INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Innovative Project

Safe city for cyclists

Written by :

Jean-Rémy Hok
Aymen Boukezzata
Arnaud Vergnet
Thomas Berton
Mohamed Arselan Meslouh

– January 20, 2022 Version –

Contents

1 Introduction

For this innovative project, we proposed our own subject because we had an idea to help solve a problem we were confronted daily: the dangers cyclists face each day in the city. The solution we thought about involved IoT infrastructures and thus felt appropriate for this project. There was no company in the industry involved, but we had the support from our teachers to guide us through the project.

Thanks to the nature of our project, we were able to work on a first hardware prototype during the 2021 KETI International Hackathon. We were able to experiment with different technologies to choose the right one for this innovative project.

All the code is available on GitHub here. Each repository contains a README giving instructions on how to install and run our solution.

In this report, you will first find some details on the context around the project, what solution we came up with to solve the issues identified, how we implemented them on a technical level, and finally how we worked together as a team.

Figure 1: Our bike with the device mounted

2 Context and problematic

2.1 What is the context around this project

Many cities in France are poorly equipped with bicycle infrastructure and are not able to provide a safe environment for cyclists. Every year in France, more than 200 deaths and 1500 injuries are the result of bicycle accidents [1] and the number of accidents involving bicycles increased by 30% between 2019 and 2020 in Paris [2].

Very often, cyclists find themselves riding alongside cars which can be very dangerous knowing the limitations of the safety mechanisms inherent to bicycles: no mirrors, no stop lights, no airbags, etc.

2.2 What are the current solutions

There are already some mobile applications existing such as Geovelo where the user can report dangerous zones. But it is not automated and the user has to stop to manually enter the information.

Infrastructure information is already mapped in databases such as OpenStreetMap, but it does not clearly indicate the danger. In some places, there are bicycle lanes which are very dangerous. A user looking at OpenStreetMap would only see the cycle lane's presence and not the danger associated.

2.3 What are our resources

We are a team of five INSA students, two from computer science, two from automation and electronics, and one from a master in IoT.

We have several computers available, Raspberry Pis and ESP boards, as well as some sensors such as ultrasonic sensors or LIDARs. We also have a bike on which we can mount the device.

The project lasted for four months, from October 2021 to January 2022. We had a few slots dedicated to the project, but most of the work was done in our spare time after classes.

3 Our solution: an on-bike device

3.1 How it works

Our project aims to help cyclists and cities identify dangerous areas in the city. As summarized in figure 2, it consists of a publicly available database through a mobile app. The app features an interactive map displaying dangerous areas. It also warns users trying to find a route through such areas. Data is generated from a device on the user's bike detecting passing vehicles' speed and distance.

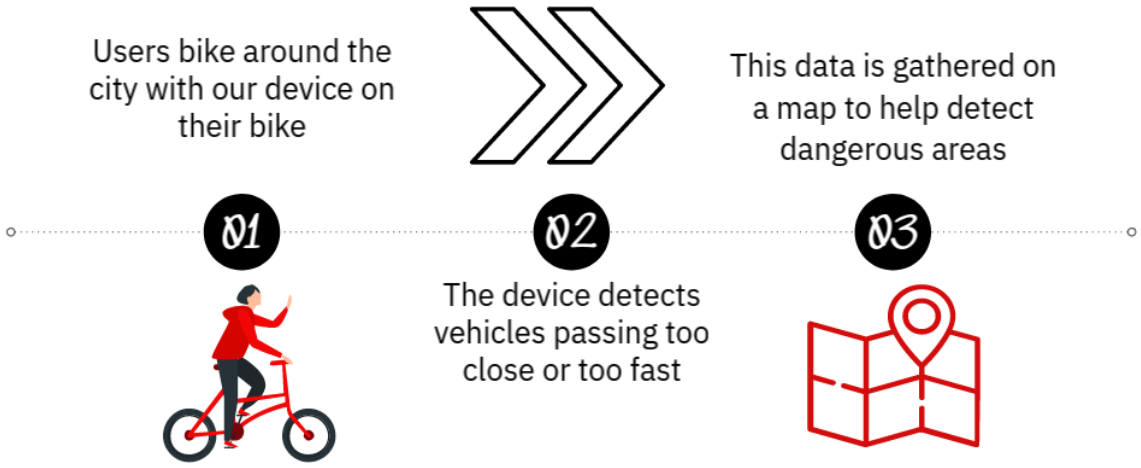


Figure 2: Overview of our solution

The embedded on-bike device uses two LIDAR (Light Imaging Detection and Ranging) to detect vehicles and communicates using Bluetooth Low Energy (BLE) to notify the user's phone a vehicle has been detected. The phone will keep these reports offline and only upload them using Wi-Fi or cellular networks when authorized by the user. This data will then be stored in a central database, accessible with a REST API.

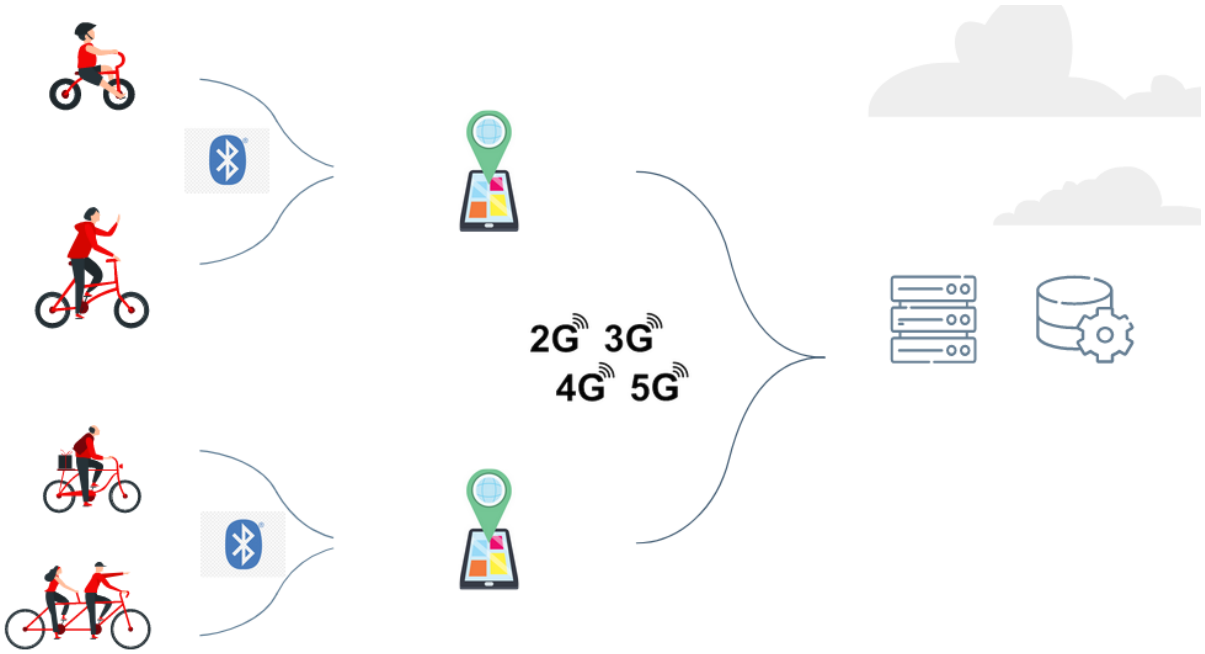


Figure 3: Detailed architecture of our solution

3.2 Why this solution

We chose this solution because we wanted to create a system allowing cyclists to voice their complaints and help each other. We wanted to have an open system everyone could use and contribute to. As nearly everyone owns a smartphone and an internet connection, using BLE and a mobile app was the best way to ensure everyone could have access to our system. In the end, the user would only need to download the app and to install our device on their bike in order to use our system.

Our solution is also designed to work on any bike to allow the project to start off and improve using personal bikes, and one day expand and be installed on city bikes or others. We designed the architecture to allow any number of devices to run at the same time: we would only need to increase the server capacity to introduce new devices.

Thanks to the project's modularity, it is also also be possible to create new embedded devices with better performance and accuracy without impacting the rest of the system.

3.3 What is our target market

Our main users will be cyclists. The app will help them plan safe routes while avoiding dangerous areas. This data will also be useful for officials in charge of urban planning to understand where cycling infrastructure is most needed. This will help guide them in building new infrastructures or improve existing ones. Our system will help cyclists have their voices heard in negotiating with the city.

4 Technical aspects

4.1 What technologies are used and why

4.1.1 Embedded Device

During the prototyping/testing phase, the main steps of the projects were the following:

- Distance Sensors prototyping and tests ;
- BLE communication prototyping and tests ;
- C++ Libraries coding for the management of the BLE communication, Sensors serial communication, Speed measure and format conversion ;
- Power supply circuit design (DC/DC conversion) for the dynamo based energy harvesting system ;
- PCB design of the custom board.

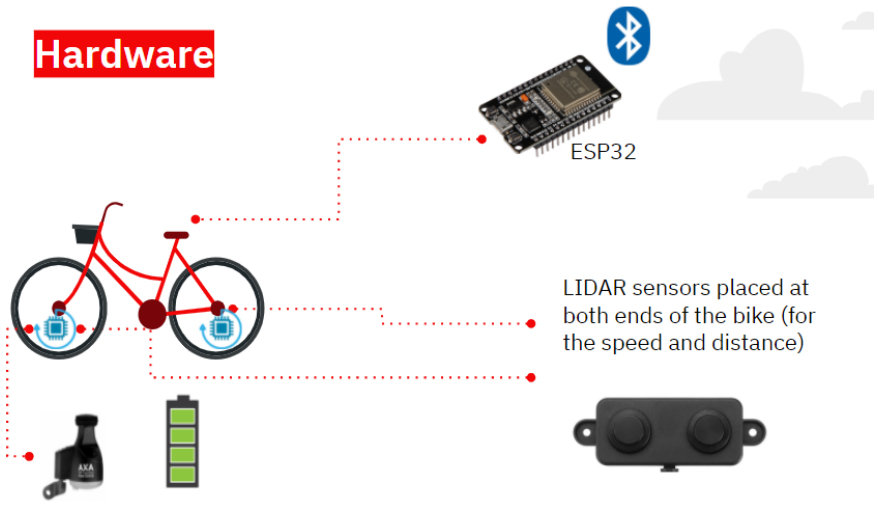


Figure 4: Overview of the Hardware System

How it works The hardware part of our project consists of an ESP32 micro-controller based smart-device that manages two LIDAR sensors. The device's role is to detect the speed and distance of cars driving alongside the cyclist (when they use car roads or bicycle lanes). These measures allow our algorithm to detect dangerous points by comparing distance and speed values to determined thresholds. If values are greater, the danger is considered of a high enough level to be reported.

