDKs available for JavaScript, NodeJS, Python, Java, Swift, Kotlin, etc.

In document-oriented databases, data is stored in the form of documents, that have a JSON format. The documents are contained inside collections, and those

documents can contain data as well as sub-collections. Each document has a unique ID inside the collection, that allows for indexing.

2.5.2 Time series databases

The storing of data which evolves over time requires specific databases paradigms, which allow easy queries over the timing of data. Those time optimized databases are called *Time Series Databases*. InfluxDB [70] is the most widely used one, is open source, and will be used in the presented solution.

It stores every incoming data as a `Point` which has different attributes:

- **measurement**: the kind of measurement contained inside the point (e.g. temperature, pressure, humidity).
- **timestamp**: the time this measurement was done.
- **tags**: additional metadata related to the measurement. The tags are indexed.
- **fields**: the data itself.

The points are stored into so-called "buckets", and are indexed by their measurement, timestamp and tags. Figure 2.3 shows a visual example of those concepts on a graph.

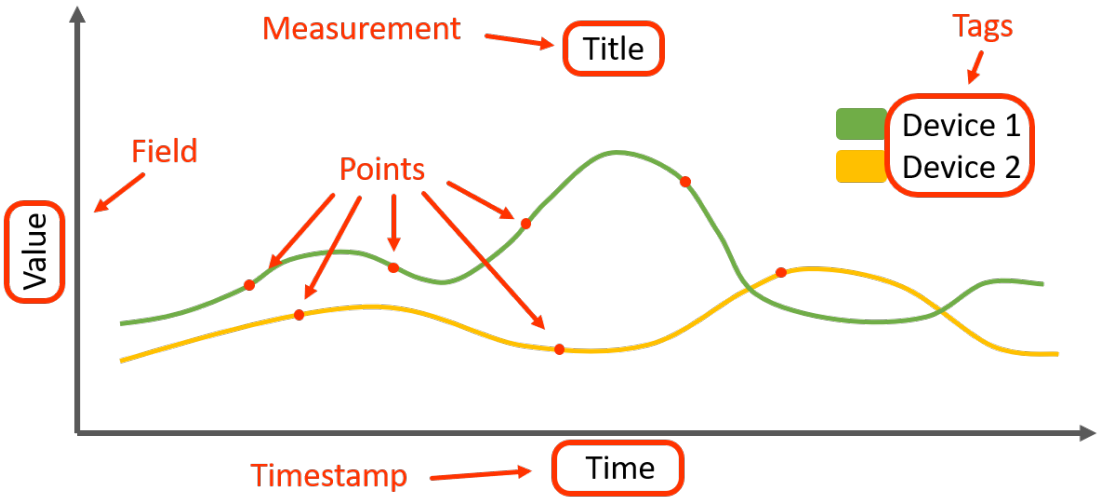


Figure 2.3: InfluxDB concepts

The data can then be retrieved using Flux [71] which is a query language specific to InfluxDB. Access to the data is enabled by tokens with specific access rights.

2.5.3 Monitoring

Monitoring platforms allow a better understanding of the raw data that get injected into the applications. In this field, Grafana is the leading monitoring platform [72]. This open-source software can be installed on-premise (i.e. on user managed servers or virtual machines) as well as on their or any other cloud environment.

Grafana's user interface consists of a web platform accessible with any browser. It allows the creation of dashboards composed of panels. Each panel makes its own queries to data sources (SQL database, InfluxDB database, etc.) and displays the retrieved data using one of the many visualization tools (graphs, gauges, tables, maps, etc.). Figure 2.4 shows examples of gauges and maps.

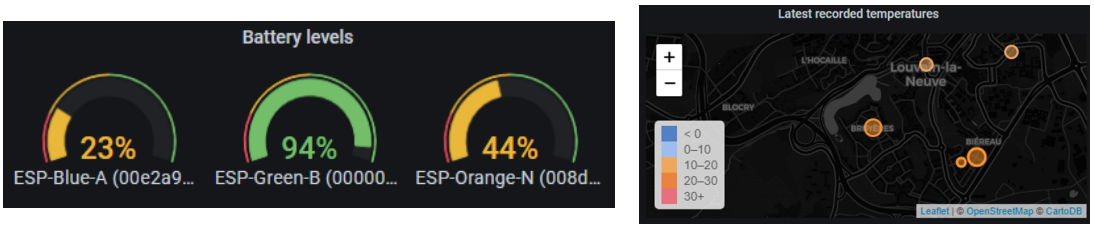


Figure 2.4: Grafana panels. Left: Gauges; right: Map

Grafana also allows event-based alerting such that specific users or groups of users can get alerted whenever a specific condition gets matched. This condition compares the values retrieved inside the database (using Flux for InfluxDB, SQL for SQL databases, etc.) and compares it to the threshold for a user-defined amount of time (e.g. value above 25 for five minutes). Those alerts can be sent using various channels like emails, Telegram, Microsoft Teams, Slack, webhooks calls and many others [73].

Dashboards access can be restricted, using their built-in connection management tool, to specific people or groups. Enabling anonymous access to specific dashboards allows anyone to view those dashboards when accessing the web platform.

2.6 Related work

This section will present previous work that is, in one way or another, related to our master's thesis.

A former UCL student, Robin Schoenmaeckers, analyzed the coverage and performance of the LoRaWAN network in the city of Louvain-la-Neuve for his

master's thesis [74]. This work helped us greatly, on two aspects. On the one hand, thanks to his work, the range of the LoRaWAN network in Louvain-la-Neuve is known, so we could know in advance if our end devices will be able to reach the fixed LoRaWAN gateways. On the other hand, we retrieved the hardware he used for his thesis, which we have extensively used for our thesis. Similar work has been done by Pasolini *et al.* [75], by measuring the performance and coverage of LoRaWAN in Bologna, Italy. Besides that, they also designed a smart city lighting application, using the IEEE802.15.4 network. Magrin *et al.* [76] also evaluated the performance of LoRaWAN, in terms of throughput, coverage, and packet losses, in a smart city scenario, by using the Network Simulator 3 (ns-3) tool. They observed that, in a scenario involving multiple gateways and a realistic traffic model, the LoRaWAN network can scale well, by measuring a packet success rate over 95 % for a gateway accommodating around 15,000 end devices. Cattani *et al.* [77] extensively studied the influence of LoRa physical parameters, as well as the temperature, on LoRaWAN transmissions, by deploying a LoRaWAN network at the Graz University of Technology in Austria, in three cases: outdoor, indoor, and underground. They also measured the effect of the temperature on the transmissions. Their findings were that higher temperatures effectively decrease the received signal strength, and that, in practice, it is more optimal, in terms of bit rate, to always use the highest data rate and opt for retransmissions if the packets were lost, than to use a lower, more resilient data rate. Finally, Sanchez-Iborra *et al.* [78] evaluated the performance of LoRaWAN in three real-world scenarios, i.e. urban, suburban, and rural, by deploying the network in three different configurations in Spain. They analyzed what the optimal LoRa parameters were to achieve the best trade-off between network robustness and data rate, in the three scenarios.

Various work compared LP-WANs with each other, or with other wireless networks that could also be used with IoT devices. Mekki *et al.* [24] compared the same three LP-WANs we mentioned in this thesis, i.e. LoRaWAN, Sigfox and NB-IoT, on several aspects, e.g. coverage, battery life, deployment, etc., and gave motivated use cases for each one of the LP-WANs. This work was the main reference for the comparison present in this thesis. Sinha *et al.* [45] compared LoRaWAN and NB-IoT, from a technical point of view, as well as with similar IoT aspects as the previous work. Raza *et al.* [22] discussed the goals of a LP-WAN, then compared several proprietary technologies, i.e. LoRa, Sigfox, Ingenu RPM, Telensa, and Qowisio, and also discussed multiple standardization initiatives intended for the IoT, i.e. from IEEE, ETSI, 3GPP, IETF, etc. Finally, Lauridsen *et al.* [40] evaluated the coverage of the three LP-WANs mentioned in this thesis, along with GPRS, in a 7800 km^2 area in Denmark. Their findings were that NB-IoT has the best coverage, closely followed by Sigfox, then LoRaWAN. The coverage of GPRS

has been found to be much lower than the three LP-WANs.

LoRa and LoRaWAN can be used for various applications, other than smart city monitoring like we do in this thesis. Rizzi *et al.* [79] discussed how LoRa devices could be used for Wireless Sensor Networks (WSN) in industries, for various sensors and actuators. Their design comes with a small modification to the LoRaWAN protocol that enables Time Slotted Channel Hopping (TSCH), to comply with soft real-time applications such as a factory. Fargas and Petersen [80] designed a geo-location system using LoRaWAN devices, without needing GPS or GSM. This system can be used to track the position of elderly people, for their safety, or animals. It has the advantage of being less power-hungry than GPS based systems. Bouras *et al.* [81] proposed another position-tracking application with wearable LoRaWAN devices, that are similar to ours in hardware and behaviour. They also addressed the choice between LoRaWAN and Wi-Fi for data transmission, and settled with LoRaWAN for battery life and range reasons. Yim *et al.* [82] showed how LoRaWAN devices could be used in agriculture, by deploying a network of various sensors to monitor the crops. Additionally, they evaluated the performance of the network depending on physical parameters, e.g. spreading factor and bandwidth, by deploying the sensors network in an American tree farm.

As we already mentioned multiple times in this thesis, when dealing with battery-powered IoT devices, one wants to minimize their energy consumption, to maximize their battery life. Following this guideline, Bouguera *et al.* [83] developed an energy consumption model for LoRa devices, that can be used to assess and compare different devices and choose the best suited one, based on its energy consumption. Alternatively, Gupta and Fujinami [84] proposed a method to optimize the battery life of moving LoRa devices, while still keeping an acceptable transmission reliability, by dynamically adapting the LoRa parameters, i.e. the spreading factor, transmission power, and bandwidth. The simulation of their method showed that it achieves, for 95% transmission reliability, a reduction in energy consumption of 47% with respect to a fixed, conventional set of LoRa parameters. Finally, San Cheong *et al.* [85] compared the energy consumption of LoRaWAN devices of class A and C, and concluded, as expected, that class C involves a higher energy consumption, and thus a lower battery life.

In this thesis, we will compare the LoRaWAN and Wi-Fi networks, mainly on the aspect of energy consumption. Klimiashvili *et al.* [86] did a similar work, by comparing LoRaWAN and Wi-Fi in terms of delay and energy consumption, in the scenario of sending a file from a source to a destination. The comparison was made using ns-3. They also developed mathematical models for the two aspects,

that they compared with their simulation results. Alternatively, Zemrane *et al.* [87] compared Wi-Fi with another popular network for IoT applications, ZigBee, in the context of IoT applications, by using the OPNET simulator. The comparison went over the networking aspect, i.e. delay, packet losses, throughput, etc.

Chapter 3

Technical description of our solution

As explained in section 1.1, the objective of this thesis is to deploy a system for the monitoring of environmental data in the city of Louvain-la-Neuve, by using mainly the LoRaWAN network. Our solution must use cheap and standard components (microcontroller and sensors), that must be battery-powered, since a wired power supply would considerably hinder their deployment. Finally, it must be easy to deploy, i.e. it must not need an expensive additional infrastructure, like new gateways or access points for wireless communication.

This chapter will present in detail the solution that we developed to meet this objective. We will first explain what are the high level functionalities that we selected for our solution, based on the environment. Then, we will describe the end devices part of our solution, i.e. the microcontroller and the sensors connected to it. After that, the transition backbone network between the end devices and the application will be explained. Finally, our monitoring application, publicly available on the Internet, at the URL `http://thesis.nicwalle.com`, will be presented. Figure 3.1 shows the overall architecture which will be detailed in this chapter.

The code related to our solution has been attached as a compressed ZIP folder to the submission, and is available on GitHub at the following url: `https://github.com/fdekeers/lorawan-smart-lln`.

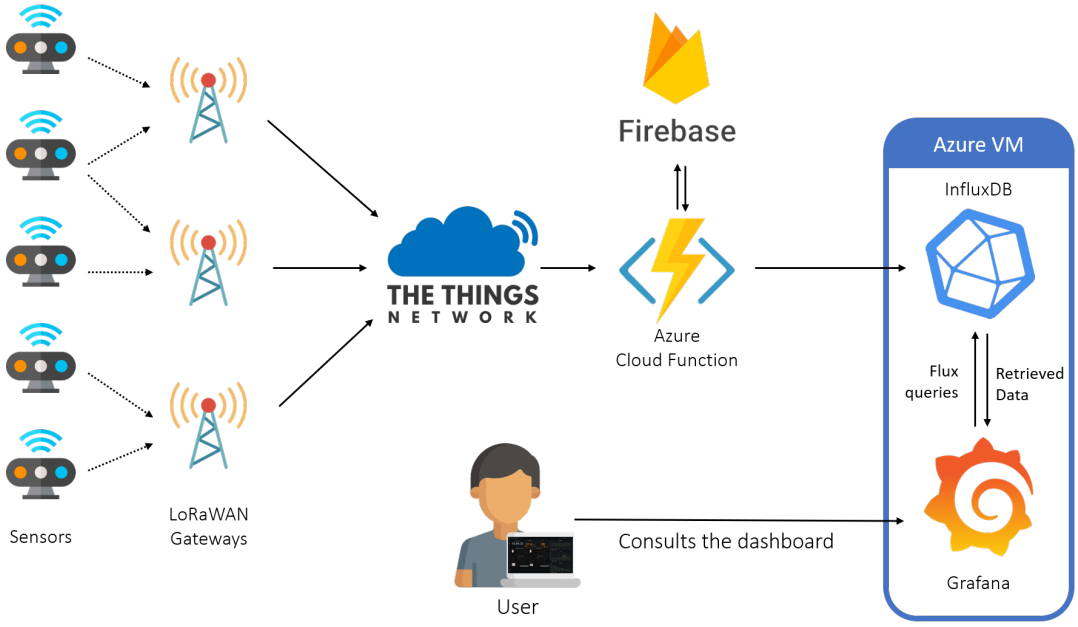


Figure 3.1: Solution architecture (Icons from `https://flaticon.com`

3.1 Environment and functional description

Before diving into the technical details, this section will present a high-level functional description of our solution, as well as the incentives that oriented our work towards it, based on the real-word environment where it will be implemented, i.e. the city of Louvain-la-Neuve.

3.1.1 Environment: Louvain-la-Neuve

The work on this master's thesis takes place in a specific city: Louvain-la-Neuve, Belgium. This city has an important particularity: it was designed and created specifically to host the French-speaking part of the UCLouvain students. As such, its population comprises a large number of student, and reached about 30,000 inhabitants in 2018 [88]. The campus has then been designed for pedestrians, with very few cars, and most parking spaces located outside or under the heart of the city. With this in mind, smart city applications such as parking spaces monitoring for drivers, smart mobility or car pollution measurements, would not be relevant in this city. Therefore, we chose to focus on an application that students could use everyday, before leaving their accommodation to go follow their courses: the measuring of environmental data in different spots of the city, and their display on

a public web interface, accessible to every student or inhabitant through their web browser.

3.1.2 High-level functional description

The practical functionalities of our application will be explained in this section. Firstly, sensing devices measure multiple environmental data in different spots of the city of Louvain-la-Neuve. The possible values are: temperature, pressure, humidity, air quality, and noise. Data are then transmitted wirelessly and collected in one place for further processing. In the normal case, the data is sent using LoRaWAN, and is received by LoRaWAN gateways, that forward the data to our The Things Network application. It is possible to define emergency thresholds for each environmental value: in this case, if one of the values exceeds its threshold, the data is not sent using LoRaWAN, but Wi-Fi, which is faster and more reliable, but consumes more power (this assumption will be evaluated in section 4.3.2). Data sent via Wi-Fi is directly received by our cloud function, which processes and inserts it in to the InfluxDB database, that stores all the measurements in a time-series fashion. After having decoded the LoRaWAN payload, our TTN application also forwards the data to the cloud function. Finally, the measurements are displayed in a user-friendly way, using graphs, gauges and maps, on our web interface, using Grafana and the data stored in the InfluxDB database. This web interface is publicly available, such that anyone can access to and profit from our measured data. All these steps will be detailed in the remaining of this chapter.

3.2 End devices

The first step on the path of the data in our system, is the measurement of environmental data by end devices. The structure of all our end devices is the same. The main component is a microcontroller, namely an ESP32 with a LoRa chip. This component is the programmable one, and thus provides all the logic to gather data from sensors and send them on the network. Multiple modules (hereafter called "sensors") are then attached to the ESP32, to provide the measurement capabilities. The actual sensors attached differ from one end device to another, but are taken from this list:

- Temperature/pressure/humidity sensor (BME280)
- Temperature/pressure/humidity/air quality sensor (BME680)
- Gas concentration sensor (MQ-135)
- Sound sensor (KY-037)
- GPS module (GT-U7)

All these components will be detailed in the remaining of this section. Then, the electrical connections needed between them will be explained. Finally, the behaviour of the whole end device will be described, with a specific focus on the wireless transmission protocol.

3.2.1 Microcontroller - ESP32

The core logic of the end devices is implemented in a programmable microcontroller unit (MCU). We chose the ESP32 by Espressif Systems [89], as it is a widely known and used MCU in IoT applications, which means that a lot of resources are available. It provides, among others, the following features:

- General Purpose Input/Output (GPIO)
- Analog-to-Digital Converter (ADC)
- Universal Asynchronous Receiver Transmitter (UART) protocol [90]
- Inter-Integrated Circuit (I^2C) protocol [91]
- Wi-Fi
- Bluetooth Low Energy (BLE)

As we need a LoRa-enabled MCU for our smart city application, we opted for a version of the ESP32 with a built-in LoRa chip by Heltec Automation, named the WiFi LoRa 32 [92], which is shown on Figure 3.2.

Figure 3.2: WiFi LoRa 32 - Heltec Automation

This MCU is programmable in C++ using the Arduino Integrated Development Environment (IDE) [93], and benefits from numerous libraries to handle its internal behaviour, its wireless interfaces, e.g. Wi-Fi or LoRa, and its connections to external sensors. It also provides an output voltage of 3.3 V, useful to power the different sensors that are connected to it. It can be powered by an USB cable connected to

a power source, by an external battery connected to the device through a JST-1.25 connector, or by directly feeding it 5V or 3.3V to its corresponding pins. Finally, it provides an OLED display, which was useful for debugging purposes, but is not used for our system, as it consumes energy, and would not be useful once the devices are deployed in the city.

To be able to use the device's LoRa interface, we need to query a Heltec license for every device. To do this, we must simply retrieve the device's chip ID by running the example Arduino sketch `GetChipID` from Heltec's ESP32 LoRaWAN Arduino library [94], then provide this ID on Heltec's license querying website [95], which gives the license as an address composed of four 32-bit hexadecimal values. The license must then be provided, in the same fashion as the other LoRa addresses (`DevEUI`, `AppEUI`, `AppKey`) in the device's Arduino code.

3.2.2 Sensors

To measure various environmental values with our end devices, we use an array of small sensor modules, adapted to be used with a MCU. Most of those sensors benefit from publicly available libraries to retrieve their measurements. Figure 3.3 shows the different sensors used in our application. Those will be detailed in the upcoming sections.

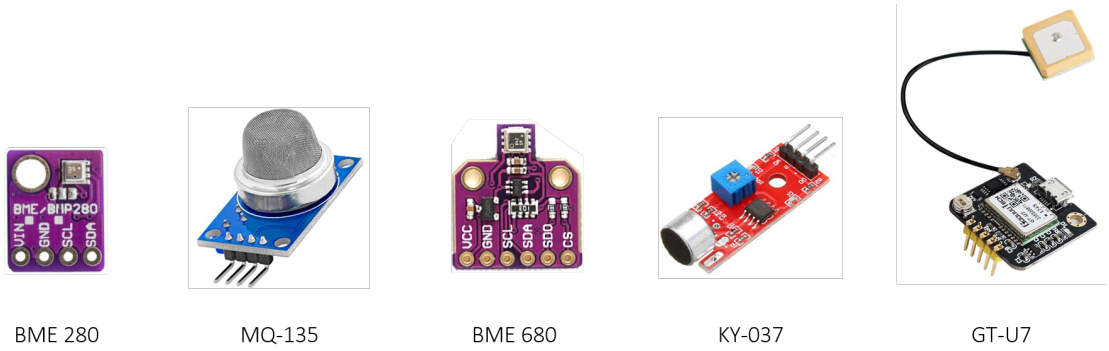


Figure 3.3: Different sensor modules used to collect environmental data

Temperature/Pressure/Humidity - BME280

The BME280 sensor from Bosch Sensortec [96] measures the surrounding temperature, pressure, and humidity, and can also compute an approximate altitude based on these values and the sea level pressure specified by the user (which is

around 101,325 Pa). For our end devices, we use a version of this sensor embedded into a circuit board, produced by AZDelivery [97], which provides the necessary connections to our MCU. The I^2C protocol is used to read the values measured by this sensor. As such, two connections to the MCU are needed: clock (SCL) and data (SDA). It can be powered on by 3.3 V, and consumes a current of the order of 1 μA.

Gas concentration - MQ-135

The first gas concentration sensor we used was the MQ-135, distributed embedded into its circuit board by AZDelivery [98]. This sensor can measure the concentration of various gases in the air, relative to a base value. We chose to measure the concentration of CO_2, as this gas is a good indicator of air pollution. However, this sensor suffers from several problems to be used in our application:

- It must be calibrated with the atmospheric concentration of the desired gas to be able to measure its concentration, which takes some time (around 30 minutes), during which the sensor must measure, and thus be powered on, continuously.
- It must be kept continuously powered on during two days before providing precise measurements.
- While powered on, the current flowing through it is high: around 150 mA.
- It takes 5V as input voltage, while our MCU's output voltage is 3.3V. This was not such a problem, though, because it also works when powered with 3.3V.

Those constraints make it difficult to use the MQ-135 in our application, as we want the sensors to be switched off most of the time to consume as less energy as possible and save battery. This is why we preferred using the next sensor, which also measures air quality while being less power-hungry, as it will be evaluated in section 4.3.3.

Temperature/Pressure/Humidity/Air quality - BME680

The BME680 from Bosch Sensortec [99] is an upgraded version of the BME280 that was already described. Besides temperature, pressure and humidity, this sensor can also measure the concentration of various toxic gases in the air. When the concentration of such gases increases, an internal resistance increases as well, and vice-versa. We can then simply measure the value of this resistance to approximate the concentration of toxic gases. Furthermore, Bosch Sensortec provides the Bosch Sensortec Environmental Cluster (BSEC) Arduino library [100], that permits to derive other environmental values, such as the Indoor Air Quality (IAQ) index,

based on this resistance. While this sensor must still be powered on continuously to provide accurate measurements, its power consumption is way lower (3.3 V - less than 0.1 mA), thus we can afford it, contrarily to the MQ-135.

Sound - KY-037

To measure the ambient noise, we use the popular KY-037 sound sensor, distributed by AZDelivery [101]. This sensor can take both 3.3V or 5V as input voltage. Its measurements can be read from the MCU in two ways: a digital measurement, that is driven high when the ambient noise is above a certain threshold, and low when its is below the threshold; and an analog voltage that indicates the measured ambient noise. The sensibility of the sensor can be configured thanks to a potentiometer. In our application, we use the analog measurement, such that we can monitor the ambient noise around the end device. The analog output of the sensor is simply connected to one of the ADC pins of the MCU. We implemented a lightweight custom Arduino library, called `SoundSensor`, that abstracts the interface between the MCU and the sensor, and makes it easy to read the sensor value.

GPS - GT-U7

In an application such as this one, it is necessary to know the location of each device, such that the collected data can be mapped to a specific place in the city, instead of simply having average values for the whole city coming from different devices. As such, our devices can use a GPS module to retrieve their position in real-time. The module used is the GT-U7 by Goouuu Tech, assembled into a circuit board by MakerHawk [102]. This module takes 3.3V as input voltage. It uses the UART protocol [90] to communicate with the MCU, and thus necessitates two connections: the Rx pin of the module must be connected to a serial Tx pin of the MCU, and vice-versa. It can measure, among others, the latitude, longitude, and altitude, when a GPS signal is found. The measurements are encoded using NMEA sentences [103], a standard encoding for geo-location data, and are decoded on the MCU using the Arduino library TinyGPS++ by Mikal Hart [104].

