INSTITUTO SUPERIOR TÉCNICO

REDES MÓVEIS E SEM FIOS

Final Report

AUTHORS

Name:	Carlos Cardoso	**Number:**	87161
Name:	Luís Coelho	**Number:**	90127

Group 10
Prof. António Grilo

2020/2021 - 2ND SEMESTER
5th of July 2021

1 Introduction

Finding a spot to park your car can be a frustratingly difficult task in big cities, you can easily waste a lot of time and patience trying to do it. Our project aims to solve this problem by developing a way to obtain real time information about available parking spots around the city and make that information available to the general public through an easy to use interface, as well as giving the users the possibility to reserve a parking spot.

This information along with the spot reserving option will be available through a mobile app which will be developed in this project.

In order to achieve the objectives of this project we will be implementing a sensor/actuator system. The sensor system will be sending our sensors' data to a cloud server which will be processing that data giving the users the possibility to access that information through the mobile app once connected to the cloud. The actuator system will work together with the cloud server in order to provide our users the possibility to reserve a parking spot through our mobile app.

2 Overall Architecture

In this section a general description of the project will be given, alongside a more concrete explanation about the use of each component, which will be further detailed in the following sections. We will finish this section with a Figure (1) representing the overall architecture of the system. This being said, considering the problem at hand, we decided to implement this IoT system using the following main components:

1. Back4App Web Server
2. Mobile Application built using Android Studio
3. ESP8266 NodeMCU
4. HY-SRF05 Ultrasonic Distance Sensor Module
5. TLS2561 Luminosity Sensor
6. LED

To better explain how we are planning to execute this system, we will follow the data flow in both directions (from the Sensor/Actuator Node to the user, and from the user all the way to the Sensor/Actuator Node).

The Sensor/Actuator Node is composed of the components listed as 3-6, and has the main purpose of determining the availability of a certain parking spot.

This sensor/actuator node will be the ESP8266 NodeMCU board connected with the HY-SRF05 Ultrasonic Distance Sensor Module, the TLS2561 Luminosity Sensor and the LED light. This board is to be placed on the pavement, in the middle of the parking spot, with it's sensors facing upwards, towards the sky. This allows the system to register if a car occupies a parking space, through the strong variation in light, captured by the luminosity sensor as data, and also through the variation of the distance to an object, captured by the distance sensor also as data. When a car is parked, this NodeMCU subsystem stays under the car, on the pavement (ideally in-ground sensors). The use of the LED will be discussed shortly.

The data captured by the NodeMCU system is then sent, in it's raw state, to a Back4App Web Server. This Web Server represents the bridge between the user of the mobile application and the Sensor/Actuator Node situated in the parking lot, and has the purpose of processing the raw data received by the sensors, determining the occupation state of a parking spot and filling a database with it's information. This database will have the location coordinates of each parking spot, their availability, the pricing of the parking spot, etc.

Ideally we would have one of this Sensor/Actuator Nodes per parking spot, however, due to the nature of this project, where only one Sensor/Actuator Node will be considered, the rest of the data needed to fill the database will be simulated through a Python script, which will produce raw sensor data to be also processed by the Web Server.

However, this Web Server not only receives data from the Sensor/Actuator Node, but also from the Mobile Application. The Mobile Application has an interface that informs the user with the location of the nearby free parking spots, which are obtained by periodically querying the database Web Server. The user can also interact with the Web Application by issuing a Reservation Command, which changes the occupation status of a given parking space from "Free" to "Reserved". Only a few parking spots can be reserved. When a Reservation Command is executed, there is a data-flow from the Web Application to the Web Server, which receives the command and updates the status of the respective parking spot in the database. Then, the Web Server also informs the Sensor/Actuator node of the respective parking space on the change, in order to activate the actuator, and essentially reserve the spot. Ideally, this would be done with some sort of motorized remote-controlled barrier, that would block the parking spot whenever a user reserved it, however, considering that we couldn't get our hands in such an actuator, we opted for a LED that was turned ON whenever a parking spot was reserved. When the user finally reaches the parking space, he can access the application once again in order to lower the barrier (turn off the LED) and park the car. When the car is parked, the Sensor/Actuator Node updates the Web Server on the status of the spot. Also, a user may only reserve one parking spot at a time.

The overall architecture of this Internet of Things system is represented in Figure 1, where the arrows represent the direction of the data-flows in the system.