These thresholds are approximate due to the lack of data and testing. However, they can be easily modified to improve precision. Our idea is to set initial values we consider logical and to run test procedures. This will allow us to refine the thresholds and converge to practical values. Our goal is to prevent a high number of false positive (FP) alarms from occurring while still remaining in a safety range. False positive are erroneous detection of objects. This can happen if the hardware and the associated software detect objects of the wrong size or speed. The main scenarios we imagined are the following:

- Speed $>$ Speed Threshold & Distance $<$ Distance Threshold: we consider the danger exists and a report is created;
- Speed $>$ Speed Threshold & Distance $>$ Distance Threshold : we consider the danger does not exist;

- Speed < Speed Threshold & Distance < Distance Threshold : we consider the danger does not exist;
- Speed < Speed Threshold & Distance > Distance Threshold : we consider the danger does not exist.

The measure of these parameters is done using two LIDAR sensors placed on both ends of the bike, facing the left side. The LIDAR modules embed a System on Chip (SoC) which performs calculations to measure the distance of passing objects. According to the data sheet, it has a range of 8 meters with a Field Of View (FOV) of 2 degrees.

The module sends an infrared modulated wave which bounces back to the module when in contact with an obstacle. The Time of Flight (ToF) principle is used to measure the distance. As described in figure 5, the LIDAR calculates the time by measuring the phase difference between the original wave and the reflected wave. It uses that time to obtain the relative distance knowing the speed of the wave (speed of light) and the wavelength (function of the frequency). The impact of the external environment on the range is minimized thanks to optimized algorithms and light paths.

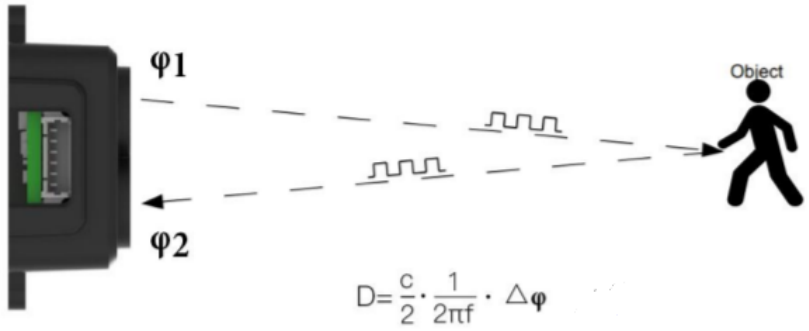


Figure 5: Time of Flight principle to measure the speed. Source: Datasheet

The range data is reliable only if the reflection surface fully covers the light spot, so the diameter of the object must at least be the same as the diameter of the light spot, and that diameter depends on the FOV (Field Of View) of the module. The formula 1 describes the minimum diameter of the object surface.

$$d = 2 * D * \tan \beta \tag{1}$$

Our LIDAR communicates with the micro-controller using a serial synchronous communication: I²C Inter-Integrated Circuit. Once the distance is measured, we can interact with the SoC embedded on the TF-Luna LIDAR module to retrieve data using commands sent through the serial port. The frames sent by the module through the serial port is structured as described in figure 6.

Byte	0	1	2	3	4	5	6	7	8
Description	0x59	0x59	Dist_L	Dist_H	Amp_L	Amp_H	Temp_L	Temp_H	Check_sum

Dist: cm

Figure 6: Serial Port output format of the LIDAR module

The frame is 8 bytes long. The first two (02) bytes are flags to announce that the upcoming two bytes represent the distance measured. It is coded with two bytes with the Least Significant Byte (LSB) coming first and the Most Significant Byte (MSB) coming second. The last Byte called "check-sum" is a correction byte to detect the loss of data. It corresponds to the sum of all of the previous bytes.

All we have to do to retrieve the distance data is to parse the serial frame byte by byte and construct a 16 bits buffer that contains the measure of the distance.

The speed is measured using the distance measurement of the LIDAR sensors and software algorithms.

When the LIDAR module at the back of the bike detects a change in the distance measured (change from no object detected to an object detected within the maximum range of 8 meters) it sets a first flag ON and saves a timestamp. The process is similar for the LIDAR sensor at the front of the bike, a second flag is set ON and a second timestamp is saved as seen in figure 7. Once both flags are ON, the speed is calculated thanks to the time difference (Timestamp2 - Timestamp1) and the distance between the two sensors, as described in formula 1.

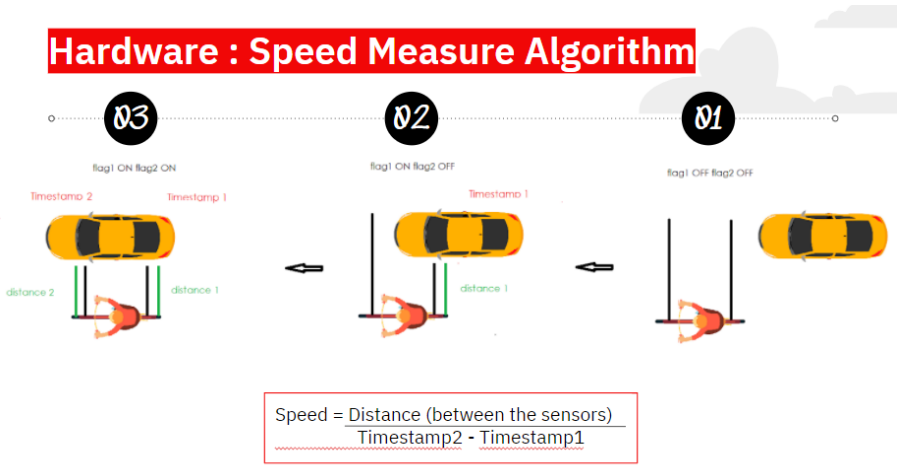


Figure 7: Flag system to avoid false positives (speed)

$$Speed = \frac{\delta D}{\delta T} \tag{2}$$

As the bike is moving, this method allows to retrieve only the relative speed of object. At the application level, the phone's GPS module will provide the speed at which the cyclist is going, which will allow the app to calculate the absolute speed of passing cars. It will also help to avoid false positive alarms since it will avoid reporting static objects.

As the LIDAR are positioned on both ends of the bike, both flags being in the ON state at the same time means the object detected is at least as long as the bike. Thus, it avoids false positives by ignoring small objects such as pedestrians and lamp posts.

Figure 8: LIDAR mounted on the back of a bike

Once the speed and distance measurements have been taken, the smart-device sends the data to the mobile application using Bluetooth Low Energy (BLE). To achieve this we have to implement a BLE server on the ESP32, also known as a BLE peripheral.

With this server, we provide one or more services which have one or more characteristics. Each

characteristic has a value and an ID. It may have descriptors, each of which have a value and an ID. The figure 9 shows our server architecture.

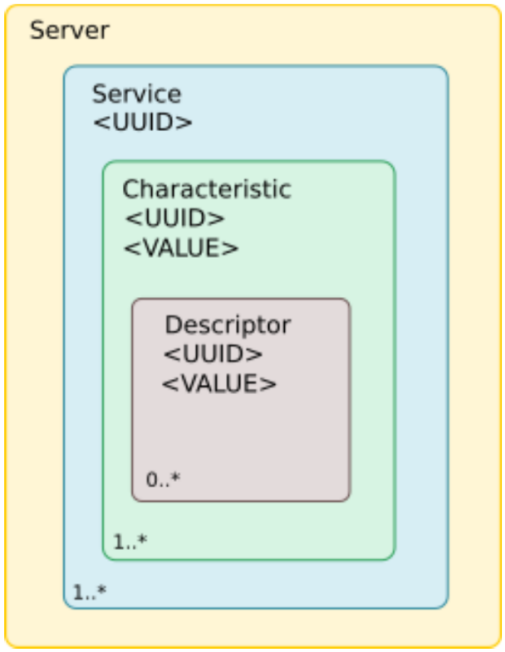


Figure 9: Architecture of the BLE server. Source: GitHub

To implement our BLE architecture, we used a BLE framework for ESP32. Thus, in our C++ code, we created a class for each part of the BLE server architecture shown in figure 9: Server, Service, Characteristic, Descriptor and Advertising.

We create the server, then the service, and finally the characteristics associated to the service. The role of these classes is to efficiently process BLE workflow while encapsulating the low level functions. They expose APIs for common actions while providing methods that can be called to tweak and tailor the operations for rarer cases.

When a BLE server is running, peer devices (clients) must be able to locate it. This is made possible through Advertising. The BLE server can broadcast its existence along with identifying information to allow a client to know what services it can provide.

The Characteristic is a stateful record with an ID and a value. It establishes the communication between our server and peer devices. If permitted, a client can read the value of a Characteristic and/or set a new value for it. This is how our smart-device sends information to the mobile app it sets the Characteristic's value to a JSON object containing the data.

The BLE framework that we use is based on the principle of real time with *FreeRtos*, to detect when an external device interacts with the ESP32. An interrupt is triggered when there is a connection or disconnection event and will directly execute callback functions that we have overwritten.

Figure 10: Free RTOS Logo

Our choices To design our smart-device, we were confronted with many choices regarding which technology to use for each purpose of our application.