This module, as the MQ-135 gas concentration sensor, is difficult to use in our application, for the following reasons:

- A GPS module consumes a lot of power. This will be evaluated in section 4.3.3.
- GPS signal is difficult to receive, e.g. when the module is indoor or has no view to the sky. In this case, the module is useless, but still consumes power.
- Our devices will mostly be immobile, which renders the use of a GPS module questionable.

Therefore, we implemented a means to retrieve the geo-location data of an end device only once at startup, and by using an internet connection instead of a GPS module, which will be described in section 3.2.4.

3.2.3 Electrical wiring

The electrical connections with each sensor can be configured separately for each device. Not all pins can be used, as some are needed for hardware functions or communication, via Wi-Fi or LoRa, or are only available as input and not output. The pinout of the ESP32 is shown on Figure 3.4. A more precise pinout showing the usage of each pin is available in [105], and a reference for pin configuration is available in [106].

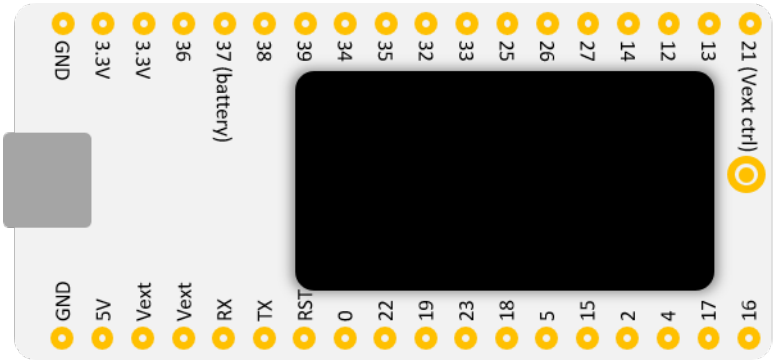


Figure 3.4: Pinout of the ESP32

For power supply, all the sensors must be connected to the ESP32 ground; all of them must also be connected to the Vext pin, except the BME680 that is connected to its 3.3 V pin, as it must be continuously powered on. Vext is a programmable voltage output of 3.3 V, which is useful to power the sensors only when the values are read. It is controlled by the voltage applied on pin 21: when pin 21 is low, Vext is up, and vice-versa. As such, pin 21 cannot be used for any external sensor. Besides the connections to the sensors, one of the ADC pins is used for battery monitoring, and this pin cannot be used by any sensor either. For simplicity, we fixed pin 36 for this usage, unlike the connections to external sensors that are not fixed, and can use any available and compatible pin.

Battery monitoring

It is important, for battery powered systems like our end devices, to be able to monitor the devices' battery in real time, such that we know when the battery

must be recharged or replaced. In our case, we designed a simple system to monitor the battery of each device, which is shown on Figure 3.5. The device's external battery, besides being connected to the device's JST input, is connected to a voltage divider, composed of two resistors. The output of the voltage divider, which is located between both resistors, is connected to an ADC pin, here pin 36. The MCU can then simply read the analog value on pin 36, and map it to a percentage. The higher the battery voltage, the higher the analog value, and thus the higher the percentage. A linear model is used to map the analog value to the battery percentage, with 4.2 V being mapped to 100%, and 3.5 V to 1%. 4.2 V is the maximum voltage value we can reach with our batteries, and 3.5 V is the measured lowest voltage with which the end devices can still work properly. The derivation of the linear model is detailed in section 4.2. The drawback of this system is that it steadily consumes power, as current continuously flows into the voltage divider. To overcome this issue, the resistors must have a high value, such that the current is low. We chose to use 470 kΩ resistors, which was the highest resistor value we disposed of.

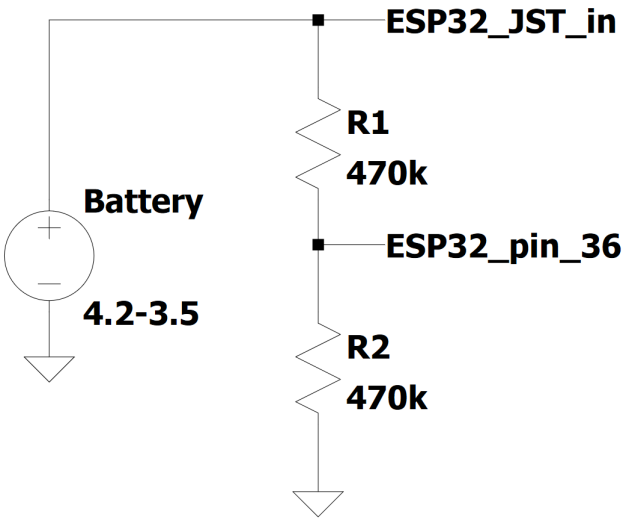


Figure 3.5: Battery monitoring system, using a simple voltage divider

Connections to sensors

The electrical connections to the sensors must respect the following rules:

- The BME280 and BME680 use the I^2C protocol. As such, their SDA (data) and SCL (clock) pins must be connected to the SDA and SCL pins of the ESP32, respectively. The default pins are respectively 21 and 22, but those

can be changed in software. As pin 21 is already used for Vext control, the default pins we chose are the pins used for I^2C communication with the OLED display, i.e. 4 for SDA and 15 for SCL. The fact that the pins are also used by the OLED display does not pose a problem, as the display is not used.

- The MQ-135 and KY-037 must simply have their analog output connected to one of the ADC pins of the ESP32. The easiest pins to use for this are pins 34 to 39 (except pin 36, that is used to monitor the battery), as they are not used for any other purpose. We chose, as default, pin 37 for the MQ-135, and pin 39 for the KY-037.
- The GPS module uses the UART protocol. As such, its Tx (resp. Rx) pin must be connected to a Rx (resp. Tx) pin of the ESP32, which can be defined in software. We chose to use by default the pins 16 for Rx and 17 for Tx, as these are the default UART pins for the second serial interface. Pin 16 is also used as OLED display reset, but this is not an issue since the OLED display is not used.

Figure 3.6 shows a schematic of the ESP32, connected to all the possible sensors, except the BME280, that has identical connections as the BME680. The connections with sensors' pins other than VCC and GND can be changed on the ESP32, provided the according configuration in the device Arduino code, and following the connections requirements for each sensor.

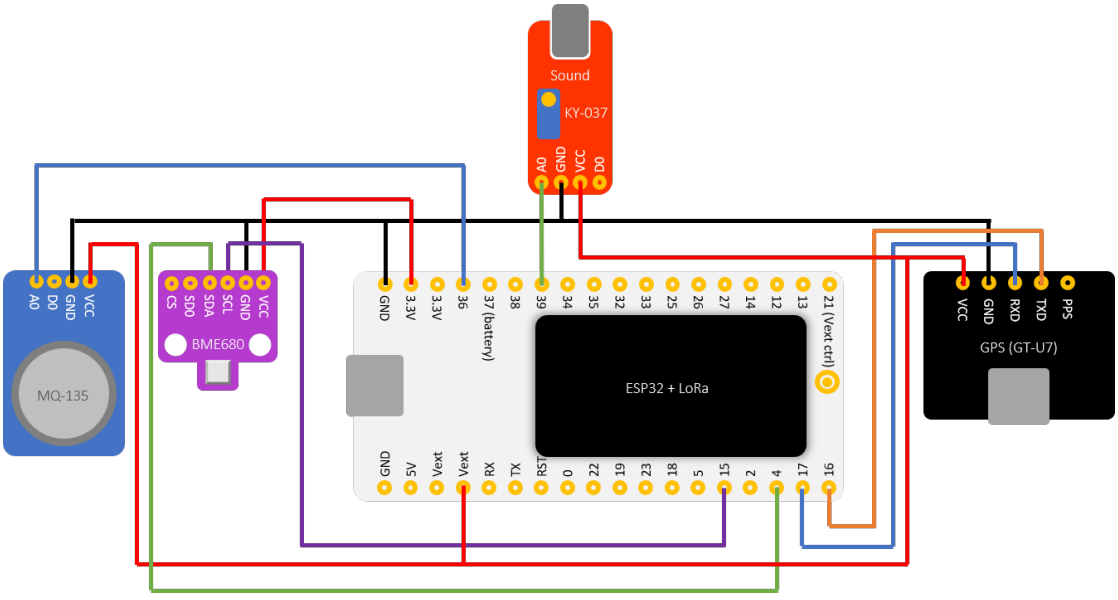


Figure 3.6: Schematic of the electrical connections between the ESP32 and all the sensors

3.2.4 Behaviour

In this section, the behaviour of our end devices will be described. This behaviour is implemented into each of the devices' Arduino sketch, which is the Arduino code for a device. A sketch has a general basic structure, composed of global constants and variables, as well as two functions, `setup` and `loop`. Usually, `setup` is run once when the device starts, and `loop` is run indefinitely afterwards. However, in our case, every time an end device wakes up from deep sleep, it will run the `setup` function again, so the operations that must be issued only at startup must be put in a specific place. The end devices' sketches extensively use the custom libraries `EspDevice`, `Packet`, and `WifiSender` libraries that we implemented ourselves, as well as several third-party libraries. The general behaviour scheme is the following:

- Setup operations
- Repeated operations
- Deep sleep

The setup operations are only done once when the device starts for the first time. The repeated operations are issued each time the device wakes up from deep sleep. The wake-up period can be configured for each device separately. While in deep sleep, all the data memory of the device is wiped, except the Real Time Clock (RTC) memory, which is thus where we will put data that must be kept even when the device goes to deep sleep.

LoRa parameters

As described in section 2.2.1, LoRa devices must be configured with various parameters, i.e. device class, Data Rate, and ADR. With the goal of optimizing the battery life of our devices, a therefore minimizing their energy consumption, the following choices were made for those parameters:

- **Device class A**: in our application, the devices only send uplink packets, and do not need downlink packets. As such, the optimal device class is A, as it is the most energy efficient [85], since we do not need the enhanced downlink reception capabilities provided by classes B and C.
- **Data Rate 5**: corresponds to a bandwidth of 125 kHz, a spreading factor of 7, and a maximum payload size of 222 bytes, as referenced in Table 2.2. This choice is twofold: on the one hand, we need a sufficiently high payload size, to be able to transmit all the environmental data in one payload; on the other hand, a low spreading factor is less power-hungry than a high one [84], which is beneficial in our case. DR6 was not chosen because it was not available with our Heltec end devices. Whereas DR6 could be beneficial by increasing the frequency and thus reducing the transmission time, the

increased frequency could also increase the power consumption. Its effect on the energy consumption is thus unclear.

- **ADR enabled**: we chose to enable ADR, such that the effective data rate is always adapted to the highest possible one while avoiding packet losses, as a higher data rate involves a lower energy consumption. We will further argument this choice by evaluating the actual energy consumption of LoRaWAN transmission with and without ADR, in section 4.3.5.

Retrieving the geo-location via Wi-Fi

As said in the section describing the GPS module (section 3.2.2), a GPS cannot be reliable in every situation. We wanted to have a backup solution, such that every device can retrieve its own location, even without a GPS module, or when the GPS module cannot give accurate measurements, due to a bad satellite signal. As such, the first task in the life cycle of one of our end devices, is to retrieve its own geo-location via Wi-Fi, by using Google Geolocation API [107]. To use this service, we first had to activate a Google Cloud account and create an API key to be used with the service. When this has been setup, HTTP POST requests can be sent from any end device to the following URL, by specifying our API key.:

```
https://www.googleapis.com/geolocation/v1/geolocate?key=API_KEY
```

Along with this request, a JSON object is sent, containing information about surrounding Wi-Fi networks, such as their MAC addresses and signal strength. From this, the service searches into its database of Wi-Fi networks, containing their position, and triangulates the position of our device. It responds with a JSON object containing the computed latitude and longitude, as well as the accuracy of the computation.

If the exchange with the service was successful, and if the computation was accurate enough (i.e. accuracy value below 500 in our code), the latitude and longitude values are kept into RTC memory. As such, this operation must only be done at startup, which prevents the devices from connecting to the service at every wake-up, and issuing unnecessary requests, as the devices will mostly be immobile.

These actions are implemented in our custom `WifiSender` library, with the help of Germán Martín's `WifiLocation` library [108].

Join request

For device activation, we chose to use OTAA, described in section 2.3.2, as it is more secure than ABP. The join request is part of the setup operations, and is thus

only sent once at startup. For the rest of its battery life, the device will use the same session. As a reminder, the join request consists of exchanging its `DevEUI` and `AppKey` with the TTN server, to negotiate the session security keys to authenticate and encrypt the LoRaWAN communication.

Reading values from sensors

The first repeated task is reading the values from the connected sensors, as well as the device's remaining battery. The sensors that are actually connected to the MCU must be added with their corresponding `add` function, in the `setup` function of the device's sketch. For each of the connected sensors, if no error has occurred, i.e. if their values have been read successfully, they are added to the `Packet` object that collects the measurements and arrange them before transmitting them. The values that each possible sensor can read are the following:

- BME280: temperature, pressure, humidity, approximate altitude
- MQ-135: CO_2 concentration
- BME680: temperature, pressure, humidity, approximate altitude, air quality index (IAQ)
- KY-037: noise
- GT-U7: latitude, longitude, altitude

The altitude can be either measured precisely by the GT-U7, or approximated with the BME sensors values. Indeed, the measured temperature and pressure can be used to compute an approximate altitude value by using the *hypsometric formula* [109, 110]:

$$h = \frac{\left(\left(\frac{P_0}{P}\right)^{\frac{1}{5.257}} - 1\right) \times (T + 273.15)}{0.0065}$$

where h is the approximate altitude [m], P_0 is the sea level pressure (fixed to 101,325 Pa), P is the measured pressure [Pa], and T is the measured temperature [°C].

The GPS giving a more precise altitude measurement, we use this one if it is available. However, if the GPS is not connected, or if it cannot give accurate value because of a bad GPS signal, we fall back to the approximate value given by the BME sensors.

Again, as the GPS measurements can be hindered by a bad satellite signal, and since their are not likely to change a lot as our devices are mostly static, we store the latitude, longitude and altitude into RTC memory, overwriting the ones potentially retrieved with Google Geolocation API, as the former are more precise.

In this way, we can send them to our application with every packet, even if the GPS could not measure it for that specific packet.

Sending values

After having read the environmental values from its connected sensors, the MCU wirelessly sends a packet containing those values. The packet can either be sent as a LoRaWAN payload, received by any TTN gateway in the range of the end device, or a JSON object sent over Wi-Fi along an HTTPS request directly to our cloud function, bypassing TTN. The former is the classic case, and the latter happens in case of emergency, i.e. when one of the sensor values has exceeded its threshold. The protocol for packet formatting and transmission is described in detail in section 3.3.

Deep sleep

After having processed all the repeated tasks, the device goes to deep sleep for a certain duration, which has a base duration configurable (in minutes) for each device, and a small random component up to one second higher or lower than the base duration. This random component was present into the default Arduino sketch from Heltec's library [94], and is used to avoid transmission collisions between devices with the same duty cycle. When in deep sleep, the device does not provide external voltage to power the sensors, to save battery. The only data that is saved during deep sleep is the one stored in RTC memory, i.e. among others the latitude, longitude and altitude, as well as the previous measurements if packet format V2 or V3 is used.

3.3 Packet transmission protocol

The most important task our end devices perform, which is central to our whole system, is the wireless transmission of packets containing sensor data. Most of the time, the packet is sent using LoRaWAN to our TTN application, which then forwards it to our cloud function. However, we have also defined an emergency case: if one of the measured values exceeds its threshold, that is configured for each sensor in the software of the device, the packet is sent as a JSON object over Wi-Fi along an HTTPS request directly to the cloud function, bypassing LoRaWAN and TTN. The rationale for this is that there are limitations on the number and size of packets we can send over LoRaWAN, and the ones that exceed the limitations are dropped, what we cannot afford in the case of emergency. However, we assumed that Wi-Fi transmission consumes more power than LoRaWAN, so we cannot exclusively use Wi-Fi to transmit data, as we want to keep a low power consumption. This

assumption will be evaluated and discussed in section 4.3.2.

As a reminder, the LoRaWAN payload size is restricted by the LoRa specifications (section 2.2.1). Moreover, as bigger packets mean a higher power consumption and air-time, we want the LoRaWAN payload to be as small as possible, since we must minimize both of this numbers, for battery life concerns and fair use policy restrictions (section 2.3.4), respectively. We designed a protocol for the packet format and sending strategy, following this guideline. Three versions of the protocol exist, that can be configured when initializing the device:

- V1: a packet containing the measurement is sent at every measurement.
- V2: measurements are batched into RTC memory, and sent as aggregates every X measurements, where X can be configured for each device.
- V3: similar to V2, but some values are sent only once for the whole batch. These values are the latitude, longitude and altitude, as they are not likely to change over the batch.

We introduced packet format V2 and V3 to further minimize power consumption, following the assumption that the LoRaWAN transmission is a power-hungry operation compared to the gathering of sensor data, and that transmitting a larger payload less often was more beneficial, in terms of energy consumption, than transmitting a small payload at each measurement. This will be evaluated and discussed in section 4.3.4.

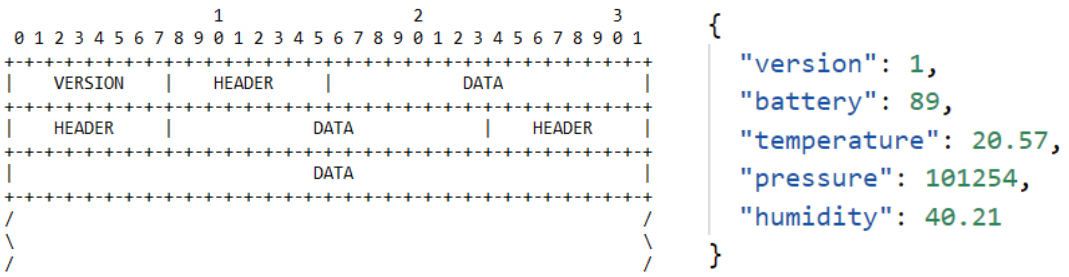
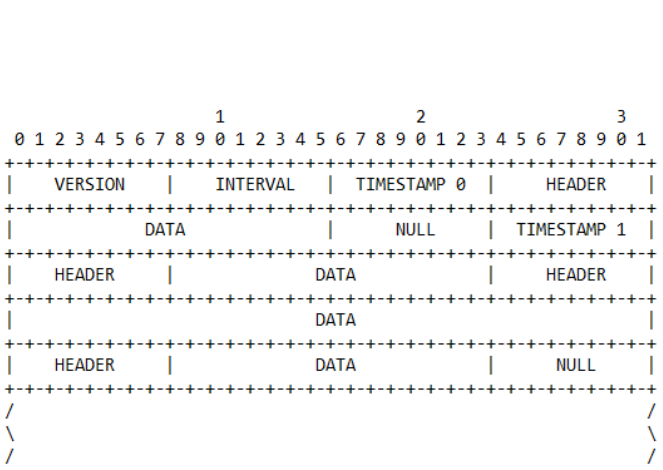


Figure 3.7: Packet format V1. Left: LoRaWAN payload; right: JSON object

Packet format V1 is shown on Figure 3.7; left is the LoRaWAN payload, and right is the JSON object. As every version, the LoRaWAN payload starts with the version number, in 8 bits. Then, every value is placed one after the other in the payload, preceded by its representative header, in 8 bits, which is used to know to which environmental measure it corresponds. A header is needed to identify the nature of the value, as the data actually present in the payload may vary, as well as their order, depending on the sensors connected to the device. The size of the data

values is variable; it can be in 8, 16, or 32 bits. The JSON object simply provides the packet version and all the measured values as fields. In addition to this, and as all the other packet versions, it also provides metadata about the sending device, i.e. its `DevEUI` and authorization token (discussed later in section 3.5.1), such that the cloud function can identify and authentify which device sent the packet. This is not shown on the figure for clarity.

Packet format V2 is shown on Figure 3.8. The first data present in the LoRaWAN payload are the version number and the time interval, in minutes, between each measurement, both in 8 bits. Then, each measurement, which contains the set of values measured at the same wake-up, is present in the same format as for packet format V1. Each measurement is also preceded by an 8-bit timestamp, starting from 0 and incrementing at each measurement, and followed by a NULL byte, indicating the end of the measurement. This NULL byte is necessary, as the size of measurements can vary, depending on the sensors that are connected to the device. The JSON object also contains the version number and interval at the beginning, then contains an array of the batched measurements. Each measurement contains, as in the LoRaWAN payload, its timestamp, then all the values similarly to packet format V1.




```
{
  "version": 2,
  "interval": 10,
  "measurements": [
    {
      "timestamp": 0,
      "battery": 87,
      "temperature": 20.92,
      "pressure": 101269,
      "humidity": 42.69
    },
    {
      "timestamp": 1,
      ...
    }
  ]
}
```

Figure 3.8: Packet format V2. Left: LoRaWAN payload; right: JSON object

Packet format V3 is similar to V2, the only difference being that the constant values, i.e. latitude, longitude and altitude, are only placed once at the beginning

of the LoRaWAN payload or JSON object, after the interval and before the first measurement. In the LoRaWAN payload, they are also preceded by their 8-bit header, as we cannot predict which value will actually be present in the payload beforehand. Since some values are only transmitted once for the whole batch, packet format V3 allows to save bandwidth and thus power and air-time, compared to V2, for a same size batch. Alternatively, since one measurement takes less space in the payload, more measurements can be batched before reaching the LoRaWAN application payload size limit of 222 bytes.

All the packet formatting, depending on the version, is performed by our custom libraries `Packet`, for the LoRaWAN payload, and `WifiSender`, for the JSON object. The main `EspDevice` library simply adds the measured values to a `Packet` object, and does not handle the formatting. For Wi-Fi transmissions, the `WifiSender` library also handles the connection to the Wi-Fi access point and the transmission itself.

Table 3.1 shows the different data that are currently supported by the protocol. For each data, the following values are shown:

- Data: name of the data, with its corresponding unit. "Raw" means the data is transmitted as the analog value read by the ESP32, i.e. an integer between 0 and 4095.
- Type: data type used to represent the data.
- Width: size of the data, in bits.
- Header: header byte representing the data in the LoRaWAN payload.
- JSON field: field representing the data in the JSON object.

New data can easily be defined, by simply defining its header byte and JSON field. Our libraries then provide functions to add data to the LoRaWAN payload in the `Packet` library, i.e. `appendIntToPayload` or `appendFloatToPayload`, or the JSON object in the `WifiSender` library, i.e. `appendInt` and `appendFloat`, depending on the data type.

3.4 Backbone: The Things Network

The Things Network is the LoRaWAN network server used for our application, described in detail in section 2.3. When a packet is sent by one of the end devices using LoRaWAN, the nearby gateways forward them to our The Things Network application, that is identified thanks to its `AppEUI`. The payload must then be decoded and forwarded to our external application, i.e. our Azure cloud function. These steps are explained in this section.

Data	Type	Width [bits]	Header	JSON field
Battery [%]	int	8	0x01	`"battery"`
Temperature [°C]	float	32	0x02	`"temperature"`
Pressure [Pa]	float	32	0x03	`"pressure"`
Humidity [%]	float	32	0x04	`"humidity"`
Altitude [m]	float	32	0x05	`"altitude"`
Light [raw]	int	16	0x06	`"light"`
Latitude [°]	float	32	0x07	`"latitude"`
Longitude [°]	float	32	0x08	`"longitude"`
CO_2 concentration [ppm]	float	32	0x09	`"co2"`
Noise [raw]	int	16	0x0A	`"noise"`
Air quality index (IAQ)	float	32	0x0B	`"airQuality"`

Table 3.1: Data currently supported by the protocol

3.4.1 TTN gateways

To be able to deploy a LoRaWAN network and use The Things Network, gateways must be installed in the range of all devices, such that the transmitted payloads can be received and processed further. For our application, we can use two types of gateways: fixed gateways, and mobile custom gateways.