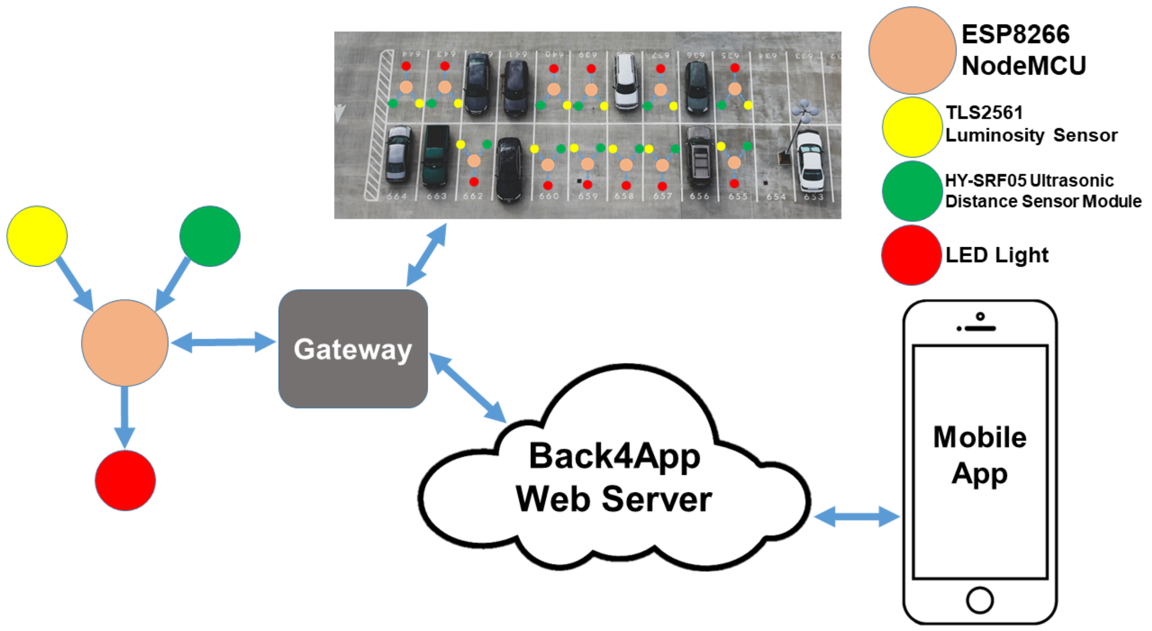

ESP8266 NodeMCU
TLS2561 Luminosity Sensor
HY-SRF05 Ultrasonic Distance Sensor Module
LED Light
Gateway
Back4App Web Server
Mobile App

Figure 1: Architecture of the Sparking Internet of Things Application.

We can also note that, in every parking lot there will be a gateway or router, to allow the Sensor/Actuator system to communicate with the Back4App Web Server.

3 Performance requirements

In order to implement the proposed Internet of Things system, several existing studies and products on market were considered. By doing this, we were able to determine some expected performance requirements associated with a Smart Parking application. The data collected can be found in the Table 1.

Table 1: Performance Requirements of the Smart Parking Application

System Requirements	Description
Delay Bounds	In order to keep our system as real-time as possible, we are aiming to reach a 5 second delay bound, while updating the status of each parking spot.
Required Message Rates	The system should send keep-alive messages every 15 minutes and should also send a message every time the occupation status of the parking space is changed. Both messages are to be sent to the Web Server.
Required Message Sizes	The message content size should be at most 20 bytes in size.
Required Communication Range	Considering the range of the ESP8266 NodeMCU, the communication range of this system should be around 250 meters. This range can be extended using repeaters or by using an external antenna, other than the PCB antenna included in the module.
Autonomy and Maintenance	The system is battery-powered and should have an autonomy of at least 5 years. As for the maintenance, ideally the system should be able to function without human intervention, or maintenance, for the duration of the autonomous period.
Accuracy/Precision/Resolution of the Data	A data accuracy of 99.7% should be pursued. Considering the sensors used, they all should produce data with the highest resolution possible (0.3cm for the HY-SRF05 Ultrasonic Distance Sensor Module and 0.1 Lux for the TSL2561 Luminosity Sensor) and have optimal precision in our desired use range (up to 50 cm for the HY-SRF05 Ultrasonic Distance Sensor Module and up to 40000 Lux for the TSL2561 Digital Luminosity/Lux/Light Sensor). Thankfully, every sensor used fulfills this requirements.
Support of Mobility	This system does not need to be mobile, as it's main purpose is to stay stationary in the parking spot.
Percentage of lost messages	Packet loss should be less than 10%.
Detection Range	The system should be able to correctly detect the presence of vehicles in a range of up to 50cm around the sensors.