Why LIDAR? The choice of the LIDAR sensors as distance sensors is due to the following characteristics and specifications:

- The TF-Luna module is relatively affordable (25 euros / retail price) ;
- The TF-Luna module's ranging accuracy is about 2cm standard deviation at a distance of 8m, which represents an accuracy of 99.975% ;

- The electrical characteristics are suitable for our application: Power supply = 5V for an average current consumption of 70mA ;
- The operating range of TF-Luna detecting white target with 90% of reflectively is 0.2-8m and 0.2-2.5m with over 10% of reflectively which still efficient in the worst case.
- The TF-Luna communicates with the Serial port (UART) or I^2C port or even both of them in sequence with the integrated multiplexing output.

Why Bluetooth Low Energy? The choice of the Wireless communication protocol is critical when it comes to IoT systems. The main goal is to manage the power consumption optimally while maintaining performances above a certain critical standard.

The main criteria we used to choose a wireless communication protocol were the following:

- The range or distance: For our application, we were looking for a short range wireless protocol ;
- The data-rate: For our application we were looking for a wireless protocol with an average data-rate which means that it did not have to be specifically high as long as it managed to send few hundred Bytes at any time (it had to operate on an open frequency band) ;
- The power consumption: For our application, the wireless protocol used had to require low power due to our lack of power supply.

Bluetooth Low Energy was thus a perfect compromise for our device. Also, since we decided to use the user's phone as a middle-ware interface, it was very convenient for us as most phones already embed Bluetooth Transceivers.

Why an ESP32? The choice of the micro-controller was based on the following criteria:

- Price (low-cost) ;
- Availability at INSA ;
- "Arduino framework" enabled ;
- Includes a UART serial communication interface ;

The ESP32 is a micro-controller that fulfills all of those criteria, and differentiates itself by the fact that it already has a **Bluetooth Low Energy Module embedded** to the Development Kit board.

4.1.2 Power Supply Unit

How it works Our system is embedded on a bicycle. Therefore, to ensure its autonomy, it has to be powered by a power storage unit (Battery).

The ESP32 as most general electronics products, works on a 3.3V supply. The battery used for the purpose of the prototyping is a Lithium-Ion battery that provides a voltage range between 2.7V and 4.2V with a mean voltage output of 3.7V. These characteristics are common to very widely used battery systems.

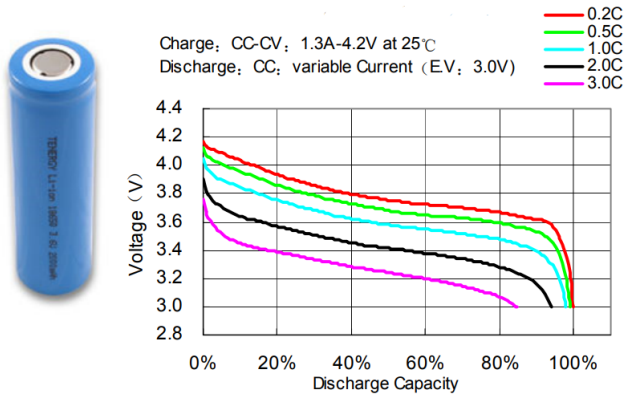


Figure 11: Rate discharge characteristics Li-ion battery. Source: Datasheet

To ensure a stabilized 3V3 output voltage to the power input of the ESP32, we needed to use a Low-dropout regulator (LDO Regulator) which in turn needed an input of at least 5V (4V4 in theory). In this case, we used the LDO regulator (ASM1117) seen in figure 12 already integrated to the ESP32 development board.

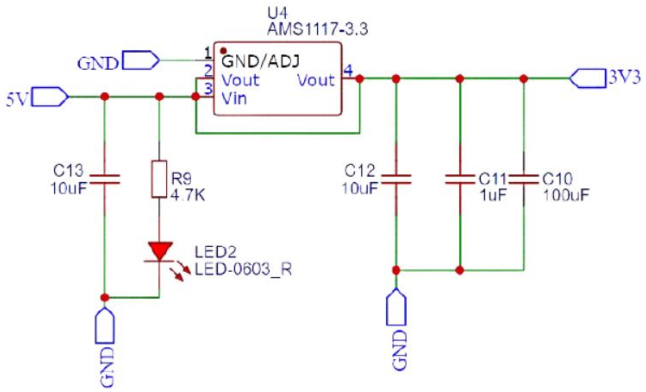


Figure 12: Low-dropout regulator 3V3 AMS1117. Source: Datasheet

To provide the minimum 5V voltage to the LDO Regulator, we performed a DC/DC boost conversion to boost the battery voltage to at least 5V. For that purpose, we used the IP5305 Integrated Circuit (IC) seen in figure 13. It has an embedded SoC allowing us to increase the voltage up to 5V and maintain that value thanks to a closed feedback regulation loop.

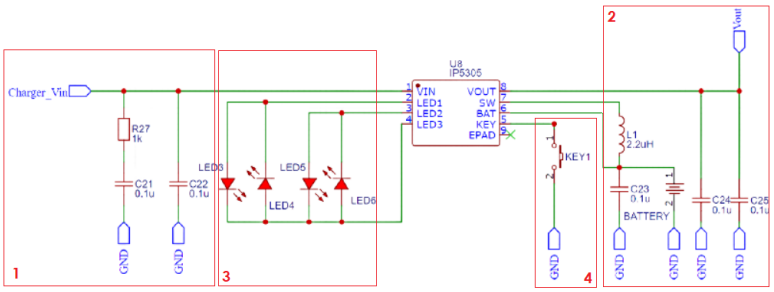


Figure 13: IP5305 Circuit. Source: Datasheet

The first block of figure 13 is the 5V battery charger input with a low-pass filter used to make our signal smoother.

The second block represents the boost converter of the circuit. It can be simplified by the schematic shown in figure 14.

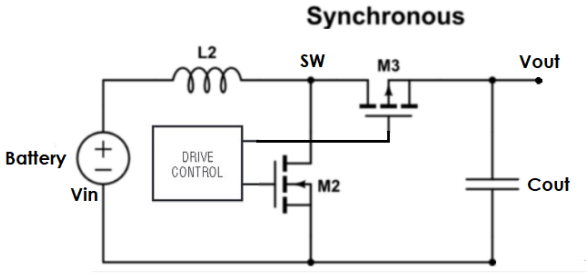


Figure 14: Synchronous Boost Converter

The integrated circuit is able to maintain a stable output voltage of 5V thanks to a regulation loop. The control of the boost converter switch is a pulse width modulation (PWM) which allows us to adjust the duty cycle value to steady the output voltage depending on the battery voltage in the SW pin of the IC. The two MOSFETs switches "M2" and "M3" are controlled by a control unit to maintain a high efficiency. This type of boost converter is called "synchronous".

The IP5305 also allows us to monitor the state of charge of the battery through the LEDs on the third block of figure 13. The tables in figure 15 show the evolution of the state of the battery in charge and discharge.

Discharging mode, 4 LEDs as the indicator

SOC（%）	L1	L2	L3	L4
SOC≥75%	ON	ON	ON	ON
50%≤SOC＜75%	ON	ON	ON	OFF
25%≤SOC＜50%	ON	ON	OFF	OFF
3%≤SOC＜25%	ON	OFF	OFF	OFF
0%＜SOC＜3%	1.5Hz blink	OFF	OFF	OFF
SOC=0%	OFF	OFF	OFF	OFF

Charging mode 4 LEDs as the indicator

SOC（%）	L1	L2	L3	L4
Full	ON	ON	ON	ON
75%≤SOC	ON	ON	ON	1.5Hz blink
50%≤SOC＜75%	ON	ON	1.5Hz blink	OFF
25%≤SOC＜50%	ON	1.5Hz blink	OFF	OFF
SOC＜25%	1.5Hz blink	OFF	OFF	OFF

Figure 15: IP5303 battery state. Source: Datasheet

The last block diagram in figure 13 represents a smart ON/OFF key. This feature is interesting in our system, because it has an integrated smart load, which allows the IC to turn off if the current consumption is below 100uA.This feature can be used when the main system disconnects from the application, so the microcontroller goes into deep sleep mode, so the microcontroller consumes less than 100uA and is turned off by the IP5305 IC, therefore we can save the battery life.

Finally, to increase the lifetime of the battery and thus make our system more autonomous, we thought of an energy harvesting system based on a Generator Type Electrical Machine (Alternator). A dynamo as in figure 16 is the perfect solution for a "bicycle-use" alternator. The dynamo provides a voltage and a current that both increase according to the mechanical power exerted at the input (rotation velocity). The maximum power provided by the dynamo we had in our possession is about 3 Watts. During the tests, we observed a saturation in the output current at around 200mA. Since we are regulating the voltage to 5V, the maximum power we can obtain is 1.25W which is not negligible.

To maintain the output voltage of the dynamo, we used another Low-dropout regulator LM317. It is more powerful and can support 40V in input, provide 1A and regulate the output voltage to 5V (see figure 17).

Figure 16: Bike dynamo mounted on a bike

Figure 17: Dynamo output 5V regulation

This second power subsystem supplied power to our device with the same 5V voltage as the primary subsystem, but providing a proportion of the current which helped in preserving the capacity of the battery longer.

The power system was composed of the two subsystems detailed above working together. The primary being the battery system, and the secondary being the dynamo.

With Otti power analyze we measured our maximum current consumption needed for the device around 620mA and an average current consumption of 220mA:

- 2x LIDAR = 2x70mA = 140mA ;
- ESP32 in normal use = 80mA ;
- ESP32 in BLE use (transmission) = 180mA ;
- In total: 620mA max (launch) and 220mA in average.

In figure 18, the green graph represents the power consumption of the system (ESP32 + 2 LIDAR) without the dynamo, while the red graph represents the complete system with dynamo (ESP32 + 2 LIDAR + dynamo).