The fixed gateways have been installed by UCLouvain, on the university buildings. There are currently three such gateways in the city, that we can use freely for our application, as any deployed gateway can be used by any TTN user, provided they are in the range of our devices. Figure 3.9 shows the location of those three gateways in the city, as advertised by The Things Network on their homepage [50].

Two custom gateways have been built by a former UCLouvain student, Robin Schoenmaeckers, for his master's thesis where he analyzed the performance and coverage of the LoRaWAN network in Louvain-la-Neuve [74], by following the instructions by Gonzalo Casas [111]. Those gateways are composed of the following components:

- A Raspberry Pi 3, which runs the software: packet forwarding and connection to TTN.
- An iC880A-SPI concentrator board, which handles the radio modulation and demodulation.
- An antenna, to improve the gateway range.

Figure 3.10 shows one of those custom gateways. Their advantage is that they are mobile, and can be placed in the direct neighbourhood of the end devices, to

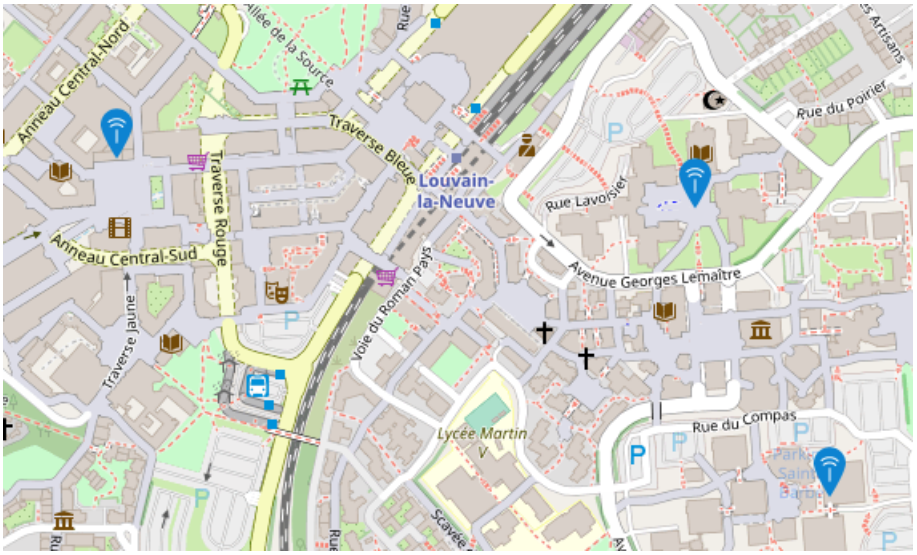


Figure 3.9: Fixed gateways location in Louvain-la-Neuve

ensure their data is correctly received. However, they must be connected to the Internet to operate correctly, either via an Ethernet cable, or via Wi-Fi.

3.4.2 Payload decoder

The LoRaWAN packets are first decoded, thanks to the payload decoder (described in section 2.3.3), which is implemented in JavaScript and configured in our TTN application. It translates every received payload into a JSON object, that will then be forwarded to our cloud function. The decoded JSON object is identical to the one sent directly by the end devices in case of emergency, as both are sent to the cloud function with the same purpose. In practice, the decoder parses the payload and updates the fields of the JSON object, depending on the place of the value or the preceding header. It is composed of two main functions that are called depending on the first byte of the payload, which represents the packet version: `decoder_v1` is used for V1, and `decoder_v2` is used for V2 and V3. Our payload decoder is available in the `the-things-network` directory of the thesis' repository: `https://github.com/fdekeers/lorawan-smart-lln/tree/main/the-things-network`.

3.4.3 Forwarding to the cloud function

After having been decoded, the payload can be forwarded to the backend. Our TTN application has been configured with the HTTPS URL of our cloud function, such that every uplink decoded payload is forwarded as a JSON object to the cloud

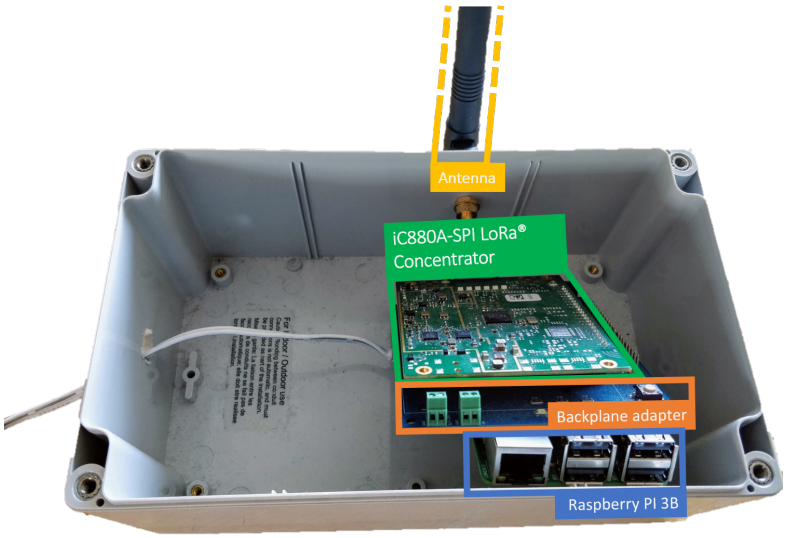


Figure 3.10: Custom gateway

function.

3.5 Backend and monitoring platform

After having been forwarded by our The Things Network application, data arrive in our cloud backend, depicted on Figure 3.1. The backend infrastructure contains multiple components such as cloud functions, virtual machines and databases. The following sections will detail the path taken by a measurements packet through all those components.

3.5.1 Entry point: cloud function

As explained in the previous section (section 3.4.3), when The Things Network receives and decodes an incoming packet, it forwards it towards an HTTPS endpoint. This endpoint is, in this implementation, a cloud function running on Microsoft Azure [64].

The Azure choice was made by convenience, due to Microsoft's partnership with UCLouvain. It can be replaced by any other cloud function provider or a self-hosted HTTPS endpoint. The programming language used for that cloud function is Node.js.

Authorization token

When receiving the HTTPS POST request, for security purposes and to avoid unauthorized requests, the *authorization* header is checked. This header has to contain a valid JSON Web Token [112] signed using the application's secret (i.e. a user-defined password or certificate). If this header is valid, it is then checked against a database to assess that, in addition to being valid, it has not been revoked. If a token validation fails, the function's execution stops and responds with an HTTP 401 error code.

Those tokens can be generated and signed using the secret (i.e. a password). They do not need to contain any specific information nor expiring date. This design choice comes from the fact that the IoT sensors will be placed around in the city for as long as possible. Specifying an expiry date means they would need to be updated regularly for the cloud function to still accept HTTP requests coming from our TTN application or directly from the devices, which conflicts with the idea of leaving the devices in autonomy for their whole battery life. To revoke a token, one only needs to remove it from the token database. A revoked token will pass the signature check but fail the validity check. To ease the token management, a small web application has been created and is available in the `web-app` directory of the thesis' repository: `https://github.com/fdekeers/lorawan-smart-lln/tree/main/web-app`.

Firebase

Once the token's validity has been checked, the function retrieves extra metadata linked to the device inside Firebase Firestore's database. Each device ID is associated to a document inside a *devices* collection. Those metadata can be:

- Display Name: a human readable name for the device. Allows a better visualization (explained later in section 3.5.2)
- Geo-location: the location (latitude and longitude) of the device. This value will be used and updated if certain criteria are met. Those criteria will be detailed in the next section (section 3.5.1, Device location).
- Any other relevant data. *Note: only the data listed above will currently be inserted in the database or have an impact on the rest of the code's execution. Code modifications are required to support new data.*

Device location

Depending on the devices, there can be various location sources:

- Device on-board GPS: acquired thanks to the (optionally) attached GPS module.

- Wi-Fi location: acquired via Wi-Fi thanks to Google Geolocation API (described in section 3.2.4).
- The Things Network device location: manually specified by the admin on The Things Network application dashboard.
- Firebase Location: manually specified by the admin and dynamically updated by the function.

The two first location sources (GPS and Wi-Fi) are merged by the IoT device and only the most precise is sent to the endpoint. This one will be referenced as the *payload location*.

When forwarding the decoded payload to the HTTPS endpoint, The Things Network inserts, if available, the manually specified device location in the metadata field. This location will be referenced as the *metadata location*.

The location that will be associated to the measurements is the first one available in this ordered list:

1. Payload location: if available, gets inserted in the Firebase Firestore database to keep the most recent location
2. Firebase location
3. Metadata location
4. Default: latitude = 0°, longitude = 0°

Dealing with batch measurements

The packet formats V2 and V3, as explained in section 3.3, allow devices to send multiple measurements from different timestamps at once. To deal with those data and insert them into the database with the most accurate timestamp possible, the cloud function needs to recompute the timestamp. Figure 3.8 shows an example of a V2 JSON object.

The actual time for the measurement i, $\texttt{time}_i$, will then be computed as follow:

$$\texttt{time}_i = \texttt{current\_time} - \texttt{interval} \times (\texttt{MAX\_TIMESTAMP\_VALUE} - \texttt{timestamp}_i)$$

where `current_time` is the current time measured by the cloud function, `interval` and $\texttt{timestamp}_i$ come from the JSON object, and `MAX_TIMESTAMP_VALUE` is the maximum timestamp value in the *measurements* array present in the JSON object.

3.5.2 Time-series database and monitoring interface

To store the measurements, the choice has been made to use a time series database, as we are dealing with values evolving over time. The most popular, and the one we chose to use, is InfluxDB [70], described in section 2.5.2.

InfluxDB

InfluxDB has been installed on a virtual machine hosted on Azure using the official documentation [113]. The choice for Azure was made for convenience, and any publicly accessible server can be used.

When installed, InfluxDB exposes a UI as well as an API to interact with. The latter requires creating tokens with associated rights to be used. The data can then be inserted inside buckets and queried using the Flux query language.

Insertion of data into InfluxDB

Note: for the following paragraphs, the measurement notion in InfluxDB is not the same as the one from the packets. In the packets, a measurement is a collection of variables which have been measured at the same time (e.g. temperature, pressure, etc.). In InfluxDB, a measurement corresponds to a single variable with all its related information (e.g. temperature with its longitude, latitude, device ID, etc.).

After getting all the required data from the payload and Firebase, the cloud function creates InfluxDB measurements associated to the measurement's location, time, device name and device ID. Figure 2.3 shows the different concepts of measurement, tags, fields and timestamp of InfluxDB. Each *Point* inside the bucket is associated to one measurement. It contains a timestamp, zero or more tags which are indexed, and fields containing the evolving values.

The following example assumes a temperature measurement:

```
{
  "measurement": "temperature",
  "timestamp": "2021-04-03T08:04:30+00:00",
  "tags": {
    "deviceId": "...",
    "deviceDisplayName": "...",
  },
  "fields": {
    "value": MEASURED_TEMPERATURE,
```

```
    "latitude": ...,
    "longitude": ...,
  }
}
```

Those values are inserted inside the database using the Node.js InfluxDB client SDK [114], which relies on the REST API.

Monitoring with Grafana

To allow for an easier and more user-friendly visualization of the data, we deployed a monitoring interface, using Grafana (described in section 2.5.3), which is publicly available at the URL `http://thesis.nicwalle.com`. It has been installed on the same virtual machine as InfluxDB using the official documentation [115] (and can be installed on any other server as well).

The different panels on the dashboards query the database using the Flux query language. The relevant data is retrieved over HTTP. Each data can be displayed in many ways, such as on a map to show the last value in multiple locations, on a graph to see the evolution of the value, or in other different forms such as gauges for the battery level. Figure 3.11 shows the monitoring dashboard with different visualization panels, with actual data coming from our deployed devices.

Figure 3.11: Overview of the Grafana monitoring interface

Chapter 4

Experiments

In this fourth chapter, we will describe the experiments that we made throughout the year, to understand and characterize our system. Those experiments consisted of a first prototyping with simple devices, the design of our battery monitoring system, the electrical characterization of our end devices in multiple scenarios to estimate their battery life, the reliability assessment of the wireless transmissions, and the real-world deployment into the city of Louvain-la-Neuve.

4.1 Prototyping and configuration

Before developing our solution with the ESP32 and the various sensors, we set up a prototype of the application. This section will describe this prototype, and answer to the following question: "**What is the necessary configuration on end devices and TTN application to be able to transmit data using LoRaWAN ?**".

For this prototype, we used two different types of devices, both retrieved from Robin Schoenmaeckers' thesis [74]: the custom gateways, that were already described in section 3.4.1, and simple end devices produced by The Things Network, called "The Things Nodes" [116], hereafter referred to as "nodes", that are specifically intended for the prototyping of LoRaWAN applications, by providing simple configuration and transmission. They possess basic sensors (light, temperature, accelerometer), cannot be connected to external sensors, and can only communicate via the LoRaWAN network. Even if they are not very useful in our case, they are still accommodated by our application, and can be used as simplistic end devices.

For the end devices to be able to send LoRaWAN packets to the application, identifiers and keys must be provided into their Arduino code. For the nodes, as

they can only use OTAA, we need to provide the `AppEUI`, used to identify the application that must process the join request, and the `AppKey`, used to derive the session keys, i.e. `NwkSKey` and `AppSKey`. Both of those keys are generated when the device is created on the TTN application. The `DevEUI`, in the case of the nodes, is hard-coded into the hardware and cannot be changed, thus it must be provided to the application. However, for generic devices, i.e. our ESP32, we can provide any `DevEUI` in software. In this case, the TTN application can also generate it. In the case of other devices, if ABP is used instead of OTAA, the `DevAddr`, `NwkSKey` and `AppSKey` must be provided in software, and can again be generated by the TTN application.

On the application side, numerous parameters must be configured, for each end device:

- Unique device ID
- Device activation method, i.e. OTAA or ABP
- Frequency bands used, i.e. European 838 MHz band
- LoRaWAN version and regional parameters version, provided by the device manufacturer (in our case, version 1.0.2 rev B)
- Device class, i.e. A, B or C

For generic end devices, the activation method, frequency band, and device class must also be provided into the device's software. For the nodes, however, only the frequency band must be provided, the other parameters being fixed.

One detail that must be taken into account when encoding and transmitting a payload, is its endianness [117]. Indeed, LoRaWAN payloads are usually encoded in big-endian, i.e. with the most significant byte occurring first, whereas the end devices' architectures are generally little-endian, which is the case for the nodes as well as the ESP32. When populating the payload, the byte order of the data must thus be reversed, otherwise the values will be decoded incorrectly by the application.

4.2 Battery monitoring system

As already mentioned, it is important, for battery-powered devices, to be able to monitor their own battery, such that an operator knows when the device's battery must be recharged. This section will detail our battery monitoring system, described in section 3.2.3, and more precisely **how to minimize its power consumption**, and **how to accurately monitor the battery percentage using this system**

in conjunction with the ESP32.

As a reminder, the battery monitoring system is composed of a simple voltage divider. The output of the voltage divider, which depends on the voltage of the battery, is then read by the device to derive the remaining battery percentage. The mapping between the battery voltage and the computed percentage is not trivial, and involves a linear model, which will be described in this section.

4.2.1 Base concepts

Before delving into the derivation of the linear model, some base theoretical concepts must be explained, to fully understand the model: what is a voltage divider, and how does the ESP32 read analog input values.

Voltage divider

A voltage divider [118] is a simple electrical assembly, composed of two resistors in series, which divides an input voltage by a factor depending on the resistors' value. A basic voltage divider schematic is shown on Figure 4.1. Based on the notations of this schematic, the voltage V_{out} can be computed from the other values, with the following equation:

$$V_{out} = \frac{R_2}{R_1 + R_1} \times V_{in}$$

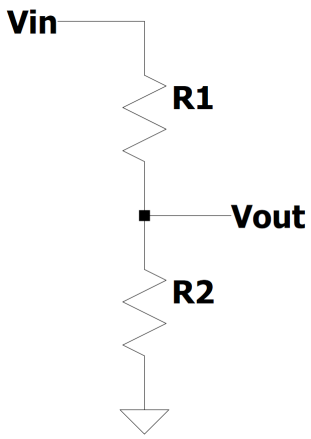


Figure 4.1: Basic voltage divider

As a connection between a positive voltage V_{in} and the ground exists in this assembly, it involves a steady current flow. The value I of this current can be

computed with the following equation, by following Ohm's law [119]:

$$I = \frac{V_{in}}{R_1 + R_2} \tag{4.1}$$

ESP32 analog read

With its Analog-to-Digital (ADC) pins, the ESP32 can read an input voltage between 0 V and 3.3 V. The software function used is `analogRead(GPIO)` [120], where `GPIO` is an integer, being the GPIO pin number of the ADC pin used, according to the pinout schema shown on Figure 3.4. The value read is encoded as a 12-bit unsigned integer, which means the function returns an integer value between 0 and 4095. The higher the input voltage, the higher the returned value. However, the mapping of voltage to integer is not linear, and contains a plateau at 4095 before decreasing, as discussed in [120].

4.2.2 Minimizing current

To minimize the power consumption of end devices, and thus increase their battery life, we want to minimize the current flowing into the voltage divider. To this end, following equation 4.1, we observe that we must maximize the resistors' values. For simplicity, we chose to use the same resistor value twice, such that $V_{out} = \frac{1}{2}V_{in}$. Therefore, we use two resistors of 470 kΩ which is the greatest value we disposed of. With this value for both resistors, the maximum current I evaluates to the following, with an input voltage V_{in} set to its maximum value, 4.2 V:

$$I = \frac{4.2}{470\text{k} + 470\text{k}} = 4.468\ \mu\text{A}$$

This current value is very low, and was not even detected by a multimeter when measuring the current with and without the voltage divider. Moreover, it is two to three orders of magnitude lower than the deep sleep current of the end devices, which will be measured in section 4.3.1. We can thus conclude that a voltage divider with two 470 kΩ resistors involves a negligible current flow, and can be used as a battery monitoring system without hindering our devices' battery life.

4.2.3 Linear model

An end device cannot work until its battery voltage drops to 0 V; it will stop working whenever the battery crosses a low voltage threshold. To accurately monitor our devices' battery, we want to map the maximum battery voltage, i.e. 4.2 V, to 100 %, and this low threshold to 0 %. We determined this low threshold, which is 3.5 V, by powering an end device with decreasing voltage, and observing when

it stopped working correctly, i.e. when it could not use LoRaWAN transmissions anymore.

To this end, we designed a linear model, that computes a battery percentage value based on the analog value returned by the `analogRead` function. To develop this model, we proceeded in two steps: a preliminary model to derive the relation between battery voltage and analog value read, then the actual model linking the battery percentage to the analog value.

Analog value as a function of battery voltage

The preliminary model is a linear model giving the analog value as a function of the battery voltage. As the analog value is read as the output of the voltage divider, the battery voltage is divided by 2 before being read. As such, the problem, mentioned earlier, of the plateau with high voltage values is not relevant here, since the voltage read by the ESP32 will never be above 2.1 V, i.e. half of the maximum possible voltage of 4.2 V. As such, the relation between the analog value read and the input voltage can be approximated to a linear relation, which we will derive with experiments.

Battery voltage [V]	Analog value [/]
3.656	1725
3.665	1727
3.891	1895
4.145	2091
4.185	2112

Table 4.1: Measurements of analog value depending on the battery voltage

To derive this preliminary model, we measured, for multiple battery voltages, the voltage with a multimeter, as well as the analog value read by the ESP32, corresponding to this voltage. Results are shown in Table 4.1. Based on those results, we derived a linear model, shown on Figure 4.2. The linear equation of this model, with x being the battery voltage, and y the analog value, is the following, along with its coefficient of determination R^2:

$$y = 743.71x - 996.7$$
$$R^2 = 0.9996$$

As R^2 is very close to 1, we can conclude that the approximation with a linear model is relevant in this case. With the derived equation, we can compute the

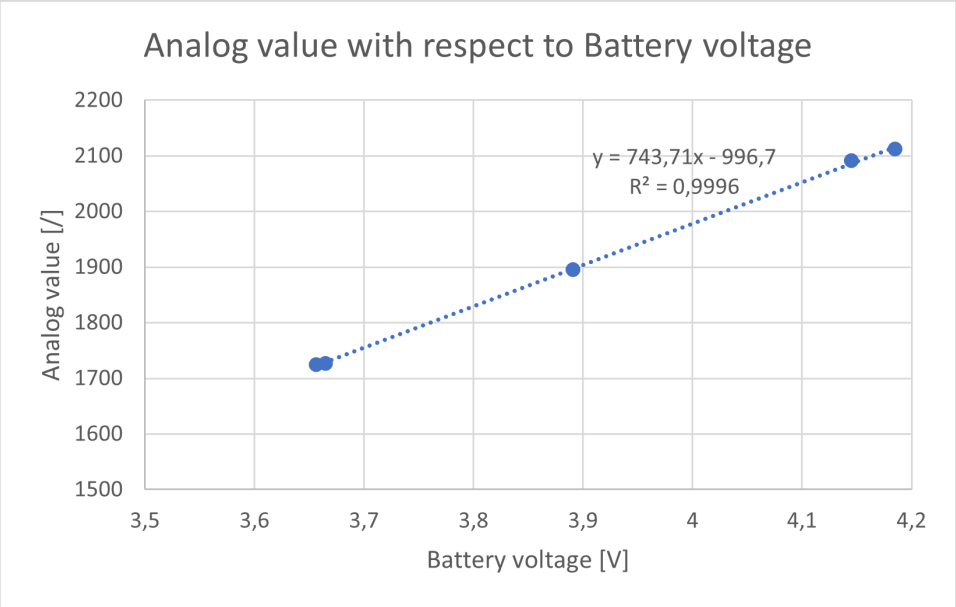


Figure 4.2: Linear model of the analog value as a function of the battery voltage

analog value for the two extreme voltage values, 4.2 V and 3.5 V:

$$y_{4.2} = 743.71 \times 4.2 - 996.7 = 2126.882 \approx 2127$$
$$y_{3.5} = 743.71 \times 3.5 - 996.7 = 1606.285 \approx 1606$$

For simplicity, and as decimal precision is not so important in this case, we will use the approximate integer values 2127 and 1606 for the remaining computations.

Battery percentage as a function of analog value

Knowing the analog values corresponding to the minimum and maximum battery voltage values, we can then derive a linear model giving the battery percentage as a function of the analog value. The choice of a linear model was made for simplicity, because an extremely precise battery measure is not needed in our case. A linear model respects the following equation, with x being the analog value read by the device, and y the battery percentage:

$$y = m \times x + p$$

The parameters m and p can be derived based on the extreme values computed earlier:

$$m = \frac{\Delta y}{\Delta x} = \frac{100 - 0}{2127 - 1606} = 0.1919$$
$$p = y - m \times x = 0 - 0.1919 \times 1606 = -308.25$$

We finally obtain a linear model giving the battery percentage (y) as a function of the analog value (x), which respects the following equation:

$$y = 0.1919x - 308.25$$

In practice, to not lose too much precision when computing the battery percentage, m and p are computed in software based on the extreme analog values, and are reused as such for further calculations, instead of using fixed values. Due to imprecision in the values used or in the models, or due to small variations among different end devices, it is possible that the computed battery percentage goes over 100% or below 1% while still being functional. To overcome this inconsistency, the battery percentage value is limited to 100% or 1% before being sent in the packet. Those tasks are carried out inside the function `getBattery` of our custom `EspDevice` library.

4.2.4 Assessment

To assess the validity of our battery monitoring system, we can observe the battery percentage values sent by different devices, and their evolution over time. Figure 4.3 shows a graph of the battery percentage of different end devices, evolving over time, as displayed by Grafana.

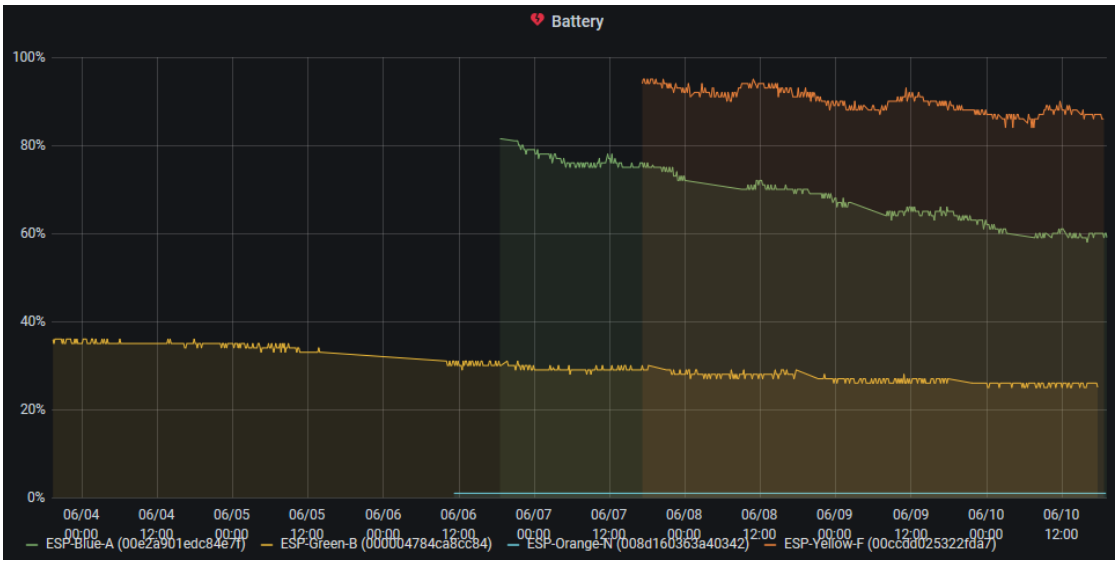


Figure 4.3: Battery values received from the end devices, displayed by Grafana

The device related to the blue curve steadily sends a battery of 1%, because the battery monitoring system has not been installed on this device. For the other

devices, which have a batter monitoring system installed, we can observe that the system works correctly, with battery values that continuously decrease over time. The small local instabilities come from the fact that the analog reading of the voltage is not extremely precise.

4.3 Energy consumption & Battery life

As already mentioned multiple times, one critical aspect when using IoT devices is their battery life. Indeed, to minimize maintenance costs, the battery life must be predicted and maximized. The objective of the following experiments is thus to answer the questions "**What is the battery life of our end devices ?**", and "**What are the parameters influencing this battery life ?**". To be able to forecast and increase the battery life of our end devices, we made extensive measurements of their energy consumption, in different scenarios. We were then able to determine the parameters for optimizing energy consumption and battery life, and to compute a battery life estimation using those parameters.

The measurements were made at the UCLouvain Welcome platform [121], that provides equipment for electrical measurements to students. To measure the energy consumption, we used the following equipment:

- An ESP32 end device, connected to a BME680 for most of the measurements. As such, the packets sent during the measurements will contain most of the accommodated data, and be representative of a real world operation.
- A DC power source, the E3631A by Keysight [122].
- A digital multimeter (DMM), the DMM7510 by Tektronix [123]. This instrument can measure low deep sleep current as well as fast high transients, and can export the measurements via USB.

The experimental setup is shown on Figure 4.4. Except mentioned otherwise, the DC power source was set on 3.7 V, as it is a value in the range of possible voltage, and the base voltage of our batteries. It was connected in series with the DMM and the end device. The actual behaviour of the end device during the measurement depends on the measurement, and will be precised. For each measurement, the DMM measured the current flowing through the end device with respect to time, for a specific duration.

By knowing the voltage V and measuring the current I, we can derive the energy consumption of the device. As a base, we have the equation for the power consumption P depending on the voltage V and the current I, and the energy E

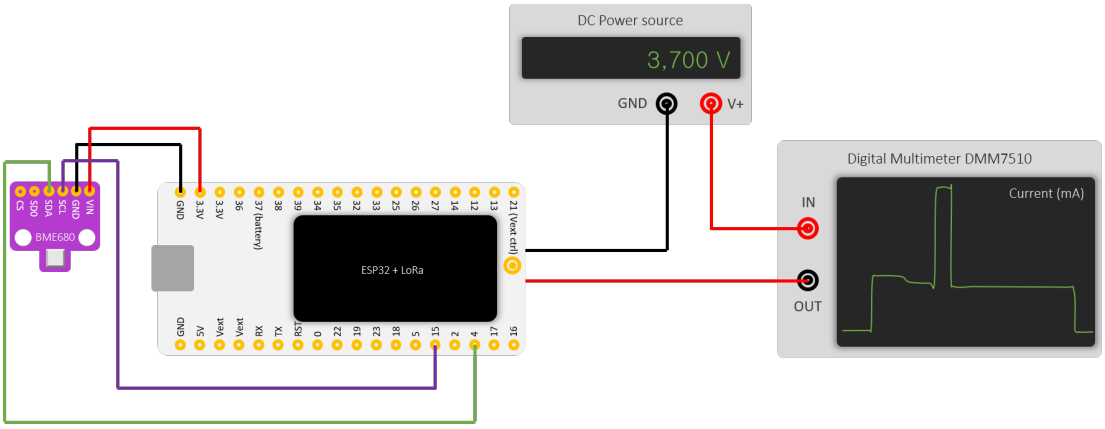


Figure 4.4: Experimental setup for energy consumption measurements

depending on the power P and the time t [119]:

$$P = VI \quad [\mathrm{W}]$$
$$E = \int P \,\mathrm{d}t \quad [\mathrm{J}]$$

Therefore, and as the voltage V is fixed, we can compute the actual consumed energy E with the following equation:

$$E = \int VI \,\mathrm{d}t = V \int I \,\mathrm{d}t \quad [\mathrm{J}]$$

When using battery-powered devices, it is usual to use the electric charge Q, in milliampere hours, to express the battery life [124]. This quantity is equivalent to a voltage-independent energy, in joules per volt. It can be computed from the energy E by dividing it by the voltage V, or by simply integrating the current over time. This gives Q expressed in ampere seconds, it must thus be multiplied by $\frac{1000}{3600}$ to get the result in milliampere hours:

$$Q_{As} = \int I \,\mathrm{d}t \quad [\mathrm{As}]$$
$$Q_{mAh} = \frac{1000}{3600} Q_{As} = \frac{1000}{3600} \int I \,\mathrm{d}t \quad [\mathrm{mAh}] \tag{4.2}$$

By measuring the current flowing through the device, it is thus possible to compute the electric charge Q consumed by the device. Every measurement was repeated three times. Only one result for a specific measurement will be shown

at a time, to not clutter the graphs, but the full measurements are available in appendix B. On the graphs showing the electrical consumption when the device transmits, only the wake-up consumption is shown, i.e. the deep sleep current is not shown. The graphs and computation of Q were made with Python, by using the parsing script available in the `measurements-parser` directory of the thesis' repository: `https://github.com/fdekeers/lorawan-smart-lln/tree/main/measurements-parser`.

Most measurements relate to the electrical consumption of a LoRaWAN transmission. Those measurements always produce waveforms of a similar shape, composed of the same parts. To not clutter the graphs by indicating those parts on every graph, an example is shown on Figure 4.5.

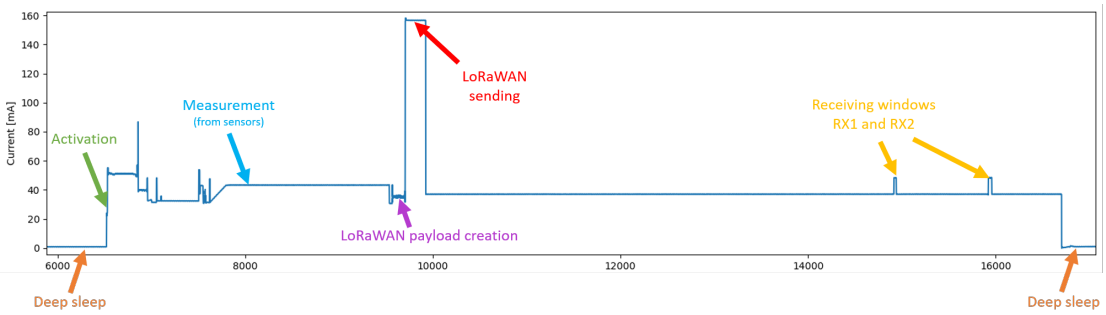


Figure 4.5: Example of a LoRaWAN transmission, with the different parts highlighted

For some LoRaWAN measurements, the RX2 receiving window is not present, because the device went to deep sleep directly after the first window. This is probably due to a communication error or a small bug, but this has no influence on the correct packet transmission, and can even be beneficial in our case, because we do not use downlink packets, and it saves a small amount of battery life. For Wi-Fi transmissions, the first parts are similar, but the transmission itself is obviously different, and much more complex, as it will be shown later.

4.3.1 Deep sleep

To be able to estimate the battery life of one of our devices, the first thing to evaluate is its energy consumption when it is in deep sleep. Our first sub-question is thus: "**What is the deep sleep energy consumption of a device ?**".

To answer this question, we simply measured the current flowing through the device when in deep sleep, which is shown on Figure 4.6. We first observe that the current seems periodic, with a period of 20 ms, i.e. a frequency of 50 Hz. This is certainly due to electrical interference with the power source, which was powered on by a 50 Hz, 230 V wall supply. Therefore, such oscillations would not occur when the device is powered on by a battery. We also observe the average value of the current, which is 0.8 mA. For the remaining of this section, we will consider the deep sleep current to be constant at 0.8 mA. The actual differences between the three measurements are small, and are probably due to small variations in the internal electrical contacts.

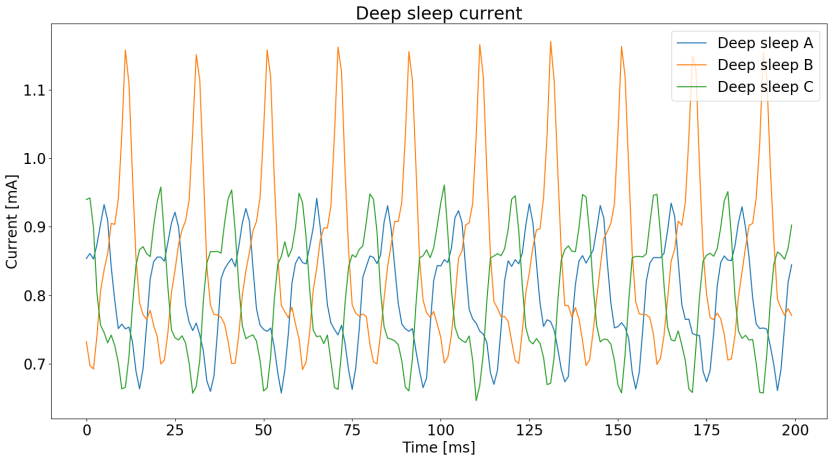


Figure 4.6: Deep sleep current measurements

We can compute the rate at which the battery loses its electric charge while the device is in deep sleep. Since the deep sleep current is 0.8 mA, to lose 1 mAh of charge in deep sleep, the battery will take $1/0.8 = 1.25$ h, which is equal to 1 hour and 15 minutes.

4.3.2 LoRaWAN vs Wi-Fi

For wireless transmission of packets, we rely on two different networks, LoRaWAN and Wi-Fi. The former is used in the general case, and the latter is used in the emergency case, when a value exceeds its emergency threshold. Throughout the thesis, we assumed that LoRaWAN was less power-hungry than Wi-Fi, and therefore chose to use it as the general case, to save battery. In this sub-section, we will evaluate this assumption, and answer the question "**Which is the less power-hungry**

network to use, between LoRaWAN and Wi-Fi ?". A similar question was evaluated by Klimiashvili *et al.* [86].

To this end, we measured the electric charge consumed by a device, when it transmits over LoRaWAN and over Wi-Fi. For those measurements, the device was configured to not interact with any sensor, and to send arbitrary data, to ensure that no electrical consumption related to an external sensor was involved. As such, the "measurement from sensor" part of the waveform shown on Figure 4.5 is absent. The messages, LoRaWAN or Wi-Fi, all contained one measurement, i.e. packet format V1 was used. To also measure the effect of Wi-Fi interferences on the transmission, the electrical consumption of Wi-Fi transmissions has been measured in two situations, differing by the Wi-Fi network utilization:

- low utilization, i.e. all other devices than the end device under test were disconnected from the access point;
- heavy utilization, i.e. two phones and laptops were connected to the access point, which were streaming YouTube videos in HD and downloading torrents.

The results are shown on Figure 4.7. The full measurements are available in appendix B.1: Figure B.1 shows the electrical consumption when using LoRaWAN, Figure B.2 when using Wi-Fi with a low network utilization, and Figure B.3 with a high utilization.

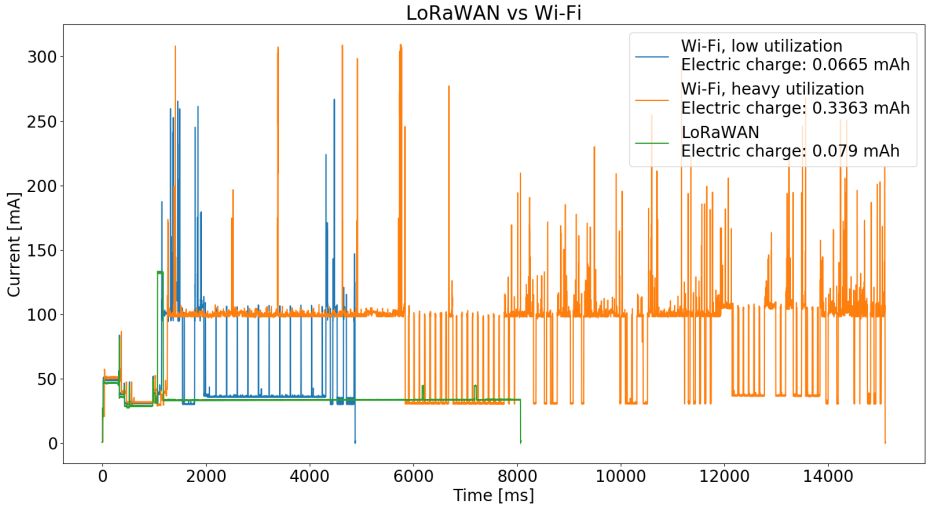


Figure 4.7: Electrical consumption - LoRaWAN vs Wi-Fi

We first observe that the first part of the device activation, i.e. setup and payload creation, is identical for all the cases, then the transmission itself differs. In the optimal case for Wi-Fi transmission, i.e. low network utilization, we observe that the electrical consumption of a LoRaWAN transmission is greater than the one of a Wi-Fi transmission (0.079 mAh and 0.0665 mAh, respectively), even if the LoRaWAN transmission peak is very small. This is due to the LoRaWAN receiving windows, taking place after the uplink transmission. As these receiving windows are part of the LoRaWAN class A specifications, the device must accommodate them, and thus stay awake until them, to comply with the specifications and take part in a LoRaWAN network.

However, we also observe that the electrical consumption of a Wi-Fi transmission might vary largely, depending of the network utilization. Whereas the consumption is indeed lower than LoRaWAN in the case of a low utilization, it becomes way higher in the case of a high utilization (0.3363 mAh). This increase in consumption is due to network interferences: when multiple communications use the same network, the electromagnetic waves may interfere [119], which introduces errors in the transmission. When using Wi-Fi, the communication protocol is HTTPS and thus TCP, the packets will therefore be retransmitted until no errors occur [125], which involves a higher electrical consumption. LoRaWAN transmissions are less impacted by interferences, thanks to the limited air-time and different orthogonal spreading factors, and are not retransmitted in case of errors, simply because the end device does not know an error has occurred. In practice, Wi-Fi networks are used by multiple devices at the same time, even more in the case of public networks. The optimal conditions can thus potentially never occur, which induces a Wi-Fi transmission cost always higher compared to LoRaWAN.

Besides, the infrastructural costs needed to use LoRaWAN are lower than the ones needed to use Wi-Fi. Indeed, as the range of LoRaWAN is way higher, and as the backbone network used, The Things Network, is open and free, one must simply purchase LoRa-enabled devices and configure them accordingly as well as the application. As gateways tend to be present in most major cities, at least in Europe [50], the transmitted packets can be forwarded by any gateway in the neighbourhood. When using Wi-Fi, one must dispose of an access point in the direct surroundings of each device, which requires a subscription to a network provider to enable Internet connectivity. Otherwise, one could rely on public Wi-Fi networks, but the configuration for such networks is more cumbersome, and is not guaranteed to be accommodated by every end device. Moreover, their utilization is higher, which results in a higher electrical cost for Wi-Fi transmissions.

With this in mind, we can conclude that, even if, in the optimal case, a Wi-Fi transmission is less power-hungry than a LoRaWAN transmission, LoRaWAN is still the best choice for the general case. Indeed, the optimal case for Wi-Fi may never occur in practice, and the infrastructural cost for LoRaWAN deployment is lower.

4.3.3 Sensors

Our end devices accommodate a variety of sensors, described in section 3.2.2. Those sensors are not equivalent in terms of energy consumption, and the MQ-135 and GPS have already been described as being power-hungry. In this section, we will evaluate this statement, by answering the question: "**Which sensors are the less power-hungry ?**".

We measured the electrical charge consumed by an end device during a measurement and a LoRaWAN transmission, using packet format V1, i.e. direct transmission, in four scenarios, which differed in the sensors connected to the device, and collecting environmental values. The sensors in the four cases were the following:

- BME680;
- BME680 and GPS;
- BME280 and MQ-135;
- BME280, MQ-135 and GPS.

Even if the connected sensors were different, the transmitted values were identical in each case, the only exception being that, in the case of a BME680, the air quality index was transmitted, replaced by the CO_2 concentration in the case of a MQ-135. The results are shown on Figure 4.8, and the full measurements for each scenario are available in appendix B.2.

We can first observe that the duration of the measurement is longer when the GPS or the MQ-135 are involved. This is due to the fact that a warm-up period is needed before reading the values of those two sensors, such that their values are accurate. During this period, the sensors are kept powered on, and the serial port of the GPS is prepared by repeatedly reading it, before reading the actual values. This period has been arbitrarily set to 10 seconds, as visible on the waveform.

Moreover, we can observe that the current flowing through the device is higher when using a MQ-135 instead of a BME680, or when adding a GPS module. This higher current, in conjunction with the aforementioned warm-up period, involves a higher electric charge consumption. This confirms our assumptions on the high

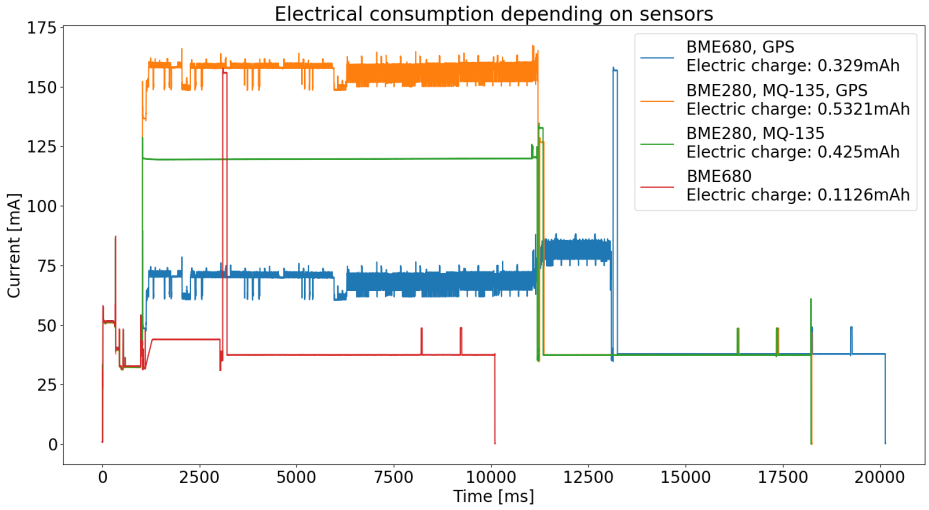


Figure 4.8: Electrical consumption depending on the connected sensors

energy consumption of those sensors. Since the values retrieved are similar, to optimize battery life, the best choice is therefore to use a BME680 sensor, instead of a BME280 with a MQ-135, to measure air quality. For geo-location data, it is more energy efficient to not use a GPS, and retrieve the device's location by using Wi-Fi, as this operation is only issued once when the device starts, contrarily to the GPS measurement that happens every time the device wakes up.

4.3.4 Benefits of batch sending

The decision of gathering multiple measurements inside a single packet, i.e. by using packet format V2 and V3, as described in section 3.3, was assuming that the LoRaWAN transmission is a power-hungry operation compared to the measurement only, and that the increased power used for the transmission of a larger packet was not significant compared to the saved power due to the reduced number of transmissions. In this section, we will evaluate these assumptions, and thus answer the question "**Is it more energy efficient to use direct or batch transmissions, and with which batch size ?**".

To this end, the electrical consumption of a measurement only, and of a measurement followed by a transmission, will be measured and compared, for all the possible batch sizes. For those measurements, the end device under test was connected to a BME680 sensor, from which it gathered values and transmitted

them over LoRaWAN, to simulate a real-world operation of the device. The values present in the payload were the following: battery, temperature, pressure, humidity, latitude, longitude, altitude, and air quality.

Batch vs direct sending

Figure 4.9 compares the electrical consumption of:

- in blue, a single measurement without sending anything (full measurements on Figure B.8);
- in orange, a single measurement followed by the transmission of a batch of size 4, using packet format V3 (full measurements on Figure B.9);
- in green, a measurement followed by the direct transmission of this single measurement, using packet format V1 (full measurements on Figure B.10).

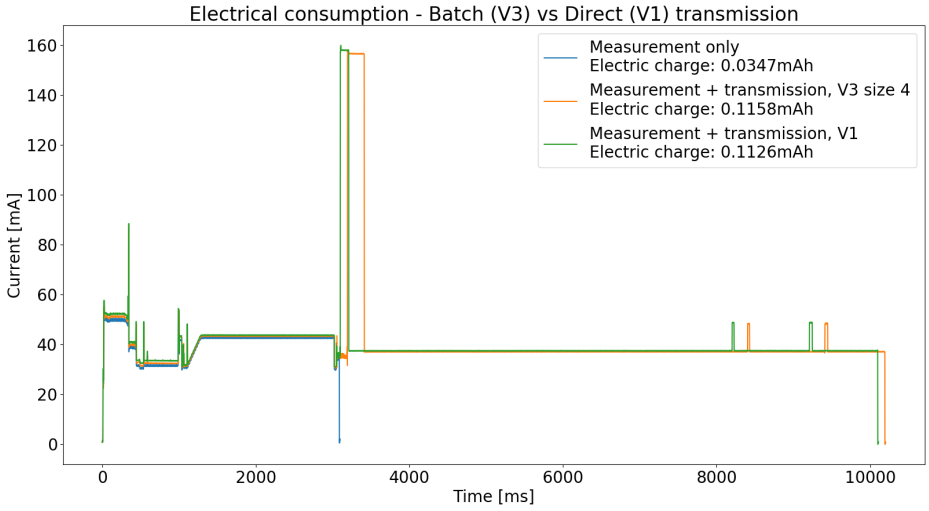


Figure 4.9: Electrical consumption - Batch (V3) vs Direct (V1) transmission

One can observe than the electrical consumption of the measurement only (0.0347 mAh) is less than a third of the electrical consumption of the measurement with the transmission. Additionally, a direct sending (containing a single measurement) consumes nearly as much energy as the measurement and the sending of a batch of size 4: 0.1126 mAh compared to 0.1158 mAh. The small difference comes from the longer time needed to create and send the payload containing the batch.

To compare both configurations, we will consider the situation where the device needs to send four measurements:

- **Direct sending**: the device wakes up four times and does the following: (1) measure, (2) send the measured value. Each activation consumes 0.1126 mAh. The transmission of the four measurements will consume 0.4504 mAh in total.
- **Batch sending**: the devices wakes up three times and only measures and stores the measured data. This operation consumes 0.0347 mAh per activation. The fourth time, the device measures the last data and sends the three previously measured values as well as this last one. This operation consumes 0.1158 mAh. The total consumption will be $3 \times 0.0347 + 0.1158 = 0.2199$ mAh

A similar observation can be made for every packet format and batch size. In general, batch transmission is thus much more energy efficient than direct transmission. The more measurements are batched, the more energy efficient a transmission is.