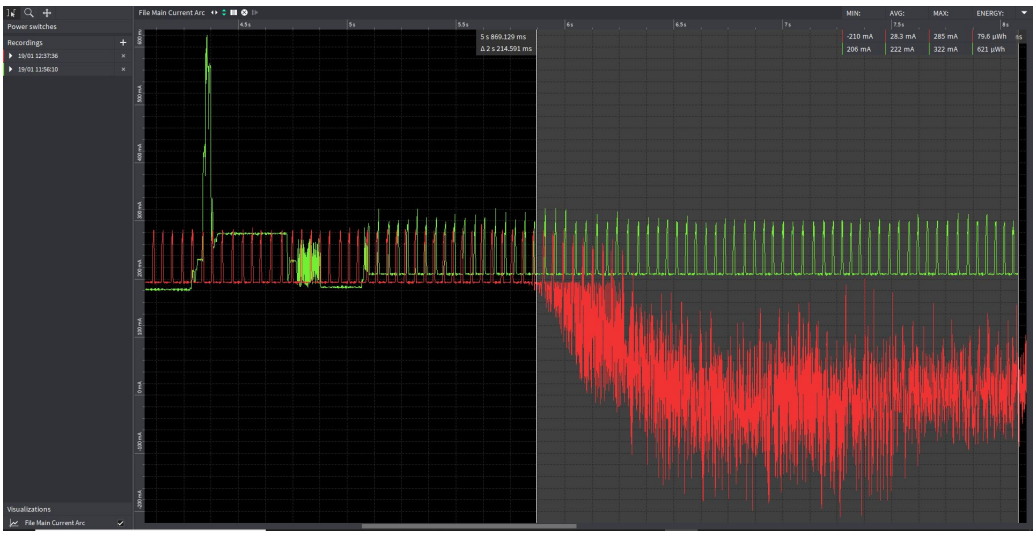


Figure 18: Power consumption with and without dynamo

The average current of the system without dynamo (green graph) is about 220mA, while the system with dynamo (red graph) consumes an average current of 28mA, which optimizes the battery life by 87.2%.

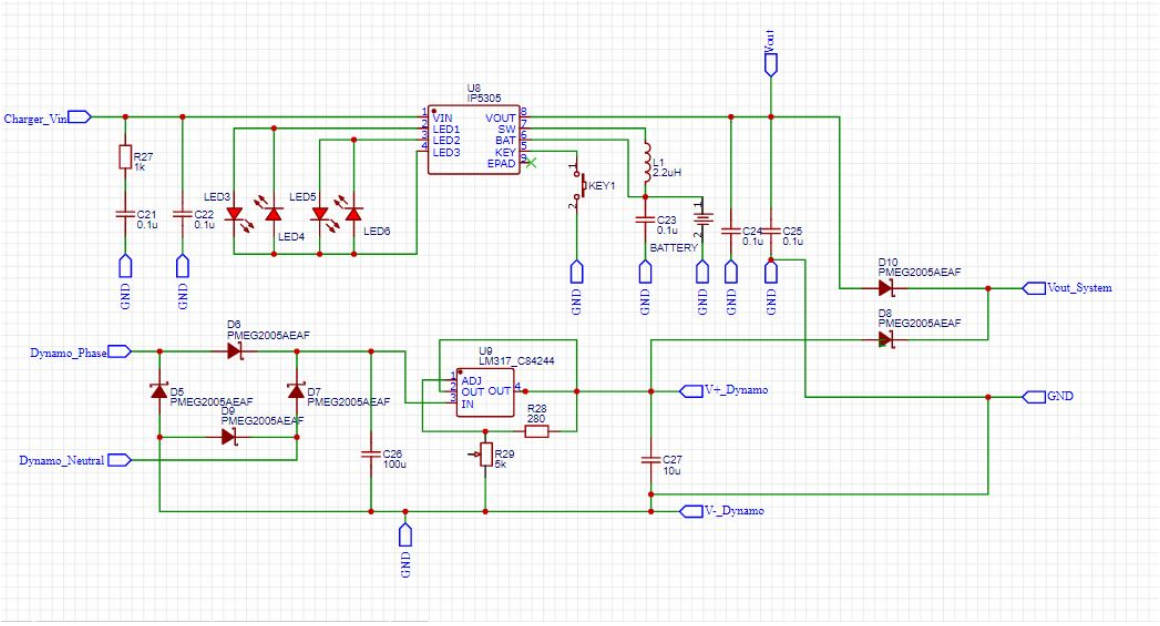


Figure 19: Full power supply system

Our choices The different choices we had to make considering the power-unit were mainly about the following components:

Why the LDO? Linear Voltage Regulator are simple and cost-effective. In terms of performances, they provide a stable voltage in output by using a higher input voltage.

The AMS1117-3V3 LDO is already embedded to the ESP32 dev-kit which means this choice was imposed. The second regulator (dynamo sub-system) was chosen for the following reasons:

- Available at INSA ;
- Low-cost ;
- Good enough for the purpose of the application.

An alternative would have been to go with a buck converter. We ran tests on a several buck converters (MP1584EN for example) and it was not conclusive. The output voltage was varying between 3V and 4V and would not stabilize.

Why the IP5305? The IP5305 is a commonly used IC in power products (COTS - power banks for instance). It integrates a boost conversion block, and allows us to retrieve the battery life status. It also has the following features:

- Synchronous boost converter providing 1A ;
- Boost converter efficiency up to 91% ;
- Adaptive charging current control, excellent adapter compatibility ;
- Integrated Key ON/OFF;
- Smart load detector, switching to standby mode automatically.

Why Li-Ion Battery? The Li-Ion batteries are the most widely used batteries in the market. They are affordable and they present a very conclusive "Ragone" diagram characteristic (Power density / Energy density).

4.1.3 Mobile app

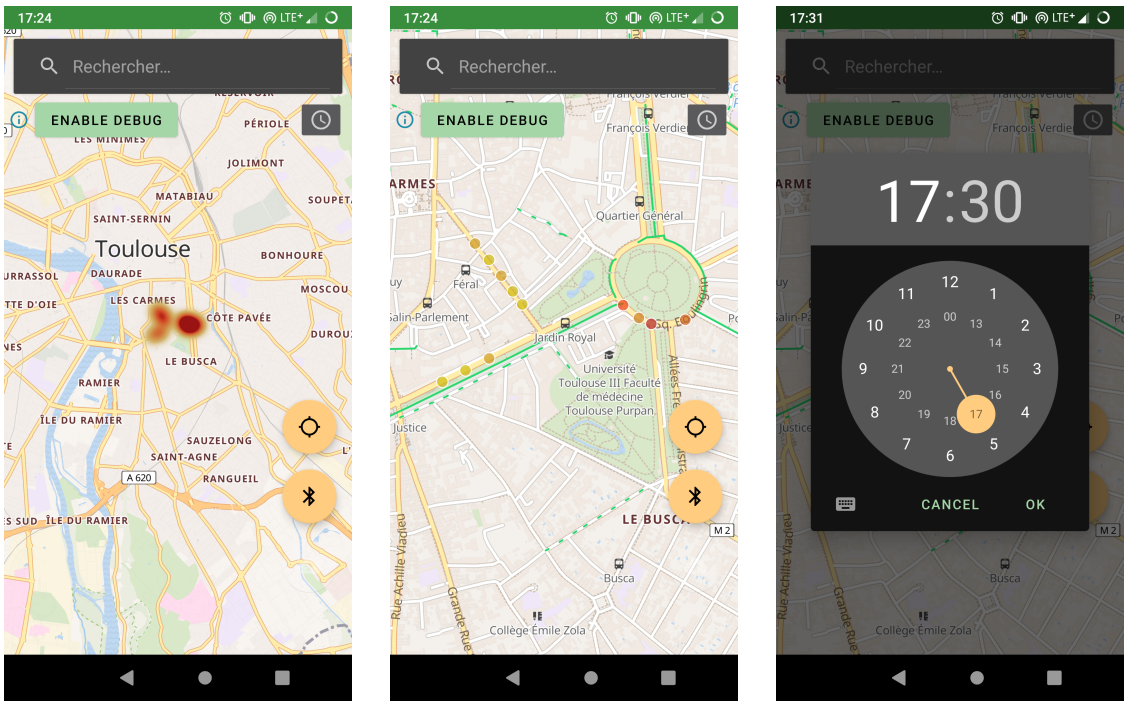
How it works Our project relies heavily on a mobile app to provide a service to the user. To speed up development and reduce costs, we chose to only target Android. Thus we wrote our app using the official Android language Kotlin.

Map View The app shows a map to the user, allowing them to see dangerous areas fetched from the server as well as bicycle paths in green. The map is rendered using the MapLibre renderer, using map tiles from the provider Geoapify. Along with map tiles, Geoapify provides place search and routing APIs. Geoapify uses data from the OpenStreetMap project, an open database mapping the earth. Using MapLibre we were able to customize the map tiles sent by Geoapify by providing a custom style. Our custom style was made to show bicycle path in green to make them stand out more.

Danger areas When zoomed out, dangerous areas are represented using a heatmap as shown in figure 20a. When zomming in, the heatmap disappears to show individual zones as in figure 20b. Each zone has a color representing the level of danger, from yellow to red. Zones displayed take into account the current time of day. Because danger on the road changes depending on the time of day, this feature is important to have accurate data. To allow users to plan their journey on a different time of day, they can click on a button to select a custom time using the selector in figure 20c.

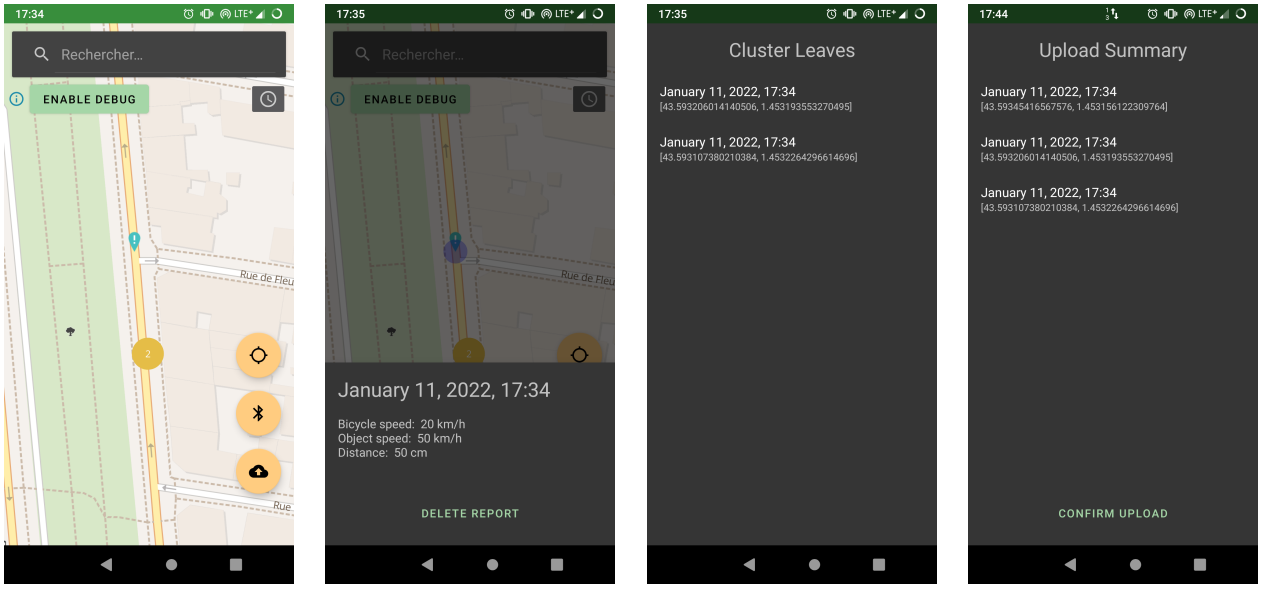
Local reports The map also shows individual reports made by the user with their on-bike device. Those reports are represented as blue pins. When pins are too close to each other (depending on the zoom level), they form a cluster indicating the number of reports inside. Those representations are shown in figure 21a. When the user clicks on a pin, a bottom sheet with detailed information appears as in figure 21a. This displays information such as the date, the object and bicycle speed, and the distance to the detected object. They can also choose to delete a report in case it is too identifiable or erroneous. If the user clicks on a cluster, the app will show all reports part of this cluster, ranked by date as seen in figure 21c.

The user can upload their reports at any time by clicking on the upload button. This will bring up a list of all reports to upload shown in figure 21d, where the user can click on each one to display its information and position on the map. When reports are uploaded, the server calculates the new danger areas by taking into account the newly uploaded reports. Those areas are then retrieved by the app to show up to date information.



(a) Heatmap (b) Areas (c) Areas

Figure 20: Danger zones

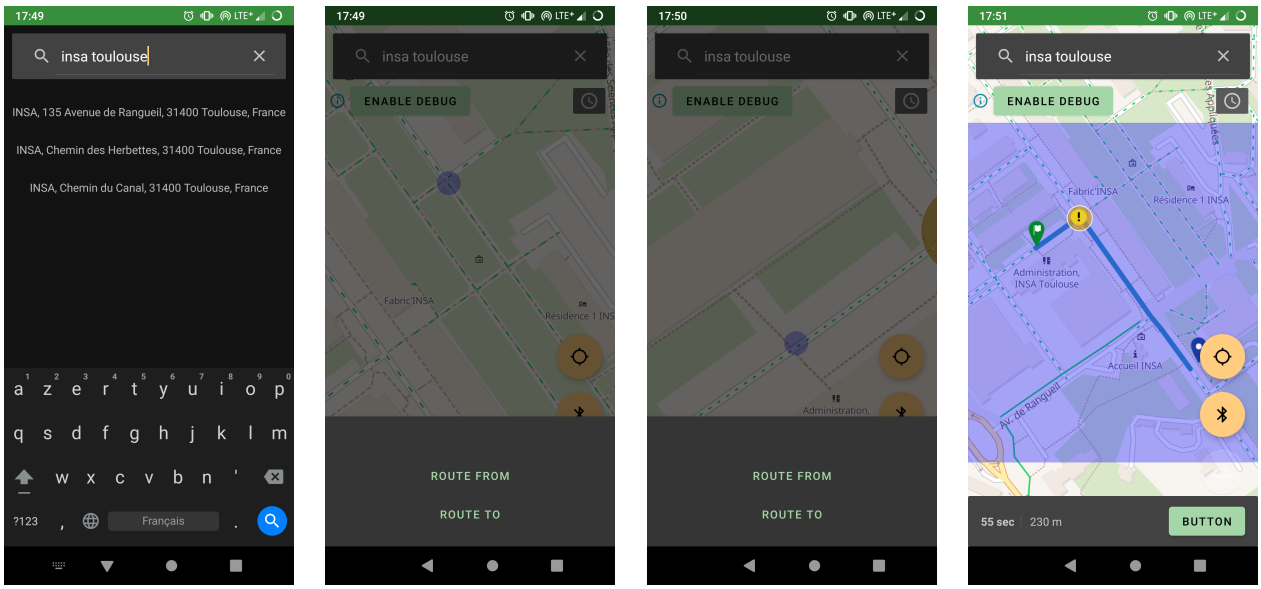


(a) Pin and cluster (b) Report data (c) Reports in a cluster (d) Upload Summary

Figure 21: Local reports

Place search Users can also search for a place by its name to show its location on the map using the search field shown in figure 22a. This feature is essential and expected by users on any application providing a map as it allows to quickly move to a known place.

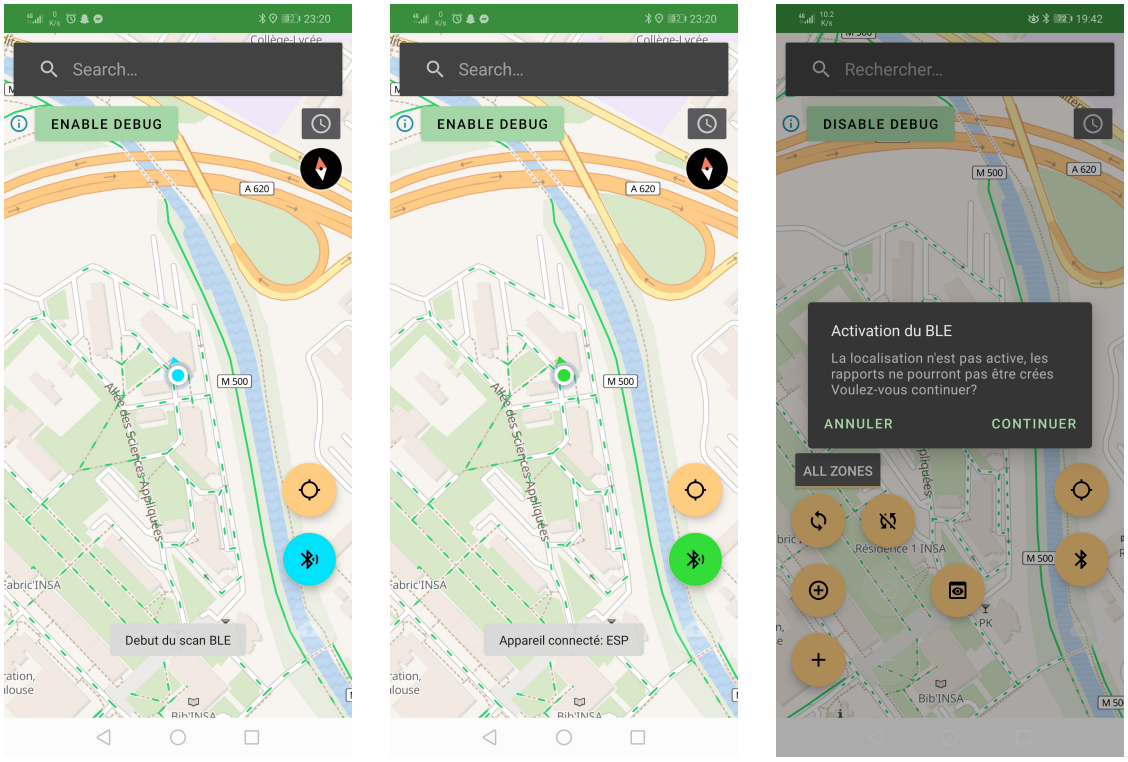
Routing It is also possible to compute a route between two points. If the route goes through a dangerous area, the user will be warned the path is not the safest by showing a warning sign as in figure 22d. To create a route, the user can click on any POI (Point of Information) or long press anywhere on the map. This can be combined with the search feature to create routes faster.



(a) Place search (b) Click on a POI (c) Long click on the map (d) Route generation

Figure 22: Place search and routing

Bluetooth Low Energy To retrieve reports from the device on the bike, the app can connect using BLE. The user simply has to click on a button to turn on BLE scanning as shown in figure 23a. The app will then automatically connect to any device broadcasting the right service ID (see figure 23b). If the device has no GPS signal available while connecting to the BLE device, the user will be warned (see figure 23c). When receiving a danger report from BLE, if the device does not have any GPS signal, the report will be discarded.