Batch size

As one can imagine, the more the batch size, i.e. the number of consecutive measurements batched in the packet, is increased, the more energy will be consumed by the transmission. Table 4.2 shows the payload size in bytes, depending on the batch size. As explained in section 2.2.1, a LoRaWAN payload is limited to 222 bytes, which limits the batch size to 5 for packet format V2, and 8 for V3. The maximum batch size is larger for V3 than for V2, as some values, i.e. latitude, longitude and altitude, are only present once for the whole batch, instead of in every measurement in the batch, which saves space in the payload.

Batch size	V2 Payload size [bytes]	V3 Payload size [bytes]
2	80	65
3	119	89
4	158	113
5	197	137
6	/	161
7	/	185
8	/	209

Table 4.2: Payload sizes as a function of the batch size

Figure 4.10 shows the mean and standard deviation of the electrical consumption of a measurement followed by a LoRaWAN transmission, based on the packet format

and the batch size. The full measurements data are shown in Table B.1 in appendix.

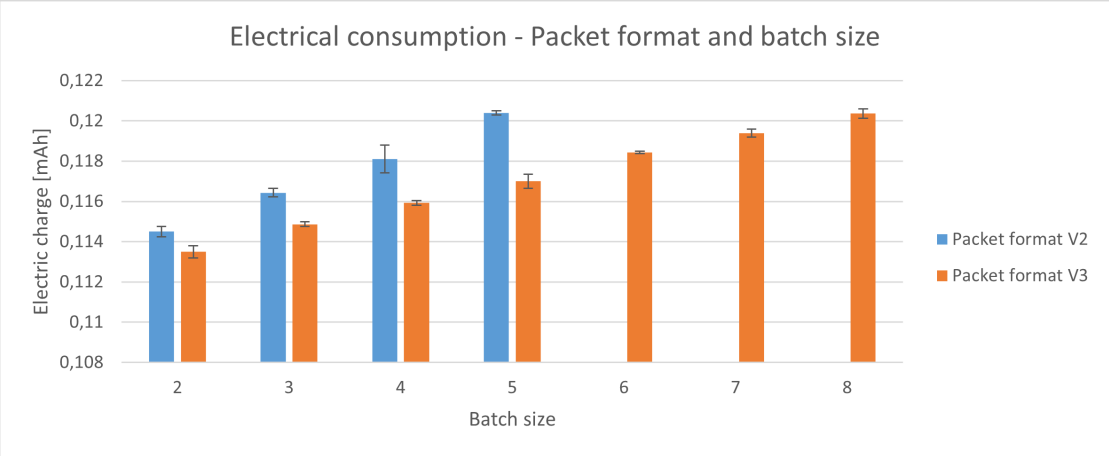


Figure 4.10: Energy consumption depending on the packet format and batch size

We can first observe than, as expected, the electrical consumption is always smaller with packet format V3 than with V2, for the same batch size, as the payload size is smaller. The graph shows that increasing the batch size from two to five, using packet format V2, only increases the consumed electric charge by 5% (0.0061 mAh). The same computation can be done between batch sizes two and eight using packet format V3; the increase is around 6% (0.007 mAh).

By increasing the batch size, we reduce the frequency of packet transmissions, which account for more than 2/3 of the electrical consumption of a device wake-up. In exchange, the electrical consumption of transmissions is only slightly increased. Therefore, we can conclude that, to minimize electrical consumption, and thus optimize battery life, we must use packet format V3 with a batch size of 8. However, in that case, transmissions only happen every eight device wake-ups, thus introducing a higher latency between the actual measurement and the time when it is received by the cloud application. A trade-off must thus be made, between higher battery life, and more frequent update of the cloud application.

4.3.5 ADR influence

As described in section 2.2.1, the LoRaWAN network has tunable parameters, that might modify the energy consumption depending on their configuration. Increasing the device class from A to B or C will increase the energy consumption, as measured

by San Cheong *et al.* [85], as the reception capabilities of end devices are enhanced. Besides, increasing the spreading factor also increases the energy consumption, as evaluated by Gupta and Fujinami [84]. However, enabling Adaptive Data Rate (ADR) or not has an unclear effect on the energy consumption. The objective of this sub-section is thus to **evaluate the influence of ADR on the energy consumption**.

We measured the electrical consumption of a device, for all the possible combinations of packet format version and batch size, with ADR enabled and disabled. The device was connected to a BME680, and retrieved and transmitted all its values. The mean and standard deviation of the results are shown on Figure 4.11, and the full results are shown in Table B.2 in appendix.

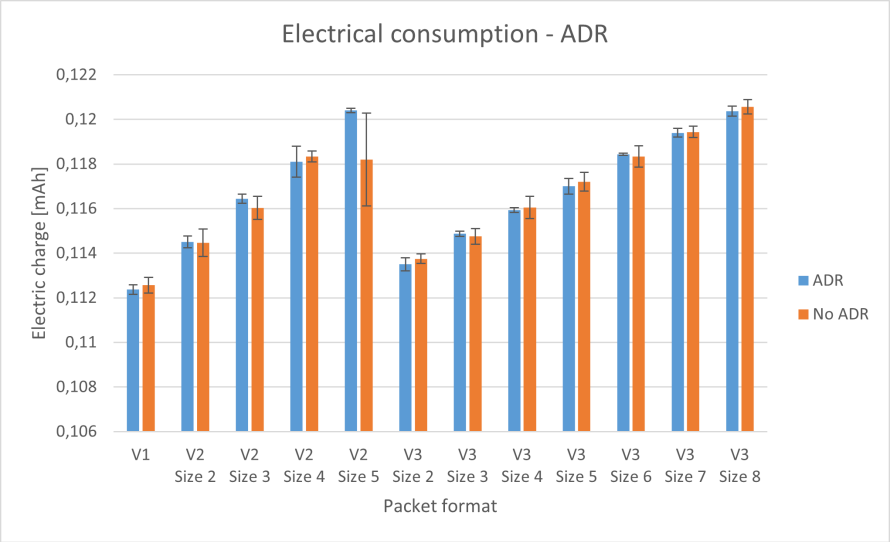


Figure 4.11: Comparison of the electrical consumption with and without ADR

We observe that the electrical consumption is nearly identical in both cases. The only exception comes from the batch of size 5, sent with packet format V2, but the high standard deviation indicates that the values were spread out, potentially due to an unexpected behaviour during this measurement. We can conclude that ADR has no impact on the electrical consumption of packet transmissions. The best choice for our devices is thus to activate it, as it might adapt the spreading factor of our devices to always use the lowest one, and thus minimize power consumption.

4.3.6 CPU frequency

The ESP32 board, used as microcontroller for the end devices, is a powerful IoT board, able to run at a CPU frequency of up to 240 MHz. We suppose that, the lower the CPU frequency, the lower the energy consumption. In this section, we will evaluate the **influence of the CPU frequency on the energy consumption** of a device. The CPU frequency of our end devices can be decreased down to 80 MHz. Below that number, the board will not be able to send data over Wi-Fi anymore.

We measured the electrical consumption of the device at different frequencies: 80, 160, and 240 MHz. The device was connected to a BME680, and used packet format V1, i.e. direct transmission. The results are shown on Figure 4.12. The full measurements are available in appendix B.5.

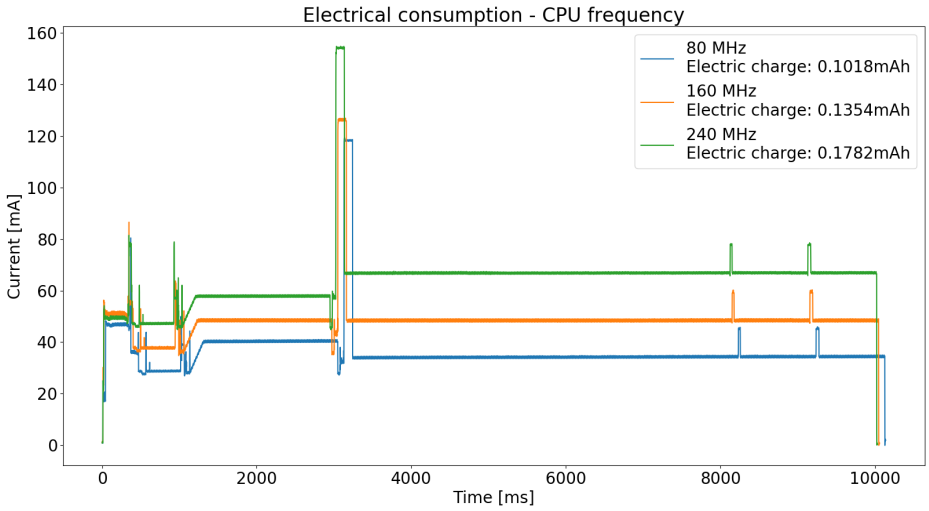


Figure 4.12: Electrical consumption depending on the CPU frequency

As expected, with a total electric charge of 0.1018 mAh, the lowest frequency of 80 MHz is more energy efficient than the two other frequencies, which require an electric charge of 0.1354 mAh and 0.1782 mAh, for the 160 MHz and the 240 MHz frequencies, respectively. In addition to that, running at a lower frequency does not have a significant impact on the execution time of the operations. Therefore, using the lowest frequency is the best solution for this application.

4.3.7 Battery life estimation

Finally, we can answer the initial question "**What is the battery life of one of our end devices ?**", and compute a battery life estimation, by applying the parameters we evaluated to be the most suited to our application. This computation uses the following assumptions:

- The device is powered by two 3.7 V, 1100 mAh batteries connected in parallel, which is equivalent to a single 3.7 V battery of 2200 mAh.
- The device wakes up to measure data every 10 minutes.
- The device is connected to a BME680 sensor.
- The device transmits data using packet format V3.
- The batch size is four measurements, meaning that data transmission occurs every 40 minutes. It allows frequent enough data on the dashboard while not compromising too much on the battery life and respecting the LoRaWAN sending frequency rules. Indeed, a V3 packet of size 4 is composed of 113 bytes, we can therefore, according to the air-time calculator [63], send one message every 605.3 seconds, i.e. approximately 10 minutes, in average.
- ADR was enabled.

Under those assumptions, as evaluated in the previous sections:

- A single measurement consumes **0.0347 mAh**.
- A measurement followed by a transmission of a batch of size 4 consumes **0.1158 mAh**.
- With a deep sleep current of 0.8 mA, when in deep sleep for 10 minutes (0.1667 hours), the device consumes $0.8 \times 0.1667 =$ **0.1333 mAh**.

It is also important to note that, at start-up, the device needs to retrieve its geo-location using Wi-Fi, perform a LoRaWAN join request, and setup some variables. This setup is only carried out once, and is thus a fixed one-time cost, which has been measured and is shown on Figure 4.13.

As this electric charge value of 0.2364 mAh is very low compared to the total battery capacity (0.01% of the battery capacity), this initial setup can be overlooked in the battery life estimation.

Each batch transmission (deep sleep periods + 3 measurements + 1 measurement followed by a transmission) consumes $4 \times 0.1333 + 3 \times 0.0347 + 0.1158 = 0.7531$ mAh. This corresponds to 40 minutes (0.6667 hours) of lifetime. The estimated battery life, t_{tot}, can be computed with the following equation:

$$\frac{Q_{tot}}{Q_1} = \frac{t_{tot}}{t_1} \Leftrightarrow t_{tot} = \frac{Q_{tot}}{Q_1} \times t_1$$

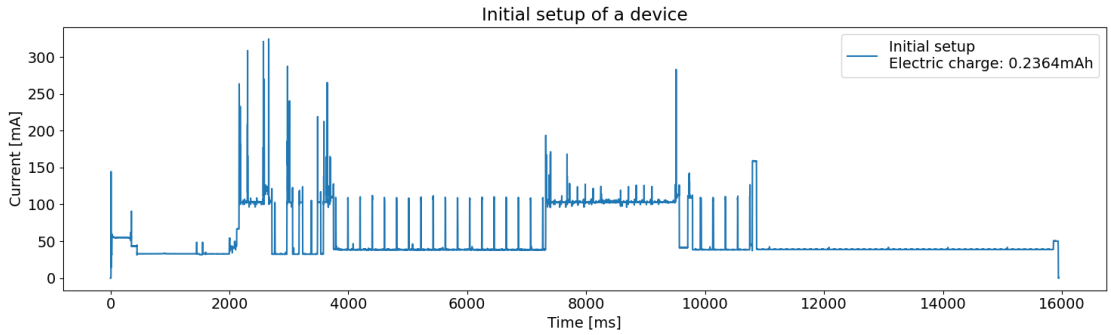


Figure 4.13: Initial setup - Electrical consumption

where Q_{tot} is the total electric charge of the battery, Q_1 is the electric charge consumed by the transmission of on batch, and t_1 is the duration taken by the transmission of one batch. With the two 1100 mAh LiPo batteries, the estimated battery life is thus equal to:

$$\frac{2 \times 1100}{0.7531} \times 0.6667 = 1947.5 \text{ hours} \approx \textbf{81 days}$$

This battery life, along with the choice of the board, will be discussed in section 5.3.

4.4 Transmission reliability

When using wireless transmissions, the reliability of the transmissions, i.e. the proportion of packets sent that are successfully received, can be an issue. Indeed, as wireless transmissions use electromagnetic waves, packet losses can occur, due to electromagnetic interferences, diffusion, or diffraction [119]. Therefore, we want to ensure the reliability of our wireless packet transmissions, using LoRaWAN and Wi-Fi. This reliability will be briefly discussed in this section, while answering the question "**Do the sent packets correctly reach their destination ?**".

Plenty of authors experimentally evaluated the reliability of the LoRaWAN network, e.g. Robin Schoenmaeckers for his master's thesis [74], Cattani *et al.* [77], and Sanchez-Iborra *et al.* [78]. As such, we will not go into the details, and simply assume that, as long as our devices are in the range of a LoRaWAN gateway, which can be a fixed gateway or one of our two mobile gateways, all the transmitted packets will be received, as we observed when deploying our devices.

However, as a reminder, The Things Network specifies a fair use policy, as described in section 2.3.4, that limits the uplink air-time to 30 seconds per device and per day. We observed that, when this fair use policy was not respected by one of our end devices, e.g. when sending a lot of messages for the energy consumption measurements, its packets were dropped by The Things Network, and thus were not forwarded to our cloud function.

As such, we must ensure that, for each device, the conjunction of the batch size and the wake-up period does not involves a higher air-time than 30 seconds per day. For instance, if packet format V3 is used with a batch size of 4, using the same packet content as the experiments with batch sending described in section 4.3.4, the payload size is 113 bytes. Using the LoRaWAN air-time calculator [63], we see that this packet will have an air-time of 210.2 ms. As such, the device may only send a packet every 605.3 seconds in average, which is slightly over 10 minutes. Therefore, since the device transmits a packet every four wake-ups, its wake-up period must be greater than or equal to three minutes to ensure the packets are not dropped.

Concerning Wi-Fi transmissions in case of emergency, as the transmission uses HTTPS and thus TCP, the reliability properties of the protocol [125], among others the retransmissions in case of errors, ensure that the packets are always correctly received by the receiver.

4.5 Real-world deployment

After having developed the software on a test setup using breadboards, the main experiment was to deploy the end devices in the city of Louvain-la-Neuve itself, to assess the **correct functioning of our system in a real-world deployment**. Therefore, the devices and sensors needed to be packaged in a way that allows them to remain outside in windy and rainy conditions, while still being able to collect environmental data accurately.

Custom PCBs have been soldered with all the needed hardware to be able to package it inside rain-resistant cases. The cases have been thought to be weather resistant and not water-proof. This means that the sensors have a clear access to the outside at the bottom of the case, while remaining out of reach for the rain. Figure 4.14 shows different packaged devices with their sensors. Figure 4.15 shows these closed cases as they are when deployed in the city and sending data.

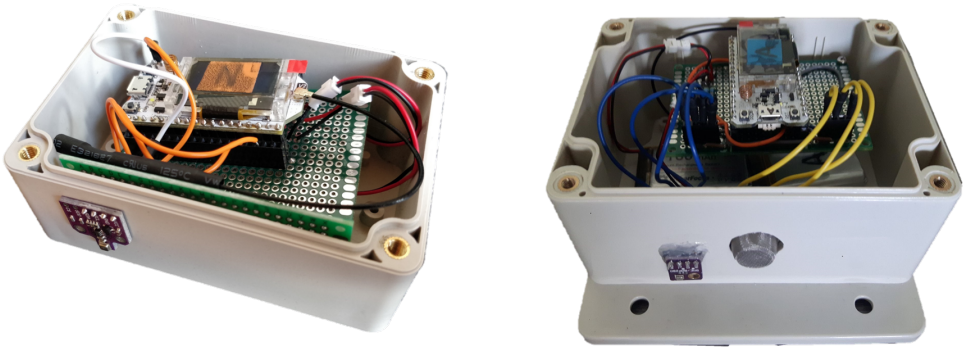


Figure 4.14: Custom PCBs packed inside weather-resistant cases

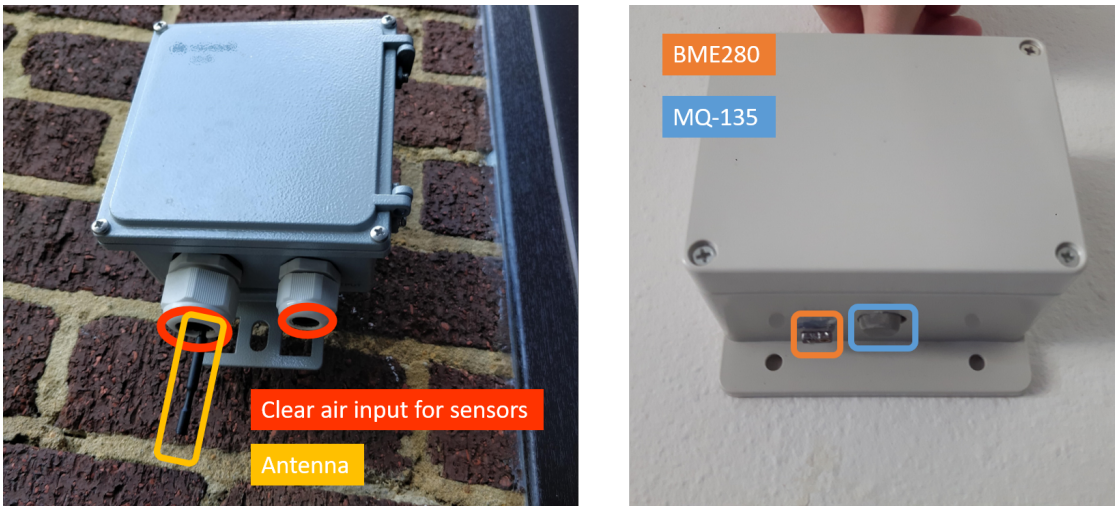


Figure 4.15: Packed devices inside weather-resistant cases as deployed in the city

Thanks to the three fixed LoRaWAN gateways present in Louvain-la-Neuve, installed by the university at the positions shown on Figure 3.9, there was an immediate access to the LoRaWAN network everywhere in the city. However, as those devices are using the hybrid model where they can send data over LoRa as well as over Wi-Fi, the Wi-Fi connectivity was an issue. When thinking about where to deploy those devices, it was important to think about the Wi-Fi connectivity as it is, by default, thought to be used inside. Therefore, we targeted balconies which have more chance to be close to an access point.

The access point's information (SSID and PSK) were provided by the balcony's owner (friends of ours). When thinking about larger scale deployment, that factor is important. An agreement could be made with the university to connect the devices to the *eduroam* network, which is available in the surroundings of the university buildings across the city. Section 5.3 will give some hints on how to improve the deployment of many devices to enable the collection of data in many locations.

To allow an hot swap of the batteries and increase the battery life, each PCB has two JST-1.25 connectors, allowing to connect two batteries in parallel.

We can assess the functioning of our deployed system, by observing the correct transmission of data packets on two interfaces: the "Live data" panel of The Things Network's application console, and the Grafana dashboard. The former shows all the LoRaWAN packets that are received by the gateways, and forwarded to our cloud function. Figure 4.16 shows this console panel, on which we can observe that the end devices' messages are correctly received and forwarded by TTN. The latter displays the last data received from every device, using maps, graphs, and gauges, as explained in section 2.5.3. Figure 3.11 depicts the dashboard, with data coming from the end devices, which confirms that the whole system is functioning properly.

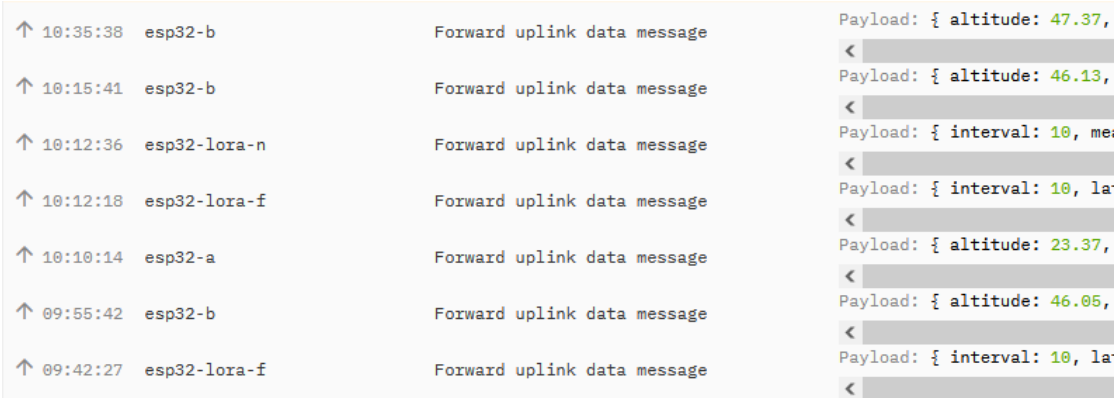


Figure 4.16: "Live data" panel of The Things Network's application console

Chapter 5

Discussion and improvements

In this chapter, we will discuss the qualities and drawbacks of our solution, on various aspects, namely the security of the end devices and transmission, the scaling of our system towards a large number of devices, and the energy consumption of the end devices. While discussing, we will also propose further improvements that could be applied. Finally, we will evaluate the cost related to the deployment of our system, taking into account the end devices and the cloud infrastructure.

5.1 Security

In applications such as this one, security can be a big concern, on multiple levels. The first one is that, as the packets issued by the end devices are sent wirelessly, either via LoRaWAN or Wi-Fi, anyone could potentially intercept them while sniffing the network traffic. Then, anybody could try to send spoofed packets, either LoRaWAN packets to our TTN application, or JSON packets to our cloud function. Finally, the URL of our cloud function could be redirected by an attacker, such that the packets never reach our actual cloud function. This section details the mechanisms implemented to overcome those issues, and also discusses the case of an attacker that has physical access to an end device. With our application, security threats are not really important, as the collected data is not critical nor secret. However, it is still interesting to secure the transmissions, for didactic purposes.

5.1.1 Encryption

When packets are sent wirelessly, it is easy for a potential attacker to sniff the traffic, and to read the content of the packets. The simple solution to prevent an attacker from reading our packets is to encrypt them before transmission.

Multiple transmissions between several pairs of transmitter/receiver happen during a successful transmission from an end device to the InfluxDB database:

- from the end device to the TTN gateways, using LoRaWAN;
- from the end device directly to the cloud function, using Wi-Fi;
- inside the The Things Network backbone to reach our TTN application;
- from our TTN application to our cloud function.

The traffic handled by The Things Network, i.e. from then end devices to the gateways using LoRaWAN, inside TTN backbone, and from TTN to our cloud function, is encrypted by TTN, as detailed in section 2.3.2. As a quick reminder, the LoRaWAN packet from an end device to the TTN server, and the HTTPS request from TTN to our cloud function, are encrypted with the `AppSKey`. Actually, the only traffic that is not protected by this encryption is the traffic sent directly from end devices to our cloud function, which occurs in case of emergency, and bypasses The Things Network. As our cloud function is an HTTPS endpoint, internet traffic toward its IP address is automatically encrypted using TLS, the encryption layer of HTTPS [126].