(a) BLE scanning for nearby devices (b) BLE connection is established between phone and device (c) Confirmation to enable bluetooth while location is disabled

Figure 23: Mobile application BLE functionalities

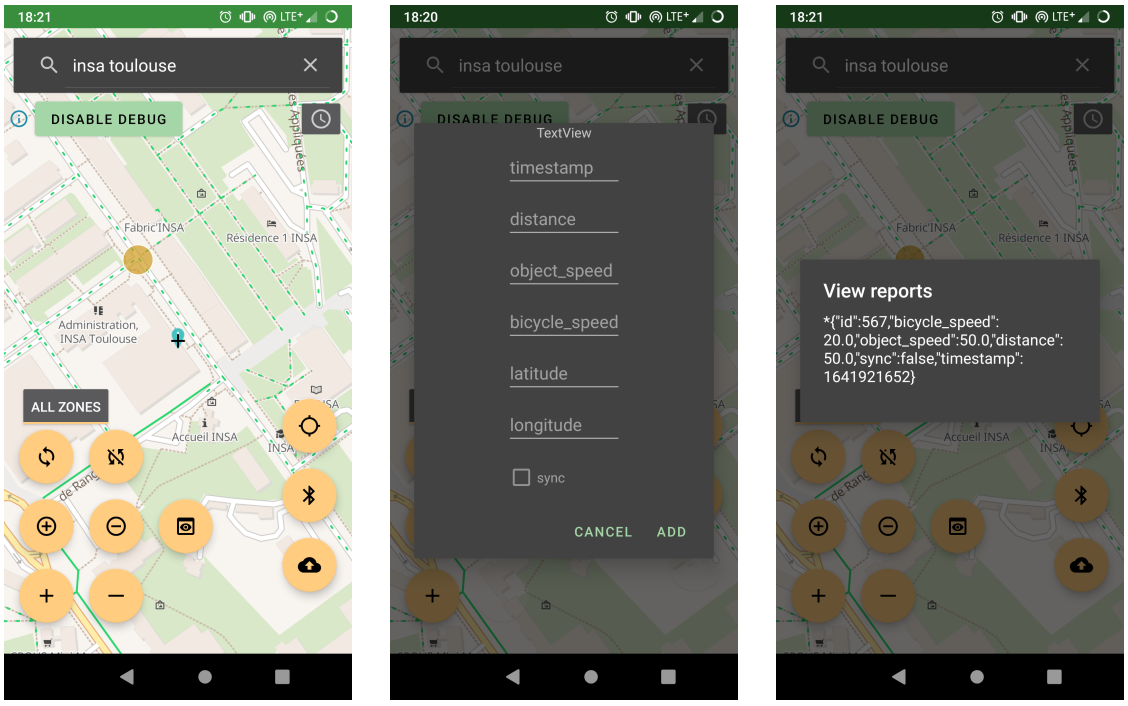
Debug To make testing easier, we added some debug features. We explained above that reports made without GPS signal were discarded. The current version will place reports without GPS signal in the center of the screen. The blue rectangle in figure 22d shows where our algorithm searches for danger zones when generating the route.

The button *Enable Debug* in the top left of the screen shows debug controls we used throughout the development to edit local reports. As shown in figure 24a, those controls show a cross-hair in the center of the screen.

The first button at the top labeled *All Zones* allows us to choose if we want to display danger areas taking into account the date, or using all reports available in the server, no matter their date.

The top row of buttons can simulate an upload by marking all local reports as synced with the server, or reverse the process. The next row allows adding a new report with the given properties using the interface shown in figure 24b, removing a report by giving an id, and showing all reports with their properties as in figure 24c. The last row of controls makes it easy to quickly add and remove reports. The first button adds a new report with pre-defined properties in the center of the screen, and the second button removes the last report.

All those debug features can be easily removed for the release variant of the app (the one uploaded to the stores), and kept only for the debug variant (the one used while developing) thanks to the build variants feature in Gradle, the build system used in Android.



(a) Debug controls (b) Add a custom report (c) Dump reports

Figure 24: Debug tools

Our choices Creating a mobile application using a custom map implementation is a challenge. We thus had to make important choices regarding the technologies to use.

Why Android and Kotlin? We chose to target only Android users to reduce costs and speed up development. We would need a costly license to work with iOS and would need access to physical devices. With Android, the tools are free and Open Source, and we can easily emulate devices. Targeting only one platform, we opted for native technologies instead of cross-platform, as it would be easier to work with low level APIs such as BLE and GPS. Kotlin was obviously the logical choice for Android development over Java because it has been the official language of Android since 2018. This proved to be a perfect choice as we were able to create a working prototype in a few weeks.

Why OpenStreetMap? For the map data we chose to use the open database OpenStreetMap as it offers more flexibility compared to alternatives such as Google Maps. When using Google Maps data, users are forced to use their map renderer. With OpenStreetMap, users can choose from several independent providers to retrieve the data, and render it using the renderer of your choice.

Why Geoapify? In our case, we chose the provider Geoapify to retrieve the map data. This web service also provides us with endpoints to do reverse geocoding and place auto-completion (retrieve coordinates from a part of a place name) and route calculation between two points. It has a free tier which was more than enough for our project.

Why MapLibre? As for the map engine, we chose MapLibre, a fork of the popular engine MapBox. MapLibre has the advantage of being a fully Free and Open Source Software (FOSS) compared to MapBox transitioning to a proprietary model, and completely independent from any provider. It simply needs to receive OpenStreetMap data to work, which Geoapify returns.

Why The Blessed BLE library? For the BLE, we used the library Blessed to add an abstraction layer over the low level Android API. This took care of most of the work and we were able to quickly have a working connection with the ESP board. In addition, this library allowed us to scan for specific BLE devices depending on the services that they offer. Using Android coroutines to manage asynchronous tasks, this library does not block the main thread and lets the application be responsive.

4.1.4 Server

How it works On the server side, the data is available graphically on a website and a REST API allows the application to retrieve and add information to the database.

More precisely, on the website, we can find a bar chart showing the number of reports according to the time of day (see figure 25) and also a area chart showing when a report is considered dangerous according to the current danger criteria (see figure 26).

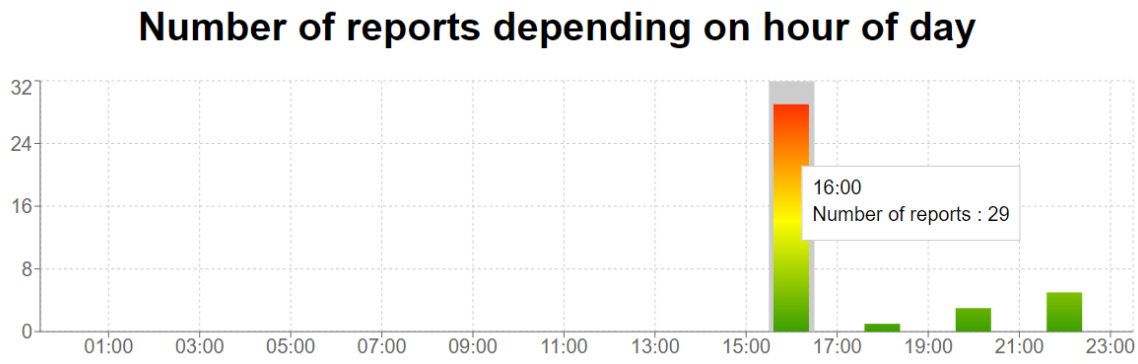


Figure 25: Danger reports graph depending of hour of day

As the figure 26 shows :

- First, we have a distance limit and a relative speed limit. We started considering as dangerous the reports where the distance was below its limit or the relative speed was greater than its respective limit.
- Then, we tried to avoid false positives by adding a minimal threshold, represented by the green part in figure 26. The threshold for the speed is used on both the relative speed and the absolute speed of the vehicle to avoid detecting stationary objects.

The REST API is used to retrieve the danger criteria, upload new reports, and retrieve the dangerous zones generated according to the database.

The details of the possible actions on those elements is listed in figure 27.

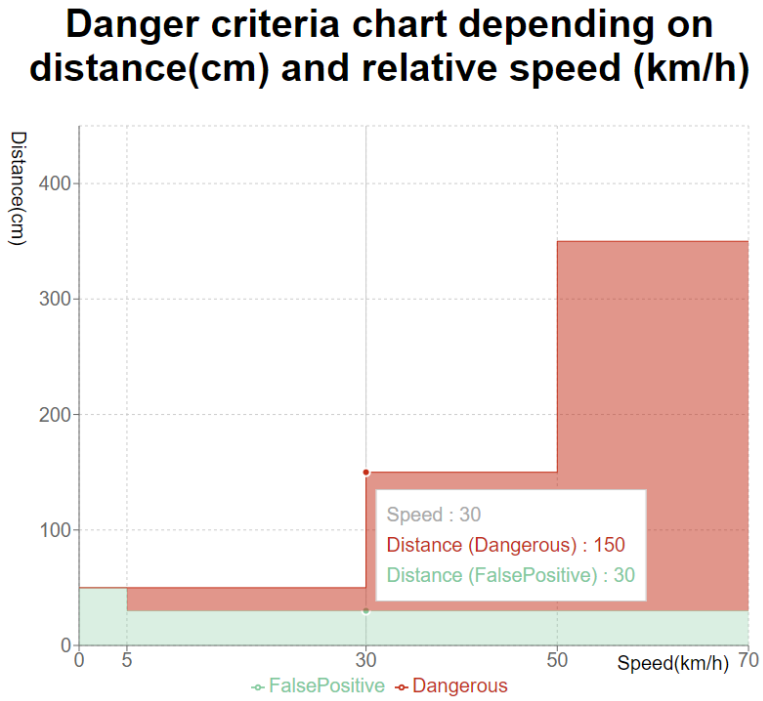


Figure 26: Danger criteria chart

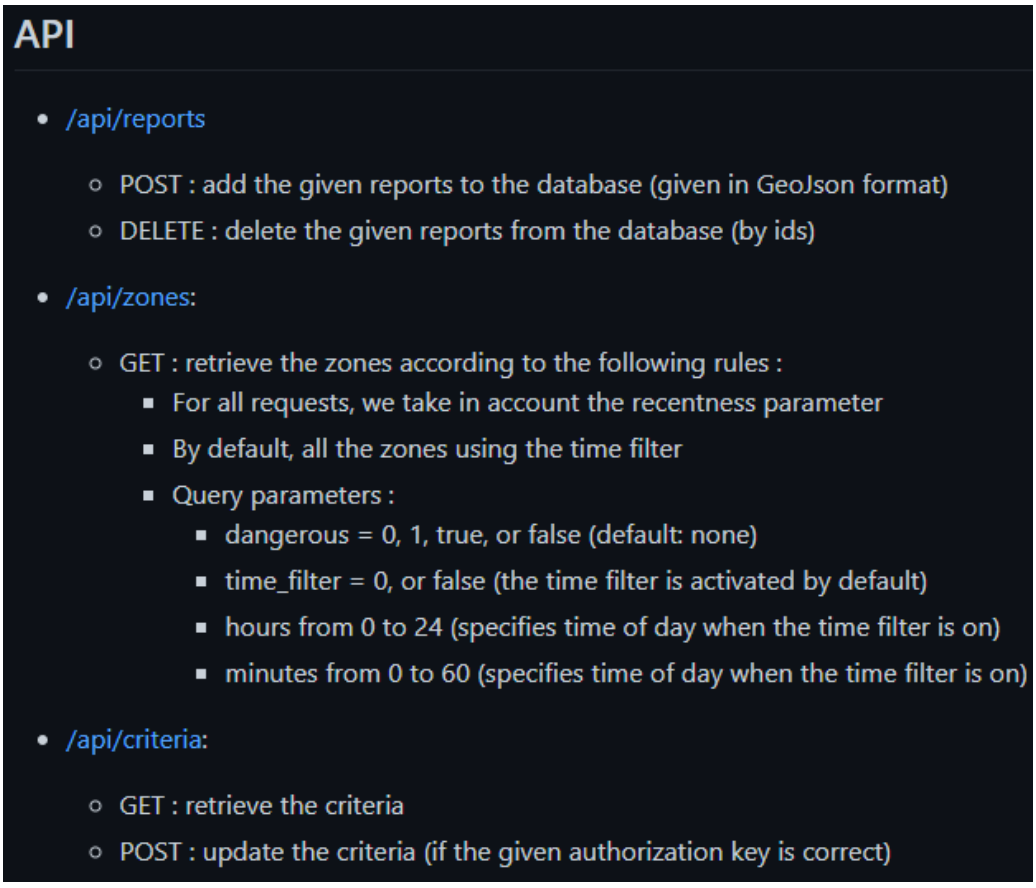


Figure 27: REST API available actions

Our choices Creating a server infrastructure requires the right hardware and software combination.

Why a Raspberry? We chose to use a Raspberry Pi 3 as the server, accessible through a public router under a name domain : https://rasp.pikouri.fr. This way, the website and the REST API are

very easily accessible. As an example before using the Raspberry Pi, to test the application we needed to have a common local WiFi access point between a phone and a computer where the server software was running.

Why MySQL? For the database, we chose to use MySQL and phpMyAdmin in order to easily manage the database. By accessing the following url https://rasp.pikouri.fr/phpmyadmin, we get to the phpMyAdmin interface, where we can access, modify, and delete data from the database.

Why NextJS and React? In terms of technology for server software, we chose to use the React framework Next.js because some team members had already used it to implement a REST API and a website and it was very quick and simple to develop. Next.js allows *server side rendering* which is a functionality that will render the pages on the server and send them as fully rendered HTML pages to the client. This makes the page loading duration very short.

Using Next.js we also have access to a library called Turf.js giving several operations on feature collections as needed in our case : the users upload reports (modeled by points), while we want to agglomerate this data and generate danger zones. We will see this point in further detail in section. 4.2.3

In Next.js the API routes are based on the file system, thus the REST API part of the project is well structured and organized.

We also used Recharts library to display dynamic, interactive and pretty graphs on the website.

4.2 What problems we encountered and how we solved them

4.2.1 Hardware & Power

During the prototyping and testing phases, the distance measure, the BLE communication and the library creation and coding went globally as planned. The coordination was very good and the deadlines set were respected. The main issues we had to deal with during the process were about:

- The distance measure using ultrasonic sensors ;
- The DC/DC conversion using buck converter modules ;
- The algorithmic process for the calculation of speed.

Distance measure using Ultrasonic sensors At the beginning of the prototyping phase, we decided to go with ultrasonic distance sensors because they were available at INSA and low-cost. In theory, such sensors are supposed to have a ranging distance of 5 meters according to the constructor data-sheet (GROVE) which we considered sufficient for the purpose of our application.

To confirm these characteristics, we ran a test of the sensors to measure the actual ranging. As shown in figure 28 tests show that the maximum effective distance capable of being measure is around 1m, which is not enough to fulfill the need of our system.

To overcome this situation, we were in the obligation of switching to a more efficient technology. Thus we ordered new LIDAR sensors, known for being more precise and more expensive at the same time. For this reason we chose entry level modules so our product remained in a reasonable price range.

As shown in figure 29, the finesse of the LIDAR sensor and the maximum range are considerably better than the ones measured with the ultrasonic sensor.

DC/DC conversion using a buck converter As mentioned previously, during the prototyping phase of the power supply unit, we ran tests on a buck converter to regulate the voltage coming from the dynamo to 5V. The results were not concluding. The output voltage was not stable and could not reach 5V.

To overcome this situation, we decided to use a Linear Voltage Regulator, which happened to work perfectly. Despite the fact that the regulator produced a lot of heat, we managed to lower this using a heat streamer.

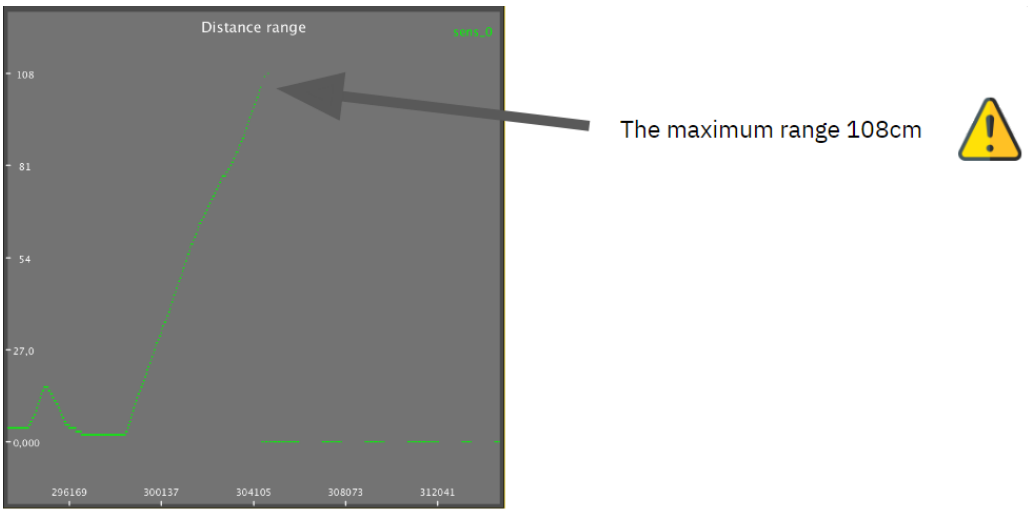


Figure 28: Distance range using ultrasonic sensors

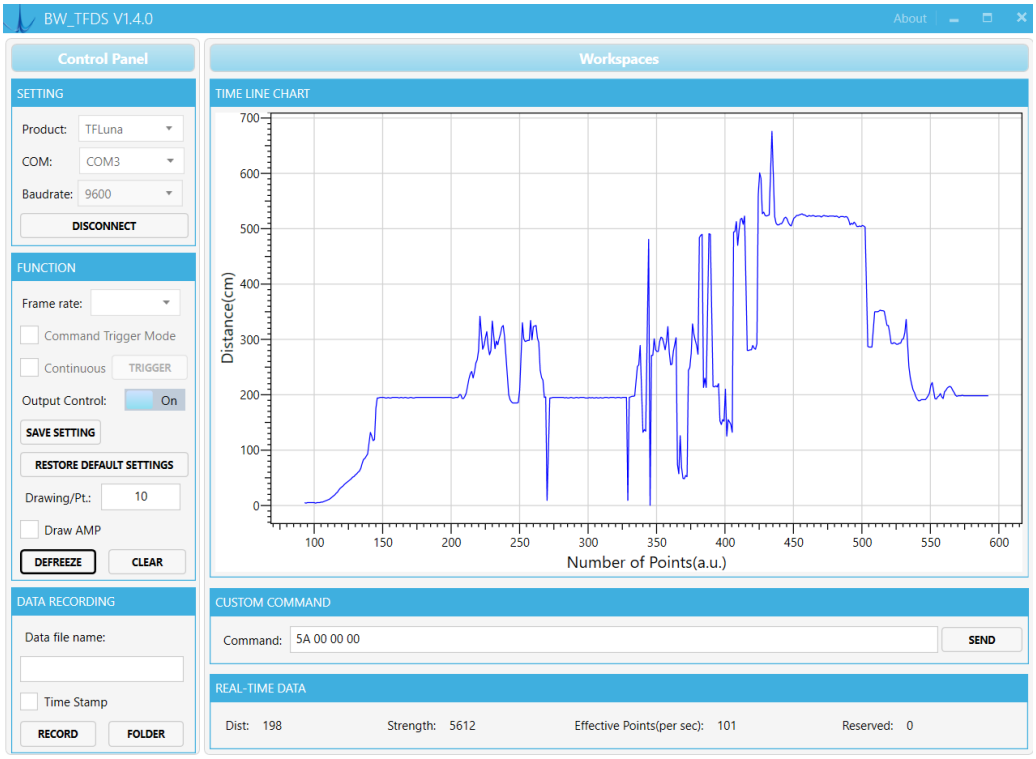


Figure 29: Distance range using LIDAR

Speed calculation algorithm During the two first sessions of "on-bike" testing, we encountered a problem related to the detection of false positive alarms.

To resolve the problem, we first made sure that the detection of objects only worked when the flag of the LIDAR positioned at the back of the bicycle was the first to be set to 1. This way, we avoided detecting static objects in the front and only detect vehicles passing in the same direction as the bike.

Second, we worked on the algorithm deciding if a vehicle is dangerous. We changed the thresholds and the conditions of detection according to what we considered as fair values (see section 4.1.4).

Finally, we limited the LIDAR modules range manually to 3 meters. That way, we ensured a second degree security to prevent the occurrence of false positive alarms.

4.2.2 Mobile app

The Kotlin language The very first problem we encountered was the language Kotlin. None of the team members working on the app had worked with it before, so we had to take the time to learn it. Thankfully this took very little time as Kotlin is very fast to learn with a background in other programming languages such as Java and Python.

Map data provider and engine The second problem was the choice of a data provider and map engine. There are many alternatives so we had to compare the different options. After some tests Geoapify stood out as a provider thanks to its ease of use and complete API, as well as its generous free tier usage.

For the map engine we chose MapLibre for the reasons stated above, but it proved quite difficult to work with at the start. MapLibre is a fork of MapBox version 9, before the switch to a proprietary license in version 10. Thus the documentation is very similar to MapBox v9, but it was difficult to find and not very explicit. The library worked well without issues, but it was difficult to find information online and to find the right documentation.