5.1.2 Authentication

An attacker could easily try to make our application display false data, by transmitting spoofed packets, either by using LoRaWAN or Wi-Fi. Both of those communication possibilities provide security mechanisms against spoofing, described hereafter.

The validity of the LoRaWAN traffic is ensured thanks to multiple keys: the `DevAddr`, and `NwkSKey`, as described in section 2.3.2, as well as the `AppEUI` when OTAA is used. The `DevAddr` and `NwkSKey` are used to identify and authentify the end device, and the `AppEUI` to identify the application able to process the join request [25, 52]. If a key is incorrect, or not specified, the traffic will be rejected, making it impossible to spoof packets without stealing the keys. When using OTAA, the `DevAddr` and `NwkSKey` are not provided as such in the device's software, but new ones are computed for each session during the join request, by using the `AppKey` [53]. This makes OTAA more secure, as even if an attacker manages to retrieve the keys, the device can simply be rebooted to compute and use new session keys. Therefore, we use OTAA with our system.

As described in section 3.5.1, JSON packets sent to the cloud function contain an authorization token, to authentify and validate the device that sent the packet, and drop it if the token is not known or has been revoked. This mechanism prevents attackers from sending spoofed JSON packets, in two ways:

- If the attacker sends a packet containing a random authorization token, the packet will be dropped because the token has not been signed using the application secret, and it thus not valid.
- If the attacker tampered with a valid token, the token can be revoked, such that the attacker's packets can be dropped even if the token was initially valid.

5.1.3 HTTPS certificates

To authenticate the server, i.e. ensure that the server to which the devices send the data is the correct one, HTTPS uses SSL/TLS certificates [127]. Such certificates are delivered to web servers by a Certificate Authority (CA), to authenticate themselves towards clients. A client can thus provide the server's certificate when connecting to it, or the certificate of one of its CAs, to ensure it is connected to the correct server. In our application, certificates are used in two cases:

- When the end device connects to Google Geolocation API to retrieve its geo-location data.
- When the end device transmits its data packet directly to the cloud function via Wi-Fi.

In both cases, the TLS certificate is provided by the end device when the HTTPS connection is set up, and checked against the server's certificate, to ensure the server's identity. However, the provided certificate is not identical in both cases, as the servers are different.

5.1.4 Physical access to the device

As our end devices are deployed outdoors in the city, an attacker could potentially steal a device and have physical access to it. In this case, they could connect the device to a computer via an USB cable, and read the serial monitor that prints debug information, e.g. the data values read, and the hexadecimal payload, but this information is not critical.

However, sensitive information, e.g. the Wi-Fi password, Google API key, or cloud function authorization token, must be provided into the Arduino sketch and loaded into the device's flash memory. Whereas it is not possible to retrieve the Arduino source code from a running device, because it is compiled before being uploaded, the binary code could be retrieved, provided the appropriate tool, which is `esptool` in the case of an ESP32 board [128]. It is then possible to reverse engineer the binary code, but this is a tedious task.

To overcome this, the ESP boards provide flash memory encryption, which prevents the flash memory of being read [129]. This can be enabled by setting a fuse value to 1, but this is impossible to undo. Flash memory encryption can prevent an attacker from retrieving the binary code of a device, and therefore to dispose of sensitive passwords or keys.

5.2 Scaling

Our application currently accommodates only five devices. However, when applied extensively to a whole city, IoT devices can be present in much larger numbers, which poses scaling issues. In this section, we will discuss the accommodation of a large number of devices to our application.

5.2.1 Message limit

As mentioned in section 2.3.4, the number of messages is limited by The Things Network, to ensure a fair use of the network. As this limit is applied **per end device** and not per application or gateway, it poses no scaling problem, and our TTN application could potentially accommodate an arbitrarily large number of end devices.

5.2.2 Automatic generation of sketches

Every end device must be programmed by an Arduino **sketch**, which contains the device's software. To make the handling of an extensive number of devices easy, the device-specific part of each device's sketch must be as short and easily configurable as possible. This is enabled by our custom Arduino libraries `EspDevice`, `Packet`, and `WifiSender`, which contain all the programming logic for the common behaviour and parameters of end devices. As such, a specific end device's sketch must only contain the following information:

- Generic configuration parameters: wake-up period, number of measurements batched in a single packet, version of the packet transmission protocol.
- Wi-Fi parameters: SSID, password, Google API key for Google Geolocation API, webhook URL and authorization token.
- TTN parameters: Heltec license, `DevEUI`, `AppEUI`, `AppKey`.
- Connected sensors, along with the ESP32 pins used for their electrical connections, and their emergency thresholds.

To ease the creation of such sketches for each device, and thus enable accommodation for a large number of devices, we developed a Python script to generate end

devices sketches automatically, based on YAML configuration files, which contain the aforementioned information in a structured manner. An example device YAML configuration file is shown in appendix A.1. The script takes the name of the YAML file as a command line argument, and produces a complete Arduino sketch for the device, using the parameters specified into the file.

To do this, the script uses template files for each of the sketch sections, e.g. Wi-Fi or TTN parameters, sensors constants, `setup` and `loop` functions, and fills them in with data from the YAML file. To run the script, Python version 3+, and the packages `sys`, `os`, `re`, and `PyYAML` are needed. The three first are already present in the basic library, and `PyYAML` can be installed with the command `pip install pyyaml`. When all the packages have been installed, run the following command in the script directory:

```
$ python3 generate-sketch.py YAML_FILE
```

The resulting Arduino sketch will be created in the directory `arduino/devices/DEVICE_NAME`. The generated sketch corresponding to the example configuration file is shown in appendix A.2. The script, along with the necessary template files, and example YAML configuration files, is available in the `generate-sketch` directory of the thesis' repository: `https://github.com/fdekeers/lorawan-smart-lln/tree/main/generate-sketch`.

5.2.3 Cloud functions

By design, the cloud function representing the entry point of the backend infrastructure is automatically scaled based on the amount of queries. Assuming an important data rate, the cloud provider will automatically adapt to handle those requests.

It is important to keep in mind that the data of a single device are being sent every 30 minutes to one hour. It means that if 200 devices were doing so, data would trigger the cloud function once every 10 to 20 seconds on average, which is reasonable for an endpoint.

5.2.4 Databases

The scaling of databases can be done in two directions:

- **Vertical**: consists of adding more resources to the host or, in the case of databases, adding more storage to store more data.

- **Horizontal**: consists of splitting and/or replicating data across multiple hosts. It can be useful to geo-replicate the data and reduce their access time when being accessed from a distant location. Another advantage of horizontal scaling is that it provides redundancy. If one of the storage hosts is down, most of the data – if not all, depending on the configuration – can still remain accessible.

Both solutions can be used at the same time. However, their added-value depends on the requirements, the context and the used database.

InfluxDB allows horizontal scaling of the storage. Thanks to their *shards* and *shard groups*, the buckets containing the data can be spread across multiple hosts.

This thesis focuses on Louvain-la-Neuve which is very local. It means the data will not be accessed from all over the world. However, when adding more devices the amount of data will increase rapidly. To support such an increase, it is interesting to consider both scaling solutions for the already mentioned benefits (i.e. more storage means more data and more hosts means a better resiliency).

5.3 End devices energy consumption

In this thesis, we explored the usage of hybrid devices which are Wi-Fi and LoRa enabled, by using the WiFi LoRa 32 from Heltec, based on the ESP32 board, as microcontroller. This MCU has the advantages of being very standard and well-known, thus involving a small price, an easy accessibility, and numerous Arduino libraries to program the board with. This board provides many capabilities, such as multiple wireless interfaces and communication protocols to be used with external devices, as described in section 3.2.1.

However, having such capabilities comes to a cost. As discussed in section 4.3.1, the deep sleep current is about 800 µA. That number is low for Wi-Fi enabled devices, but very high for LoRa standards, where the target deep sleep current of long lived battery powered device should be below 50 µA, to reach a multiple years battery life when powered by a regular battery. Using those powerful ESP32 boards might not be the best solution for all nodes. In this section, we will provide some insights on ways to further optimize the battery life of our end devices.

5.3.1 LoRa only devices

An improvement of the deployed solution could be using both WiFi LoRa 32 and LoRa only devices. An example board can be the CubeCell dev-board [130]. With

a deep sleep current of about 3.5 µA, this device is better suited for long lived applications, by making the compromise of having no Wi-Fi module, less flash memory (128 kB compared to 4 MB) and a slow CPU. The two latter are not a problem, but the lack of Wi-Fi means that the devices can not retrieve their geo-location, and send emergency data, via Wi-Fi.

Using the low power CubeCell boards in most places to get a maximum of data points in addition to some ESP32s at some key locations where the Wi-Fi emergency feature is required could be a great solution. The geo-location of the CubeCell board can be hard-coded on the device, or directly in Firebase, as they can not retrieve their own geo-location via Wi-Fi, and as a GPS module is power-hungry. This solution would allow a better monitoring of the global city without having to compromise with the battery.

5.3.2 Energy production

When using devices and sensors which are meant to be used outside, it is always interesting to take advantage of the weather. The CubeCell board, which has been presented in the previous section, has specific pin headers for solar panels. Such panels can provide extra energy to the battery during the day and thus, extend the device's battery life.

Another power source can be the wind. Using wind turbines, the devices could extend their battery life as well as measure the wind speed. This kind of data, in addition to the values already collected by our system, i.e. temperature, pressure and humidity, can be interesting for a weather forecast application.

If designed and optimized correctly, thanks to those power sources, the devices might even become completely autonomous and thus, live forever without having to worry about recharging or changing the batteries, therefore minimizing maintenance costs.

5.3.3 Battery monitoring

Section 3.2.3 detailed the battery monitoring system. This system continuously draws a small amount of current, because of the closed circuit voltage divider. This current has been computed in section 4.2.2 and is equal to 4.468 µA. It is negligible, but an improvement could be applied to the system such that current only flows when the battery voltage is read. This improvement involves the use of a NPN transistor. The improved system's electrical schematic is shown on Figure 5.1.

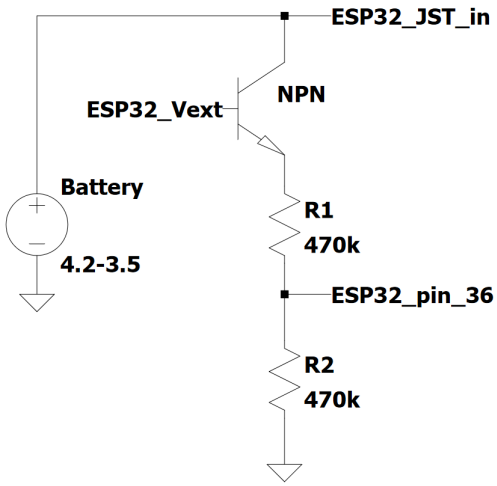


Figure 5.1: Improved battery monitoring system, using a NPN transistor

A NPN transistor acts as a voltage controlled switch, i.e. it only allows the current through when the voltage on its base is high. The base can be connected to the Vext output of the ESP32, that, as a reminder, is a programmable voltage output already used to power the external sensors. In this way, the current would flow into the system only when the battery voltage must be read, effectively minimizing the electrical consumption of the device.

5.4 Adaptive batch size

The actual batch size used by a device must be specified in its Arduino sketch. As explained in section 4.3.4, to comply with the LoRaWAN payload size limit of 222 bytes, this batch size is limited, to five for packet format V2, and eight for packet format V3. This maximum is a fixed lower bound, to accommodate the packets when all the possible sensors are connected to the device. However, if not all sensors are connected, the packets will only contain a limited amount of different values. In that case, the batch size might exceed the limit of five or eight, while still being compliant with the LoRaWAN payload size limit.

To address this issue, instead of being fixed statically with a constant in the software, the batch size could be dynamically adapted, as follows. When the sensors are added to the device in the software, the total number of bytes taken by their values could be computed. Then, the batch size could be adapted, to always have the maximum number of batches possible in one payload. The more sensors are connected, the smaller the batch size becomes. On top of that, to ensure that

packets are still transmitted often enough, a maximum sending period can be specified, so that the batch size is never increased higher than the threshold where this period would be exceeded.

5.5 Limited storage

To store the environmental data, we use an InfluxDB database. As time passes, this database will become increasingly populated, potentially reaching its maximum capacity someday. When dealing with real-time data, it is important to think about how these data can be deleted to restore storage space. Deleting the oldest data might look like a good solution, but this retention policy does not allow the comparison of very old data with newer ones, as the oldest ones might already have been deleted. In some cases, old data can still be relevant.

A solution might be to aggregate multiple old data points in one, by only keeping the mean of their values. Points which are older than one year could be aggregated on an hourly basis, the ones older than one and a half year could be aggregated on a four hours basis, etc. This solution allows the data to be kept for a long time, and is a good compromise between information loss and storage usage.

5.6 Further data processing

The current system allows the gathering, storing, and visualization of environmental data. In addition to that, InfluxDB exposes an API allowing anyone with a token and associated rights to consume it. This API enables the usage of prediction algorithms or any other data manipulation technique. Frequent meteorological data points, like the ones produced by this system, can be useful for other applications which require more further data processing, such as a weather prediction system on a local basis.

5.7 Cost of the solution

As the solution involves hardware as well as servers and services running in the cloud, it comes with a certain fixed cost per device, in addition to recurring costs for the infrastructure (virtual machines, cloud functions, Firebase, and Google Geolocation API). The remaining of this section will discuss the different involved costs. Note that all the prices are estimations because of the frequent variations of the prices.

5.7.1 Hardware

The devices used for the deployment, as shown on Figure 4.14, all contain the same base hardware, which is:

- one weather-resistant case
- one WiFi LoRa 32 microcontroller
- two 1100 mAh LiPo batteries
- Some cables and PCB boards

In addition to that, they each have their own set of sensors. Table 5.1 shows the estimated price of each component used in the system.

Component	Price per unit
ESP32 with LoRa [89]	25€
1100 mAh battery [131]	6€
Plastic case [132]	20€
PCBs and cables	3€
BME280 [96]	8€
BME680 [99]	17€
MQ-135 [98]	7€
Noise sensor [101]	5€
GPS [102]	16€

Table 5.1: Estimated prices of the components

Considering those values, a device equipped with the base components and a BME680 sensor (temperature, pressure, humidity and air quality sensor), like the leftmost one on Figure 4.15, costs around 77€. The rightmost one, with a BME280 (temperature, pressure and humidity sensor) and a MQ-135 (CO_2 sensor) costs around 75€. The Cubecell LoRa node [130], presented earlier in section 5.3.1, has a price which is similar to the WiFi Lora 32, and is around 22€. A way to reduce the overall price of the solution could be by purchasing sensors and components for a large amount of devices at once, which is feasible if the system would be deployed extensively in a whole urban area.

5.7.2 Cloud backend and infrastructure

To support and handle the collected data, we use the following cloud services:

- The Things Network community edition: this service is supported by the community and is free to use.

- Azure cloud functions: the pricing model is based on the amount of runs, considering the first one million activations of each month are free [133]. As the amount of activations per device is around 1400 per month, the free tier should not be passed anytime soon, and this service is thus in practice free at our scale.
- Firebase Firestore: this database system has a free tier allowing us to perform 50,000 reads and 20,000 writes each day [134]. These limits should not be surpassed as, for each device, the amount of reads is around 150 per day.
- Azure virtual machine: the virtual machine hosting the InfluxDB database and the Grafana instance have a cost of around 36€ per month, to which 3€ must be added for the network connectivity, and 6€ for the 30 GB of disk storage [135]. This sums up to around 45€ per month.
- Google Geolocation API: this service is normally billed at 5$ for 1000 requests [136]. However, 200$ are offered each month, which correspond to 40,000 requests. Since we only dispose of five devices, that will only send one request at startup, those 40,000 requests will never be reached, the service being thus in practice free for our application, at our scale.

The monthly cost of the backend infrastructure comes to a total of 45€. However, if the system would be deployed at the scale of a whole city, the free tiers would probably be exceeded, and the free services would become billed. Nevertheless, this should not be a problem if the system is funded by a city.

Chapter 6

Conclusion

Smart cities are going to be increasingly present all around the globe, by leveraging the potential of the Internet of Things, that allows nearly every object to become autonomous and connected. With this master's thesis, we applied those concepts to the city of Louvain-la-Neuve, by developing and evaluating a system for the monitoring of the city's environmental data.

Our system is composed of battery-powered end devices, measuring the data in different locations in the city, and transmitting them wirelessly over a Low-Power Wide-Area Network, a network specifically intended for power and resource-constrained devices. The network used in our case is LoRaWAN, one of the most popular LP-WANs, as it is an open network that any user can freely use for its application. The packets are then forwarded by a gateway network server, The Things Network, to our cloud application, that uses InfluxDB, a time series database, to store the data, and Grafana, a monitoring platform, to display them. The monitoring interface is publicly accessible via a web browser, such that every citizen can access it and benefit from the data. We leveraged the hybrid network interfaces of our end devices, by transmitting the packets via Wi-Fi instead of LoRaWAN, in the case of an emergency.

One of the most important issues, when dealing with IoT devices, is their battery. We first designed a system for the devices to be able to monitor their own battery, such that one knows when it must be recharged. To predict and optimize the battery life, we evaluated the energy consumption of our devices, depending on various parameters. We concluded that our devices must send their data in batches, use a BME680 sensor instead of a MQ-135 and no GPS, run at the lower CPU frequency possible, and that power consumption was independent of ADR. Furthermore, even if a Wi-Fi transmission might consume less energy in an optimal case, we argue that this case may never occur in practice, and that

the infrastructural advantages of LoRaWAN outweigh the potential battery life improvement. With these parameters applied, we estimated the battery life of a device with two batteries to be of 81 days, and we deployed five end devices outdoors in the city. Finally, we discussed the cost of our system, as well as its advantages and flaws, on the aspects of security, scaling, and battery life, and proposed potential improvements and further optimizations.

With this thesis, the objective was to use standard, off-the-shelf devices, to develop our system. After having evaluated the solution, mainly on the aspect of energy consumption and battery life, we observed that those devices, while simple to purchase and use, may not be the best alternative for autonomous, battery-powered devices, as their energy consumption is relatively high. In the case of a real-world deployment of such an application, a better option would be to use devices specifically designed and optimized for their usage. However, those devices come at a greater cost, which is not an issue for a large scale deployment funded by a city, but is for a master's thesis with limited budget.

Besides, the devices we deployed outdoors were still at a prototyping stage, as they are not completely resilient to bad weather conditions. A production-level deployment of such a system would need completely waterproof end devices. To this end, a mechanism to allow the sensors to measure the ambient air conditions would be needed, by using waterproof air inlets on the device cases, which would also increase the cost of our solution. Moreover, the objective of this master's thesis was not to develop a production-level system, but to learn and show how hybrid networks can be used in the case of a smart city.

Finally, the devices would have to be deployed in safe locations, out of reach of the citizens, e.g. on the university buildings, such that the devices' physical integrity is not compromised. This would involve more infrastructure and manpower, and the end devices must be ensured to have access the *eduroam* Wi-Fi network, to be able to use their Wi-Fi capabilities.

The application we developed, i.e. the smart monitoring of the city's environmental data, is a quite simple application, that was chosen to showcase the possibilities offered by the Internet of Things, Low-Power Wide-Area Networks, and cloud computing in the context of a smart city. This opens a wide array of possibilities. The collected data could be processed further, by using prediction models, to forecast the weather conditions in the next days. The list of measured data could also be expanded, by adding new sensors to the devices, or combining the current data with devices providing other measuring and networking capabilities,

such as crowd and traffic monitoring, to provide more precise monitoring of all the aspects of a living city. This would of course involve an upgrade of the cloud backend as well, to accommodate the new types of data.

This thesis allowed to embrace the capabilities of LP-WANs and IoT, in conjunction with cloud applications, to introduce the concept of smart cities into an existing city. We believe that this work can be taken further, by extending the current functionalities to various different applications, to transform Louvain-la-Neuve into a real "digital" city.

Bibliography

[1] John Kosowatz. Top 10 Growing Smart Cities. *The American Society of Mechanical Engineers (ASME)*, 3 February 2020. `https://www.asme.org/topics-resources/content/top-10-growing-smart-cities`. Accessed: 27/05/2021.

[2] LoRa Alliance. LoRaWAN - What is it ? A technical overview of LoRa and LoRaWAN. `https://lora-alliance.org/wp-content/uploads/2020/11/what-is-lorawan.pdf`. Accessed: 19/05/2021.

[3] Felix Wortmann and Kristina Flüchter. Internet of things. *Business & Information Systems Engineering*, 57(3):221–224, 2015.

[4] Gerd Kortuem, Fahim Kawsar, Vasughi Sundramoorthy, and Daniel Fitton. Smart objects as building blocks for the internet of things. *IEEE Internet Computing*, 14(1):44–51, 2009.

[5] Katrien Valkiers. L'histoire de l'IoT - L'Internet des Objets. *Siemens*, 9 November 2020. `https://magazine.siemens.be/fr/iot`. Accessed: 25/04/2021.

[6] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Sensing as a service model for smart cities supported by internet of things. *Transactions on emerging telecommunications technologies*, 25(1):81–93, 2014.

[7] Colin Harrison and Ian Abbott Donnelly. A theory of smart cities. In *Proceedings of the 55th Annual Meeting of the ISSS-2011, Hull, UK*, 2011.

[8] See Liang Foo and Gary Pan. Singapore's vision of a smart nation. 2016.

[9] Fachmin Folianto, Yong Sheng Low, and Wai Leong Yeow. Smartbin: Smart waste management system. In *2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–2. IEEE, 2015.

[10] Mohammad Aazam, Marc St-Hilaire, Chung-Horng Lung, and Ioannis Lambadaris. Cloud-based smart waste management for smart cities. In *2016 IEEE 21st International Workshop on Computer Aided Modelling and Design of Communication Links and Networks (CAMAD)*, pages 188–193. IEEE, 2016.

[11] Abhirup Khanna and Rishi Anand. IoT based smart parking system. In *2016 International Conference on Internet of Things and Applications (IOTA)*, pages 266–270. IEEE, 2016.

[12] Københavns Kommune. Smart parking. `https://urbandevelopmentcph.kk.dk/artikel/smart-parking`. Accessed: 08/06/2021.

[13] Patan Rizwan, K Suresh, and M Rajasekhara Babu. Real-time smart traffic management system for smart cities by using Internet of Things and big data. In *2016 international conference on emerging technological trends (ICETT)*, pages 1–7. IEEE, 2016.

[14] Anurag Kanungo, Ayush Sharma, and Chetan Singla. Smart traffic lights switching and traffic density calculation using video processing. In *2014 recent advances in Engineering and computational sciences (RAECS)*, pages 1–6. IEEE, 2014.

[15] Bilal Ghazal, Khaled ElKhatib, Khaled Chahine, and Mohamad Kherfan. Smart traffic light control system. In *2016 third international conference on electrical, electronics, computer engineering and their applications (EECEA)*, pages 140–145. IEEE, 2016.

[16] Seth Solomonow and Nicholas Mosquera. NYC DOT Announces Expansion of Midtown Congestion Management System, Receives National Transportation Award. *New York City Department of Transportation (NYC DOT)*, 5 June 2012. `https://www1.nyc.gov/html/dot/html/pr2012/pr12_25.shtml`. Accessed: 08/06/2021.

[17] Siva R.K. Narla. The evolution of connected vehicle technology: From smart drivers to smart cars to... self-driving cars. *Ite Journal*, 83(7):22–26, 2013.

[18] Roy Want. Near field communication. *IEEE Pervasive Computing*, 10(3):4–7, 2011.

[19] Francisco Borrego-Jaraba, Irene Luque Ruiz, and Miguel Angel Gómez-Nieto. A NFC-based pervasive solution for city touristic surfing. *Personal and ubiquitous Computing*, 15(7):731–742, 2011.