Routing We wanted to allow the user to create a route between two points, with the route avoiding dangerous areas. Creating the basic route was easy thanks to the Geoapify library. But avoiding zones proved problematic. The API does not natively allow avoiding zones, so we had to work with what the API offered. One useful feature was adding way-points the route had to pass through before reaching the destination. We used this to add way-points outside of danger zones if we detected the route went through one. This worked but resulted in unoptimized routes. In the end, we discarded the idea and simply warned the user of danger zones on their route.

Bluetooth Low Energy We encountered another problem when we started implementing the BLE connection in the application. At first, we did not use the Blessed library and tried to implement this feature using the low level Android API : we needed to configure precisely the BLE manager, and it was complicated to scan for a specific BLE device. Thus, we chose to use the Blessed library, and using a custom service ID that the ESP board advertises when looking for the phone, the application can detect the correct device and establish the connection.

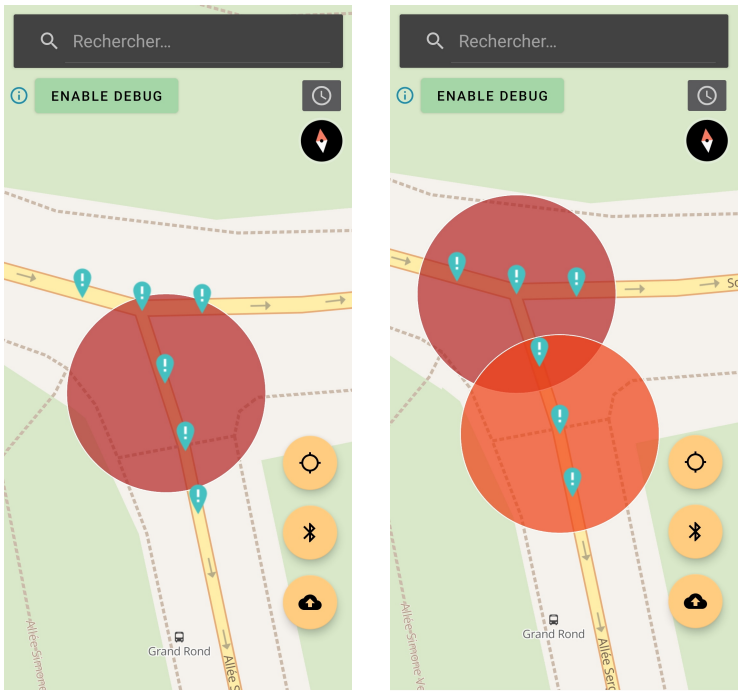
4.2.3 Server

Clustering reports As explained in the end of section 4.1.4, we want to aggregate the users' reports to generate danger zones (with their level of danger equal to the number of reports that were made in the zone). To do so, at first, we tried using the Turf.js library and more precisely its DBSCAN clustering feature to cluster the reports with a given minimal number of reports per zone, and a minimal distance between two neighboring reports. This method worked well and it generated optimal clusters when we had calibrated the two parameters. However, we figured out that there was a main problem with those clusters and the representation of the zones in the application. For instance, if we had several reports on a same road that were close to each other (approximately the minimal distance), these were grouped as one zone at the center of the road. As on the application side, we represent this zone as a circle with a fixed radius, some reports included in the cluster were not in the circle representing the zone and thus not taken in account when using the route feature (see figure 30a.

In order to solve this problem, we came up with another idea. A simple algorithm that:

- takes the optimal central report (that will also be the central point for the danger zone) that has the biggest number of neighbors
- clusters the neighboring reports which are within a certain radius (calibrated according to the radius of the circle in the application)
- and then loops over until there is no report left and only danger zones.

An example of danger zones generated using our custom clustering method is shown in figure 30



(a) DBSCAN clustering problem (b) Custom clustering example

Figure 30: Server-side clustering algorithms

5 Organization

5.1 How we planned the project

In terms of planning, we decided on a schedule at the beginning of the project using the GANTT chart in figure 31 which we followed as much as possible. We decided to manage our project using Agile methods with the help of the software *Trello*. This helped us to have a working prototype of each component very quickly. We also had periodical reviews during which we had to present our progress.

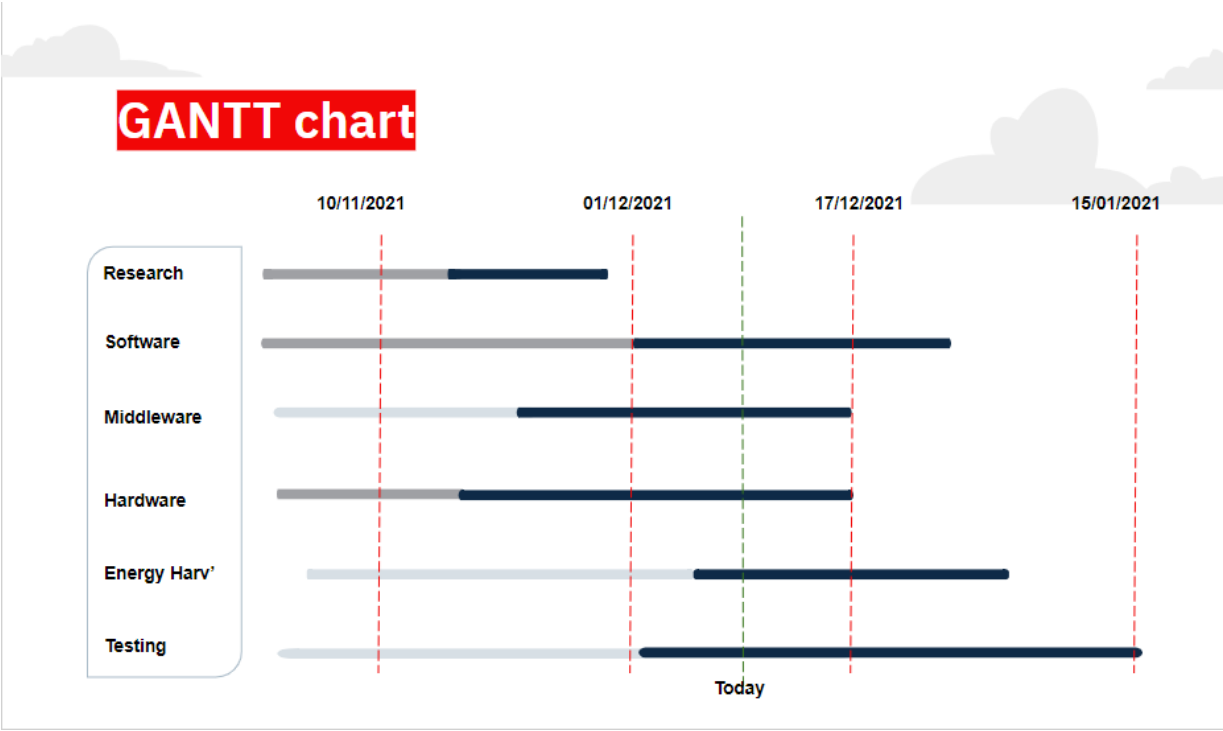


Figure 31: GANTT Chart of the project

5.2 How we organized the team work

Work was divided according to the affinities and background of each student. The students coming from Computer Science worked on the software (Server & Application) and the ones with an Electrical Engineering background worked on the hardware prototype.

Thanks to the use of Git and the platform GitHub, we were able to efficiently work as a team on the different parts.

We created a total of three Git repositories. One for the embedded software, one for the Android application and one for the Nodejs server. As the embedded software was composed of sub modules for each component (BLE, LIDAR, ...), each member worked on a different sub module in a separate branch. This allowed testing modules separately and when one was ready, it could be merged into the main branch.

Each repository contains a `README.md` file explaining what the repository is about, how it works and how to install it.

6 Conclusion

At the end of the project, we were able to create a working prototype acting as a proof of concept. It is not perfect and has room for improvements, but we are satisfied with the result.

Some of the improvements we can think of include improving the vehicle detection accuracy, by combining data from the LIDARs with a camera and image processing algorithms. We could also improve energy harvesting to make the device completely independent. Regarding software, we could setup our own routing algorithm to avoid dangerous zones. We could also improve the danger criteria by further using the device in real world scenarios.

As we aimed for a simple proof of concept, we decided not to focus on security. The communication from the mobile app and the server uses HTTPS, but this is the only security measure in place. To improve this, we could implement an account system to contribute new data, with bot detection and DDOS protection. We could also improve how the device is attached to the bike to prevent it from being stolen.

This was the first multi-disciplinary project this size we worked on, and we were able to learn a lot from it. We worked with new technologies such as LIDAR and Kotlin, which allowed us to learn new skills useful for our future. We also learned how to efficiently split tasks between us depending on our skills and to organize ourselves to deliver a working prototype on time.

To conclude we feel our project was a success because we achieved what we planned within the time constraint and we are very satisfied that we managed to work on the different parts of a fully autonomous IoT system.

This experience has been enriching on many levels and the prototyping phase was a real challenge. It made us find critical solutions in a short amount of time.

Finally, we can say that we received a huge feedback on what can be done and how depending on the environment we were working on. This made us more aware of what we could have handled differently and what improvements is yet to be added to the features of "Safe City for Cyclists".

List of Figures

References

[1] "Les accidents de vélo | fédération française des usagers de la bicyclette." (), [Online]. Available: https://www.fub.fr/velo-ville/securite-routiere/accidents-velo#_ednref (visited on 01/19/2022).

[2] L. JDD. "INFO JDD. Les accidents de vélo ont augmenté en 2020 à Paris," lejdd.fr. (), [Online]. Available: https://www.lejdd.fr/JDD-Paris/info-jdd-les-accidents-de-velo-ont-augmente-en-2020-a-paris-4026537 (visited on 01/19/2022).

INSA Toulouse
135, Avenue de Rangueil
31077 Toulouse Cedex 4 - France
www.insa-toulouse.fr

Université Fédérale Toulouse Midi-Pyrénées

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE
MINISTÈRE
DE L'ÉDUCATION NATIONALE,
DE L'ENSEIGNEMENT SUPÉRIEUR
ET DE LA RECHERCHE