[20] Alessandra Basili, Walter Liguori, and Federica Palumbo. NFC smart tourist card: Combining mobile and contactless technologies towards a smart tourist experience. In *2014 IEEE 23rd International WETICE Conference*, pages 249–254. IEEE, 2014.

[21] Kim Boes, Larissa Borde, and Roman Egger. The acceptance of NFC smart posters in tourism. In *Information and Communication Technologies in Tourism 2015*, pages 435–447. Springer, 2015.

[22] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. Low power wide area networks: An overview. *IEEE Communications Surveys & Tutorials*, 19(2):855–873, 2017.

[23] Yonghua Song, Jin Lin, Ming Tang, and Shufeng Dong. An Internet of energy things based on wireless LPWAN. *Engineering*, 3(4):460–466, 2017.

[24] Kais Mekki, Eddy Bajic, Frederic Chaxel, and Fernand Meyer. A comparative study of LPWAN technologies for large-scale IoT deployment. *ICT express*, 5(1):1–7, 2019.

[25] Jetmir Haxhibeqiri, Eli De Poorter, Ingrid Moerman, and Jeroen Hoebeke. A survey of LoRaWAN for IoT: From technology to application. *Sensors*, 18(11):3995, 2018.

[26] LoRa Alliance. Home page. `https://lora-alliance.org/`. Accessed: 19/05/2021.

[27] Hubert Zimmermann. OSI reference model-the ISO model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.

[28] Olivier B. A. Seller and Nicolas Sornin. Low power long range transmitter, February 2 2016. US Patent 9,252,834.

[29] Martin Bor, John Edward Vidler, and Utz Roedig. LoRa for the Internet of Things. 2016.

[30] The Things Network. LoRaWAN - LoRaWAN Architecture. `https://www.thethingsnetwork.org/docs/lorawan/architecture`. Accessed: 19/05/2021.

[31] The Things Network. Network - Network Architecture. `https://www.thethingsnetwork.org/docs/network/architecture`. Accessed: 19/05/2021.

[32] LoRa Alliance. LoRaWAN 1.0.2 Regional Parameters. https://lora-alliance.org/wp-content/uploads/2020/11/lorawan_regional_parameters_v1.0.2_final_1944_1.pdf. Accessed: 19/05/2021.

[33] European Telecommunications Standards Institute. ETSI EN 300 220-2 V3.2.1 - Short Range Devices (SRD) operating in the frequency range 25 MHz to 1 000 MHz; Part 2: Harmonised Standard for access to radio spectrum for non specific radio equipment. https://www.etsi.org/deliver/etsi_en/300200_300299/30022002/03.02.01_60/en_30022002v030201p.pdf. Accessed: 19/05/2021.

[34] LoRa Alliance. LoRaWAN Specification. https://lora-alliance.org/wp-content/uploads/2020/11/lorawan1_0_2-20161012_1398_1.pdf. Accessed: 19/05/2021.

[35] G. Zhu, C. Liao, M. Suzuki, Y. Narusue, and H. Morikawa. Evaluation of LoRa receiver performance under co-technology interference. In *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–7, 2018.

[36] Semtech. Understanding ADR - LoRa Developer Portal. https://lora-developers.semtech.com/library/tech-papers-and-guides/understanding-adr. Accessed: 22/04/2021.

[37] The Things Network. Adaptive Data Rate. https://www.thethingsnetwork.org/docs/lorawan/adaptive-data-rate. Accessed: 22/04/2021.

[38] Sigfox. Sigfox Technology. https://www.sigfox.com/en/what-sigfox/technology. Accessed: 20/05/2021.

[39] Carles Gomez, Juan Carlos Veras, Rafael Vidal, Lluís Casals, and Josep Paradells. A sigfox energy consumption model. *Sensors*, 19(3):681, 2019.

[40] Mads Lauridsen, Huan Nguyen, Benny Vejlgaard, István Z Kovács, Preben Mogensen, and Mads Sorensen. Coverage comparison of GPRS, NB-IoT, LoRa, and SigFox in a 7800 km^2 area. In *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, pages 1–5. IEEE, 2017.

[41] Brian Ray. SigFox Vs. LoRa: A Comparison Between Technologies & Business Models. *Link Labs*, 31 May 2018. https://www.link-labs.com/blog/sigfox-vs-lora. Accessed: 20/05/2021.

[42] Sigfox. Buy Sigfox connectivity for your IoT devices. https://buy.sigfox.com. Accessed: 20/05/2021.

[43] Sigfox. Sigfox Cloud Integration. https://build.sigfox.com/backend-callbacks-and-api. Accessed: 26/05/2021.

[44] Rapeepat Ratasuk, Benny Vejlgaard, Nitin Mangalvedhe, and Amitava Ghosh. NB-IoT system for M2M communication. In *2016 IEEE wireless communications and networking conference*, pages 1–5. IEEE, 2016.

[45] Rashmi Sharan Sinha, Yiqiao Wei, and Seung-Hoon Hwang. A survey on LPWA technology: LoRa and NB-IoT. *Ict Express*, 3(1):14–21, 2017.

[46] Orange. Internet of Things - Changing the world, one thing at a time. https://business.orange.be/en/it-solutions/internet-things. Accessed: 20/05/2021.

[47] Orange. Internet of Things - Products and Services. https://business.orange.be/en/it-solutions/internet-things/products-and-services-internet-things-live-objects. Accessed: 26/05/2021.

[48] Orange. Maker the Cloud Instance for Prototyping. https://orange-iotshop.allthingstalk.com/product/orange-maker. Accessed: 26/05/2021.

[49] AllThingsTalk. AllThingsTalk IoT Platform Documentation. https://docs.allthingstalk.com/. Accessed: 26/05/2021.

[50] The Things Network. Home page. https://www.thethingsnetwork.org. Accessed: 21/04/2021.

[51] The Things Network. Gateways. https://www.thethingsnetwork.org/docs/gateways. Accessed: 28/05/2021.

[52] The Things Network. LoRaWAN - Security. https://www.thethingsnetwork.org/docs/lorawan/security. Accessed: 28/05/2021.

[53] The Things Industries. Devices - ABP vs OTAA. https://www.thethingsindustries.com/docs/devices/abp-vs-otaa. Accessed: 28/05/2021.

[54] The Things Network. Applications & Integrations. https://www.thethingsnetwork.org/docs/applications-and-integrations. Accessed: 28/05/2021.

[55] The Things Industries. Adding Applications. https://www.thethingsindustries.com/docs/integrations/adding-applications. Accessed: 28/05/2021.

[56] The Things Network. Adding Devices. https://www.thethingsnetwork.org/docs/devices-and-gateways/adding-devices. Accessed: 28/05/2021.

[57] The Things Network. Devices - Device Registration. https://www.thethingsnetwork.org/docs/devices/registration. Accessed: 28/05/2021.

[58] The Things Stack. Payload Formatters. https://www.thethingsindustries.com/docs/integrations/payload-formatters. Accessed: 26/05/2021.

[59] The Things Network. CayenneLPP. https://www.thethingsnetwork.org/docs/devices/arduino/api/cayennelpp. Accessed: 26/05/2021.

[60] The Things Network. Applications - Integrations - HTTP. https://www.thethingsnetwork.org/docs/applications/http. Accessed: 25/04/2021.

[61] The Things Industries. Integrations - Creating Webhooks. https://www.thethingsindustries.com/docs/integrations/webhooks/creating-webhooks. Accessed: 28/05/2021.

[62] The Things Network. Duty Cycle - Fair Use Policy. https://www.thethingsnetwork.org/docs/lorawan/duty-cycle/index.html#fair-use-policy. Accessed: 22/04/2021.

[63] Arjan van Bentem. Airtime calculator for LoRaWAN. https://avbentem.github.io/airtime-calculator/ttn/eu868. Accessed: 22/04/2021.

[64] Microsoft Azure. Azure Functions Serverless Compute. https://azure.microsoft.com/en-us/services/functions. Accessed: 02/04/2021.

[65] Amazon Web Services. AWS Lambda – Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda. Accessed: 23/04/2021.

[66] Google Cloud. Cloud Functions | Google Cloud. https://cloud.google.com/functions. Accessed: 23/04/2021.

[67] IBM Cloud. IBM Cloud Functions. https://cloud.ibm.com/functions. Accessed: 23/04/2021.

[68] Google Cloud. Firebase. https://firebase.google.com. Accessed: 23/04/2021.

[69] Google Cloud. Cloud Firestore | Store and sync app data at global scale | Firebase. https://firebase.google.com/products/firestore. Accessed: 23/04/2021.

[70] InfluxData. InfluxDB Time Series Platform | InfluxData. https://www.influxdata.com/products/influxdb. Accessed: 02/04/2021.

[71] InfluxData. Get started with Flux | InfluxDB OSS 2.0 Documentation. https://docs.influxdata.com/influxdb/v2.0/query-data/get-started. Accessed: 20/04/2021.

[72] Grafana. Grafana: The open observability platform. https://grafana.com. Accessed: 28/05/2021.

[73] Grafana. Alert notifications | Grafana Labs. https://grafana.com/docs/grafana/latest/alerting/notifications/#list-of-supported-notifiers. Accessed: 11/04/2021.

[74] Robin Schoenmaeckers. Evaluating LoRa and LoRaWAN performance in Louvain-la-Neuve. Master's thesis, UCLouvain, 2019.

[75] Gianni Pasolini, Chiara Buratti, Luca Feltrin, Flavio Zabini, Cristina De Castro, Roberto Verdone, and Oreste Andrisano. Smart city pilot projects using LoRa and IEEE802.15.4 technologies. *Sensors*, 18(4):1118, 2018.

[76] Davide Magrin, Marco Centenaro, and Lorenzo Vangelista. Performance evaluation of LoRa networks in a smart city scenario. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2017.

[77] Marco Cattani, Carlo Alberto Boano, and Kay Römer. An experimental evaluation of the reliability of lora long-range low-power wireless communication. *Journal of Sensor and Actuator Networks*, 6(2):7, 2017.

[78] Ramon Sanchez-Iborra, Jesus Sanchez-Gomez, Juan Ballesta-Viñas, Maria-Dolores Cano, and Antonio F Skarmeta. Performance evaluation of LoRa considering scenario conditions. *Sensors*, 18(3):772, 2018.

[79] Mattia Rizzi, Paolo Ferrari, Alessandra Flammini, Emiliano Sisinni, and Mikael Gidlund. Using LoRa for industrial wireless networks. In *2017 IEEE 13th international workshop on factory communication systems (WFCS)*, pages 1–4. IEEE, 2017.

[80] Bernat Carbonés Fargas and Martin Nordal Petersen. GPS-free geolocation using LoRa in low-power WANs. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–6. IEEE, 2017.

[81] Christos Bouras, Apostolos Gkamas, Vasileios Kokkinos, and Nikolaos Papachristos. Using LoRa technology for IoT monitoring systems. In *2019 10th International Conference on Networks of the Future (NoF)*, pages 134–137. IEEE, 2019.

[82] Daeun Yim, Jiwon Chung, Yulim Cho, Hyunji Song, Daehan Jin, Sojeong Kim, Sungwook Ko, Anthony Smith, and Austin Riegsecker. An experimental LoRa performance evaluation in tree farm. In *2018 IEEE Sensors Applications Symposium (SAS)*, pages 1–6. IEEE, 2018.

[83] Taoufik Bouguera, Jean-François Diouris, Jean-Jacques Chaillout, Randa Jaouadi, and Guillaume Andrieux. Energy consumption model for sensor nodes based on LoRa and LoRaWAN. *Sensors*, 18(7):2104, 2018.

[84] Ashirwad Gupta and Makoto Fujinami. Battery Optimal Configuration of Transmission Settings in LoRa Moving Nodes. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6. IEEE, 2019.

[85] Phui San Cheong, Johan Bergs, Chris Hawinkel, and Jeroen Famaey. Comparison of LoRaWAN classes and their power consumption. In *2017 IEEE Symposium on Communications and Vehicular Technology (SCVT)*, pages 1–6. IEEE, 2017.

[86] George Klimiashvili, Cristiano Tapparello, and Wendi Heinzelman. LoRa vs. WiFi Ad Hoc: A Performance Analysis and Comparison. In *2020 International Conference on Computing, Networking and Communications (ICNC)*, pages 654–660. IEEE, 2020.

[87] Hamza Zemrane, Youssef Baddi, and Abderrahim Hasbi. Comparison between IOT protocols: ZigBee and WiFi using the OPNET simulator. In *Proceedings of the 12th International Conference on Intelligent Systems: Theories and Applications*, pages 1–6, 2018.

[88] Union des Villes et Communes de Wallonie asbl. Ottignies-Louvain-la-Neuve en fiche: coordonnées, bourgmestre, coalition, population et superficie. `https://www.uvcw.be/fiches-locales/25121`. Accessed: 11/04/2021.

[89] Espressif Systems. ESP32 Series, Version 3.6. https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf. Accessed: 12/04/2021.

[90] Scott Campbell. Basics of UART Communication. *Circuit Basics*, 2016. https://www.circuitbasics.com/basics-uart-communication. Accessed: 10/06/2021.

[91] NXP Semiconductors. UM10204 - I^2C-bus specification and user manual - Rev. 6. https://www.nxp.com/docs/en/user-guide/UM10204.pdf. Accessed: 10/06/2021.

[92] Heltec Automation. WiFi LoRa 32. https://heltec.org/project/wifi-lora-32. Accessed: 29/05/2021.

[93] Arduino. Getting started with Arduino products. https://www.arduino.cc/en/Guide. Accessed: 28/05/2021.

[94] Heltec Automation. ESP32_LoRaWAN Arduino library. https://github.com/HelTecAutomation/ESP32_LoRaWAN. Accessed: 28/05/2021.

[95] Heltec Automation. License querying website. https://resource.heltec.cn/search. Accessed: 06/05/2021.

[96] Bosch Sensortec. Humidity sensor BME280. https://www.bosch-sensortec.com/products/environmental-sensors/humidity-sensors-bme280. Accessed: 30/03/2021.

[97] AZDelivery. GY-BME280 Barometric sensor for temperature, humidity and air pressure. https://www.az-delivery.de/en/products/gy-bme280. Accessed: 30/03/2021.

[98] AZDelivery. MQ-135 Gas Sensor Air Quality Module. https://www.az-delivery.de/en/products/mq-135-gas-sensor-modul. Accessed: 30/03/2021.

[99] Bosch Sensortec. Gas sensor BME680. https://www.bosch-sensortec.com/products/environmental-sensors/gas-sensors/bme680. Accessed: 02/04/2021.

[100] Bosch Sensortec. Bosch Sensortec Environmental Cluster library - GitHub. https://github.com/BoschSensortec/BSEC-Arduino-library. Accessed: 02/04/2021.

[101] AZDelivery. KY-037 High Sensitive Microphone. https://www.az-delivery.de/en/products/mikrofon-modul-gross. Accessed: 30/03/2021.

[102] MakerHawk. GT-U7 GPS module. https://www.makerhawk.com/products/makerhawk-gps-module-gps-drone-compatible-microcontroller-for-neo-6m Accessed: 30/03/2021.

[103] Eric S. Raymond. NMEA Revealed. April 2021. https://gpsd.gitlab.io/gpsd/NMEA.html. Accessed: 29/05/2021.

[104] Mikal Hart. TinyGPS++ Arduino library. http://arduiniana.org/libraries/tinygpsplus. Accessed: 30/03/2021.

[105] Heltec Automation. WIFI LoRa 32(V2) Pinout Diagram. https://resource.heltec.cn/download/WiFi_LoRa_32/WIFI_LoRa_32_V2.pdf. Accessed: 29/05/2021.

[106] Random Nerd Tutorials. ESP32 Pinout Reference: Which GPIO pins should you use? https://randomnerdtutorials.com/esp32-pinout-reference-gpios. Accessed: 23/05/2021.

[107] Google Maps Platform. Google Geolocation API. https://developers.google.com/maps/documentation/geolocation/overview. Accessed: 02/04/2021.

[108] Germán Martín. WifiLocation Arduino library. https://github.com/gmag11/WifiLocation. Accessed: 02/04/2021.

[109] John M. Wallace and Peter V. Hobbs. *Atmospheric science: an introductory survey*, volume 92, chapter 3, pages 69–71. Elsevier, 2006.

[110] krismath, Cairin Michie, and Christian Fritz. How to calculate altitude from current temperature and pressure? https://physics.stackexchange.com/questions/333475/how-to-calculate-altitude-from-current-temperature-and-pressure. Accessed: 24/05/2021.

[111] Gonzalo Casas. From zero to LoRaWAN in a weekend. https://github.com/ttn-zh/ic880a-gateway/wiki. Accessed: 21/04/2021.

[112] JSON Web Tokens. https://jwt.io. Accessed: 10/06/2021.

[113] InfluxData. Get started with InfluxDB | InfluxDB OSS 2.0 Documentation. https://docs.influxdata.com/influxdb/v2.0/get-started. Accessed: 02/04/2021.

[114] Github. InfluxDB 2.0 JavaScript client. https://github.com/influxdata/influxdb-client-js. Accessed: 02/04/2021.

[115] Grafana. Download Grafana | Grafana Labs. https://grafana.com/grafana/download. Accessed: 02/04/2021.

[116] The Things Network. Devices - The Things Node. https://www.thethingsnetwork.org/docs/devices/node. Accessed: 29/04/2021.

[117] Danny Cohen. On holy wars and a plea for peace. *Computer*, 14(10):48–54, 1981.

[118] Electronics Tutorials. DC Circuits - Voltage Dividers. https://www.electronics-tutorials.ws/dccircuits/voltage-divider.html. Accessed: 10/06/2021.

[119] Hugh D. Young and Roger A. Freedman. *University Physics with Modern Physics*. Pearson, 14th edition, 2016.

[120] Random Nerd Tutorials. ESP32 ADC – Read Analog Values with Arduino IDE. https://randomnerdtutorials.com/esp32-adc-analog-read-arduino-ide. Accessed: 05/05/2021.

[121] UCLouvain. Welcome. https://sites.uclouvain.be/welcome. Accessed: 06/05/2021.

[122] Keysight. E3631A 80W Triple Output Power Supply. https://www.keysight.com/be/en/product/E3631A/80w-triple-output-power-supply-6v-5a--25v-1a.html. Accessed: 06/05/2021.

[123] Tektronix. DMM7510. https://www.tek.com/tektronix-and-keithley-digital-multimeter/dmm7510. Accessed: 06/05/2021.

[124] International Electrotechnical Commission. Electric charge. https://www.electropedia.org/iev/iev.nsf/display?openform&ievref=113-02-10. Accessed: 01/06/2021.

[125] Jon Postel et al. RFC0793: Transmission Control Protocol. 1981.

[126] Eric Rescorla. RFC2818: HTTP Over TLS. 2000.

[127] GlobalSign. What is an SSL Certificate? https://www.globalsign.com/en/ssl-information-center/what-is-an-ssl-certificate. Accessed: 10/06/2021.

[128] Espressif Systems. esptool - Espressif SoC serial bootloader utility. https://github.com/espressif/esptool. Accessed: 24/05/2021.

[129] Espressif Systems. API Guides - Flash Encryption. https://docs.espressif.com/projects/esp-idf/en/latest/esp32/security/flash-encryption.html. Accessed: 24/05/2021.

[130] Heltec Automation. CubeCell Dev-board. https://heltec.org/project/htcc-ab01. Accessed: 06/05/2021.

[131] Amazon. 1100mAh LiPo batteries. https://www.amazon.fr/dp/B087LTZW61. Accessed: 26/05/2021.

[132] Amazon. ABS case 115x90x68. https://www.amazon.fr/gp/product/B07VRJ3Z13. Accessed: 26/05/2021.

[133] Microsoft Azure. Azure functions pricing. https://azure.microsoft.com/en-us/pricing/details/functions/#pricing. Accessed: 04/06/2021.

[134] Google Cloud Platform. Firebase Pricing. https://firebase.google.com/pricing. Accessed: 04/06/2021.

[135] Microsoft Azure. Azure VM and disks pricing estimates. https://azure.com/e/ceb507105172468bbf75bc7e8e1a098b. Accessed: 04/06/2021.

[136] Google Maps Platform. Pricing & Plans. https://cloud.google.com/maps-platform/pricing. Accessed: 26/05/2021.

Appendix A

End device configuration and sketch

A.1 YAML configuration file example

```yaml
deviceName: esp-example
configuration:
  wakeupPeriod: 10
  nMeasurements: 4
  version: 3
  wifi:
    ssid: SSID
    password: PASSWORD
    googleKey: GOOGLE_API_KEY
    webhook:
      url: https://lln-smart-city.azurewebsites.net/api/InfluxDBIntegration
      token: AUTHORIZATION_TOKEN
  ttn:
    license: 0x00000000, 0x00000000, 0x00000000, 0x00000000
    devEui: 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    appEui: 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    appKey: >-
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
sensors:
  - type: bme280
    pins:
      sda: 4
      scl: 15
    thresholds:
      temperature: 30
      pressure: 102000
      humidity: 80
```

```
  - type: mq135
    pins:
      analog: 36
    thresholds:
      co2: 700
  - type: sound
    pins:
      analog: 39
    thresholds:
      noise: 4090
  - type: gps
    serialBaudRate: 9600
    pins:
      rx: 16
      tx: 17
```

A.2 End device sketch example

```
#include <EspDevice.h>

// Wake-up period and number of measurements before sending
#define WAKEUP_PERIOD_MIN 10
#define N_MEASUREMENTS    4
#define VERSION           3

/* Heltec license to use LoRaWan with the device */
uint32_t license[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000};

/* OTAA parameters */
uint8_t DevEui[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
uint8_t AppEui[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
uint8_t AppKey[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,\
                    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};

EspDevice esp(license, DevEui, AppEui, AppKey);

// WiFi parameters
const char *ssid = "SSID";
const char *password = "PASSWORD";
const char *googleKey = "GOOGLE_API_KEY";
const char *urlInflux = "https://lln-smart-city.azurewebsites.net/"
                        "api/InfluxDBIntegration";
const char *token = "AUTHORIZATION_TOKEN";

// BME280 constants
#define BME280_SDA_PIN 4
#define BME280_SCL_PIN 15
#define TEMPERATURE_EMERGENCY_THRESHOLD 30
```

```
#define PRESSURE_EMERGENCY_THRESHOLD 102000
#define HUMIDITY_EMERGENCY_THRESHOLD 80

// MQ135 constants
#define MQ135_PIN   36
#define CO2_EMERGENCY_THRESHOLD 700

// Sound sensor constants
#define SOUND_PIN   39
#define NOISE_EMERGENCY_THRESHOLD 4090

// GPS constants
#define GPS_SERIAL_BAUD 9600
#define GPS_RX  16
#define GPS_TX  17

void setup()
{
    // Run setup method of the ESP32 device
    esp.initPacket(WAKEUP_PERIOD_MIN, N_MEASUREMENTS, VERSION);
    esp.initWifi(ssid, password, googleKey, urlInflux, token);
    esp.setup();
    esp.addBme280(BME280_SDA_PIN, BME280_SCL_PIN,
                  TEMPERATURE_EMERGENCY_THRESHOLD,
                  PRESSURE_EMERGENCY_THRESHOLD,
                  HUMIDITY_EMERGENCY_THRESHOLD);
    esp.addGasSensor(MQ135_PIN, CO2_EMERGENCY_THRESHOLD);
    esp.addSoundSensor(SOUND_PIN, NOISE_EMERGENCY_THRESHOLD);
    esp.addGPS(GPS_SERIAL_BAUD, GPS_RX, GPS_TX);
}

void loop()
{
    // Run loop method of the ESP32 device
    esp.loop();
}
```

Appendix B

Electrical consumption measurements

B.1 LoRaWAN vs Wi-Fi

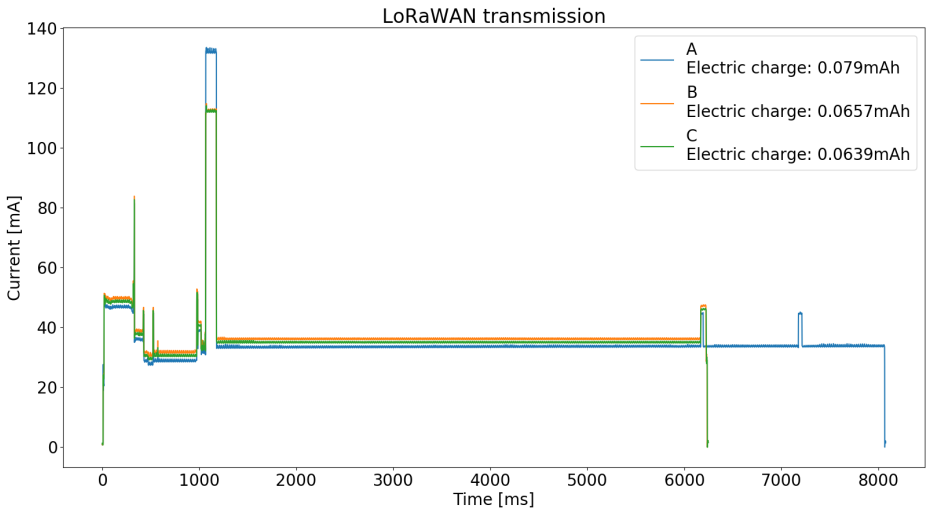


Figure B.1: Electrical consumption - LoRaWAN transmission

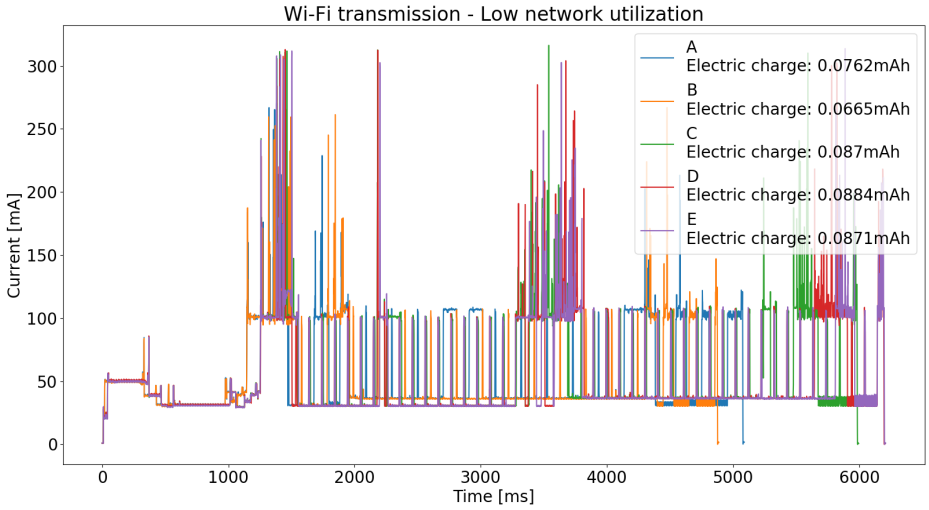


Figure B.2: Electrical consumption - Low Wi-Fi utilization

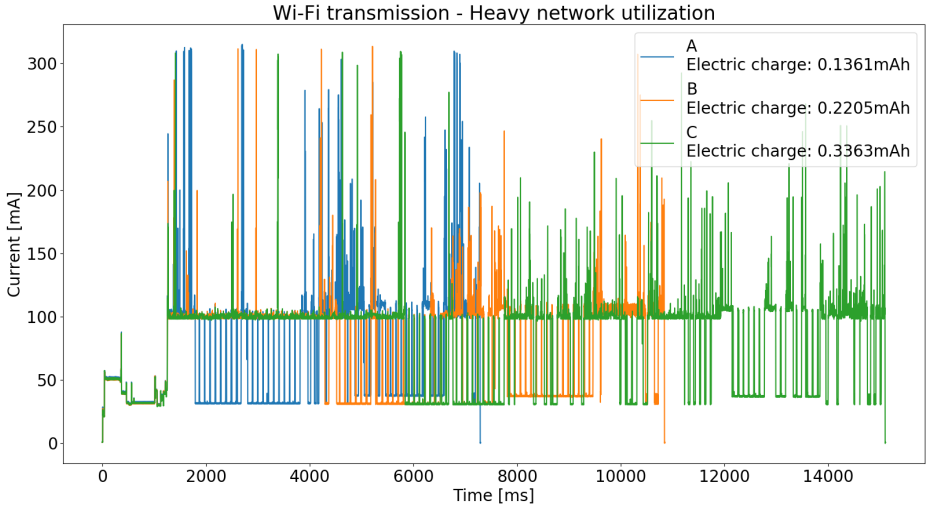


Figure B.3: Electrical consumption - Heavy Wi-Fi utilization

B.2 Sensors

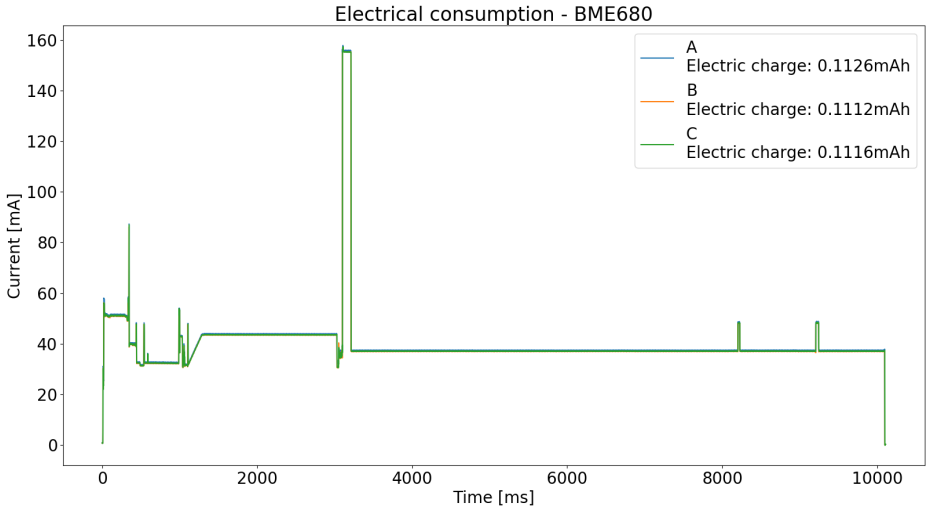


Figure B.4: Electrical consumption - BME680

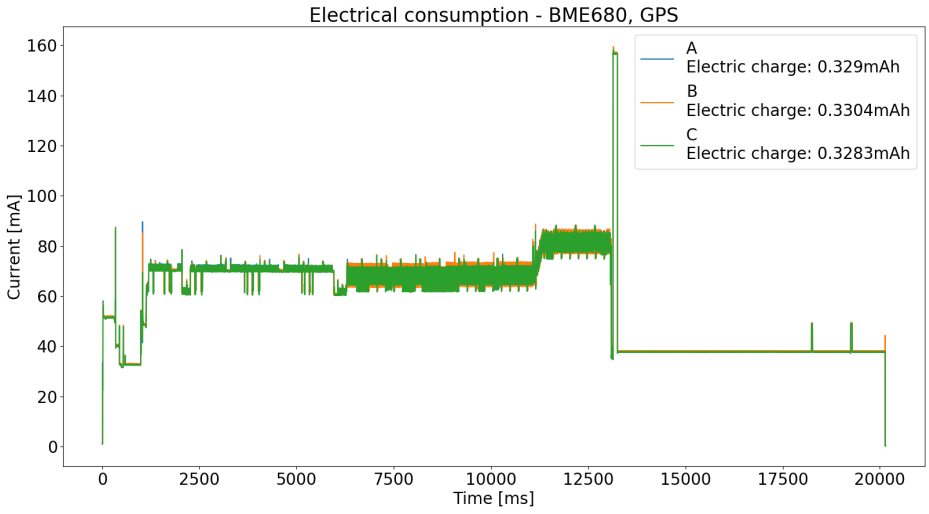


Figure B.5: Electrical consumption - BME680, GPS

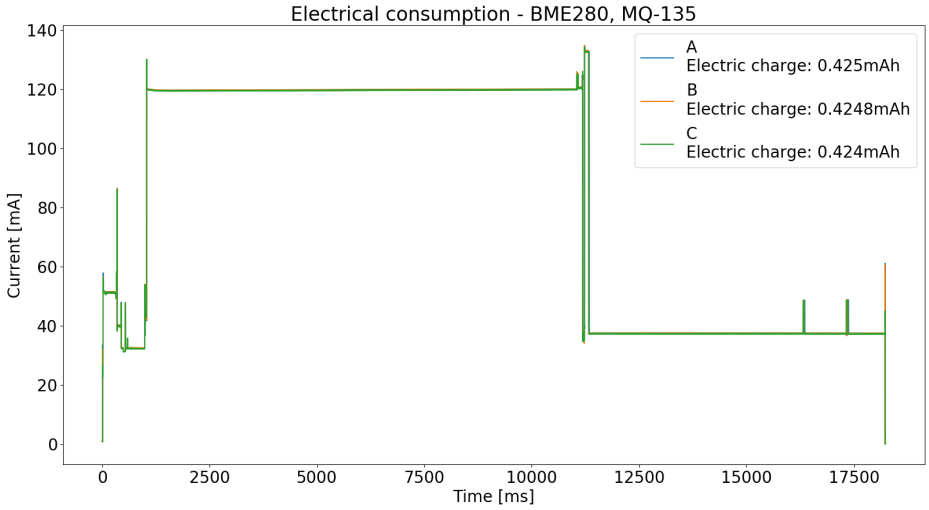


Figure B.6: Electrical consumption - BME280, MQ-135

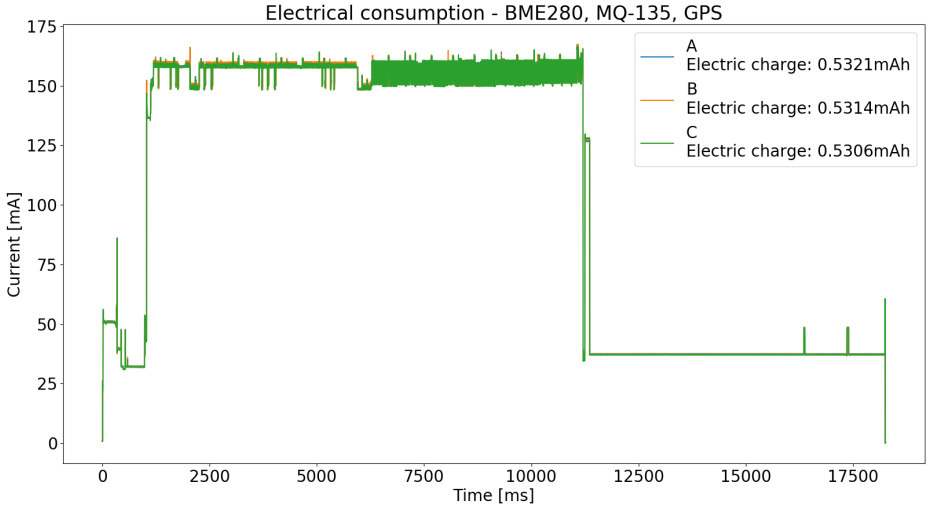


Figure B.7: Electrical consumption - BME280, MQ-135, GPS

B.3 Batch transmission

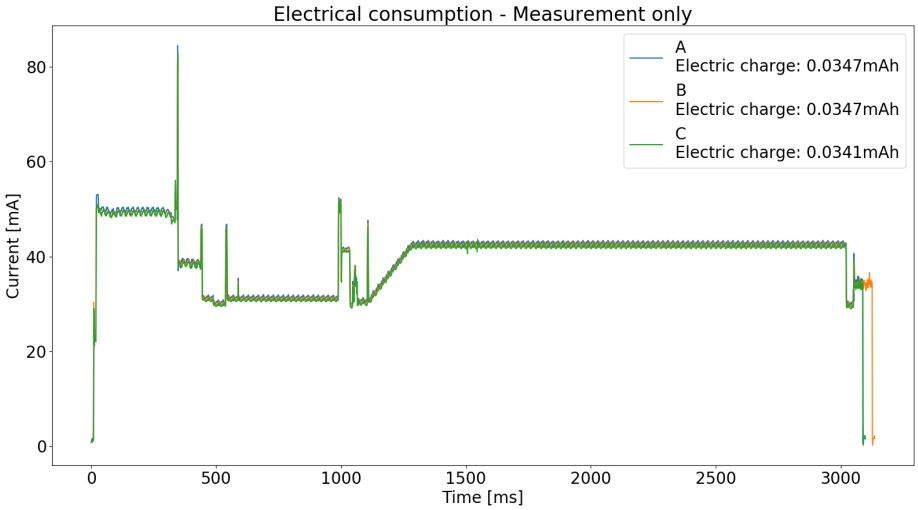


Figure B.8: Electrical consumption - Measurement only

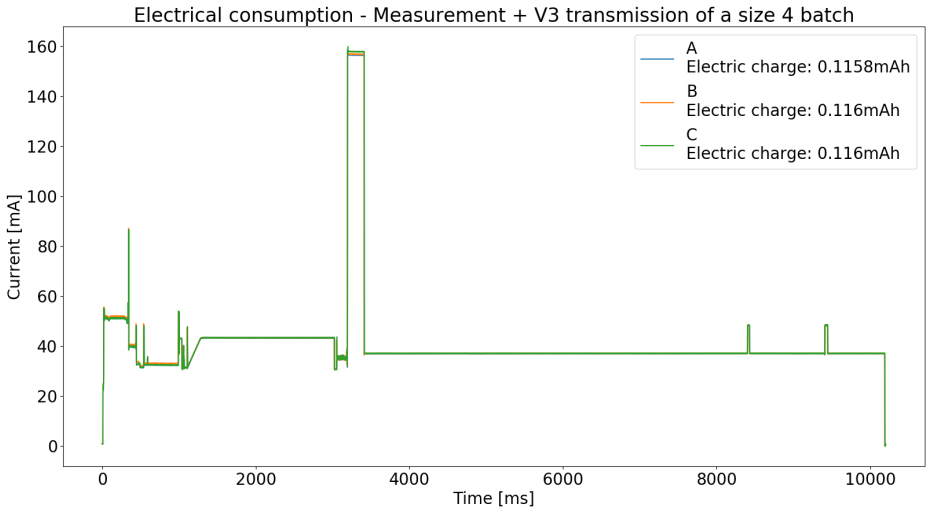


Figure B.9: Electrical consumption - Measurement + V3 transmission of a size 4 batch

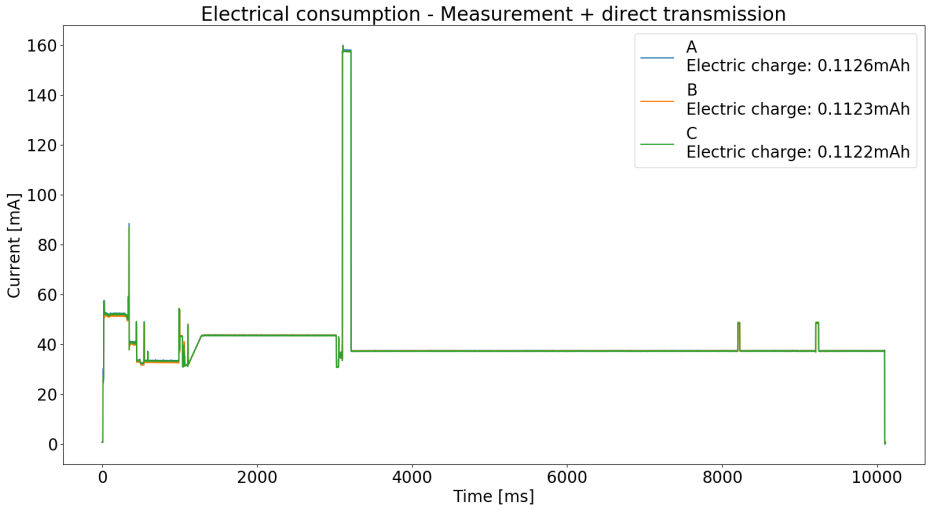


Figure B.10: Electrical consumption - Measurement + direct transmission

Batch size	Packet format	A	B	C	Mean	Std. dev. ($\times 10^{-3}$)
2	V2	0.1142	0.1147	0.1146	0.1145	0.265
	V3	0.1135	0.1138	0.1132	0.1135	0.300
3	V2	0.1162	0.1166	0.1165	0.1164	0.208
	V3	0.1148	0.1148	0.1150	0.1149	0.115
4	V2	0.1173	0.1185	0.1185	0.1181	0.693
	V3	0.1158	0.1160	0.1160	0.1159	0.115
5	V2	0.1203	0.1205	0.1204	0.1204	0.100
	V3	0.1166	0.1172	0.1172	0.1170	0.346
6	V3	0.1185	0.1184	0.1184	0.1184	0.058
7	V3	0.1194	0.1196	0.1192	0.1194	0.200
8	V3	0.1205	0.1201	0.1205	0.1204	0.231

Table B.1: Electrical consumption [mAh] depending on the packet format and batch size - Measurements data

B.4 ADR influence

ADR enabled						
Batch size	Packet format	A	B	C	Mean	Std. dev. ($\times 10^{-3}$)
1	V1	0.1126	0.1123	0.1122	0.1124	0.208
2	V2	0.1142	0.1147	0.1146	0.1145	0.265
	V3	0.1135	0.1138	0.1132	0.1135	0.300
3	V2	0.1162	0.1166	0.1165	0.1164	0.208
	V3	0.1148	0.1148	0.1150	0.1149	0.115
4	V2	0.1173	0.1185	0.1185	0.1181	0.693
	V3	0.1158	0.1160	0.1160	0.1159	0.115
5	V2	0.1203	0.1205	0.1204	0.1204	0.100
	V3	0.1166	0.1172	0.1172	0.1170	0.346
6	V3	0.1185	0.1184	0.1184	0.1184	0.058
7	V3	0.1194	0.1196	0.1192	0.1194	0.200
8	V3	0.1205	0.1201	0.1205	0.1204	0.231
ADR disabled						
Batch size	Packet format	A	B	C	Mean	Std. dev. ($\times 10^{-3}$)
1	V1	0.1122	0.1129	0.1126	0.1126	0.351
2	V2	0.1138	0.1150	0.1146	0.1145	0.611
	V3	0.1136	0.1139	/	0.1138	0.212
3	V2	0.1156	0.1166	0.1159	0.1160	0.513
	V3	0.1145	0.1150	/	0.1148	0.354
4	V2	0.1183	0.1186	0.1181	0.1183	0.252
	V3	0.1157	0.1164	/	0.1161	0.495
5	V2	0.1169	0.1206	0.1171	0.1182	2.081
	V3	0.1169	0.1175	/	0.1172	0.424
6	V3	0.1178	0.1185	0.1187	0.1183	0.473
7	V3	0.1192	0.1194	0.1197	0.1194	0.252
8	V3	0.1202	0.1208	0.1207	0.1206	0.321

Table B.2: Electrical consumption [mAh] depending on ADR, packet format, and batch size - Measurements data

B.5 CPU frequency

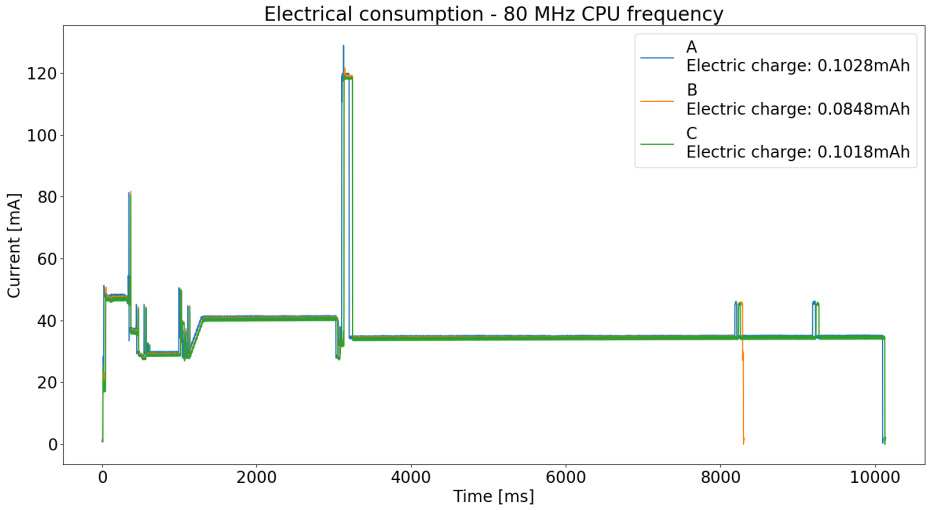


Figure B.11: Electrical consumption - 80 MHz CPU frequency

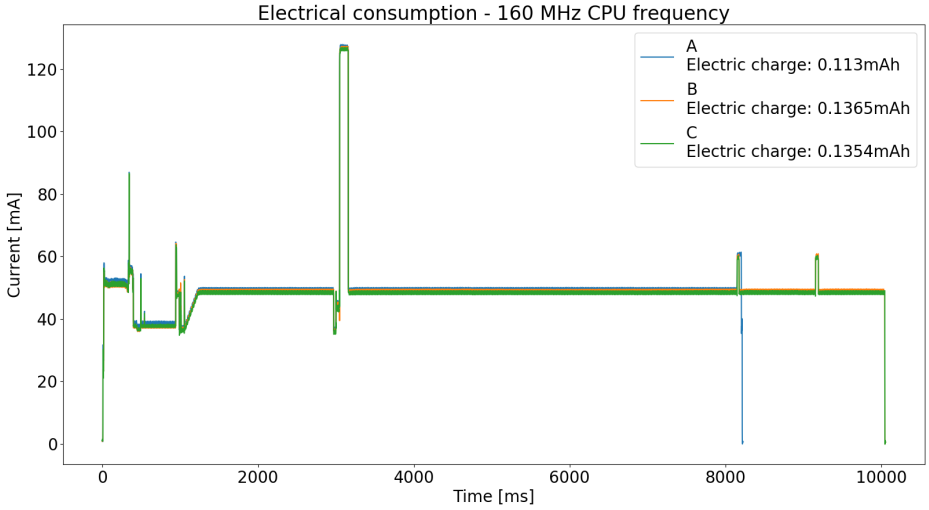


Figure B.12: Electrical consumption - 160 MHz CPU frequency

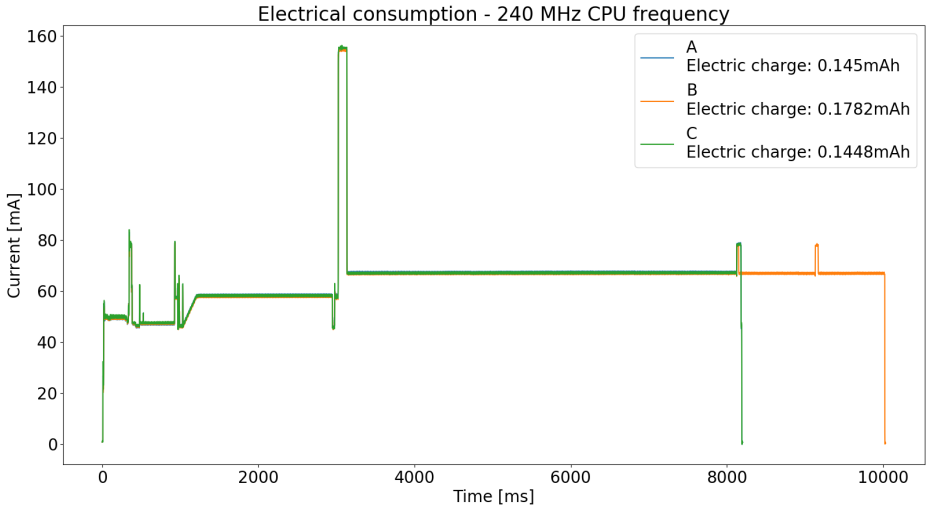


Figure B.13: Electrical consumption - 240 MHz CPU frequency

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**