4 Sensor/Actuator Node

4.1 Hardware

To implement the Sensor/Actuator system we used the following components:

1. ESP8266 NodeMCU
2. HY-SRF05 Ultrasonic Distance Sensor Module
3. TSL2561 Luminosity Sensor
4. Basic Red LED
5. Standard 9V Battery
6. 220Ω Resistor
7. Breadboard Power Supply Module Adapter Shield 3.3V/5V
8. BreadBoard
9. BreadBoard - Half Size
10. Jumper Wires
11. USB micro-B Cable

Where the use of the 220Ω resistor results from the presence of the LED Light in the system. The Breadboard Power Supply Module Adapter Shield allows to regulate the 9V of the Alcaline Battery to either 3.3V or 5V, as well as allowing for a more portable system. In this project we chose to use 5V, because it was the minimum voltage required for the HY-SRF05 Ultrasonic Distance Sensor to work.

The Fritzing diagram of this system can be seen in the Figure 2.

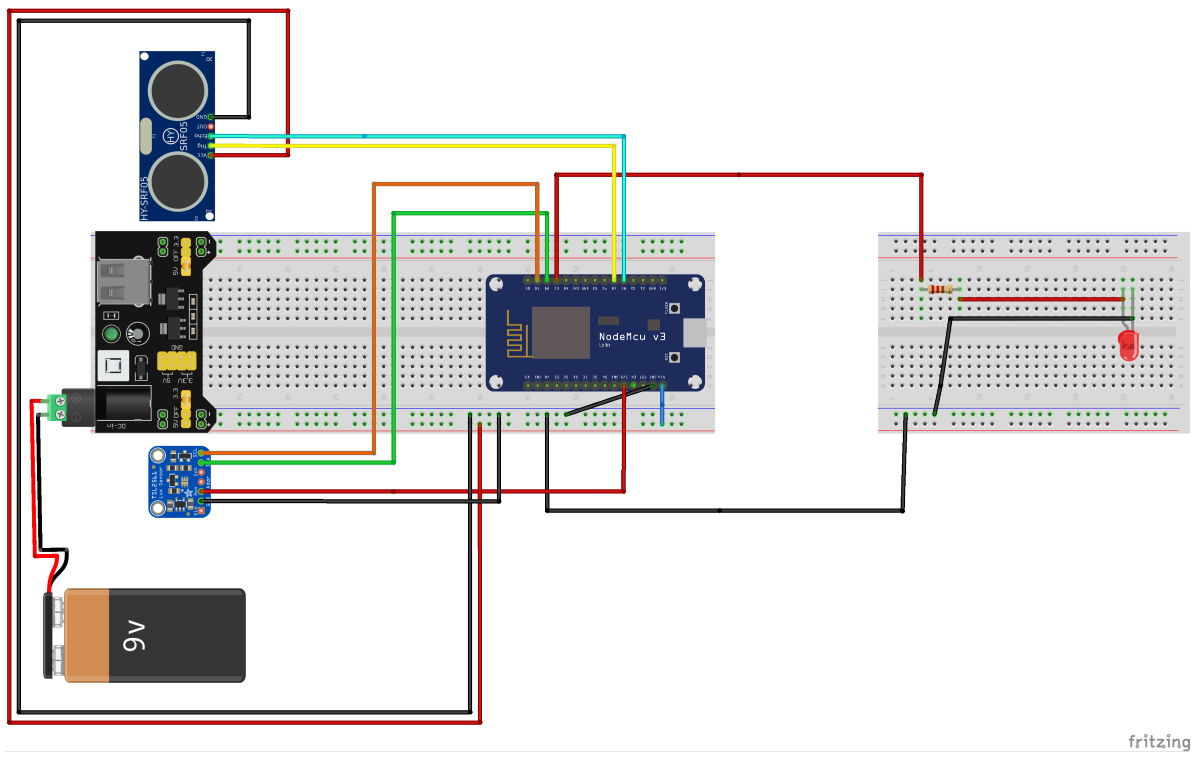


Figure 2: Hardware scheme of the ESP8266 NodeMCU subsystem.

4.2 Software

The software of our project will be spread across all systems. Inside the sensor system our software will be retrieving the data variables from reading the sensors and connecting and sending that data to the cloud server. This can be seen in Figure 3.

The actuator system will be retrieving data from the cloud server and acting accordingly with that data activating the actuator or deactivating it (state = high or state = low). This can be seen in Figure 3. All the software of the sensor/actuator system was created using the Platformio IDE.

The cloud will be receiving the data from the ESP8266 NODE MCU board and processing it accordingly. After processing the data the cloud server will update the respective information regarding the occupation and reservability of the parking spots. This can be seen in Figure 4.

As for the mobile app, it will be constantly connecting to the cloud server and retrieving the parking data accordingly. It will also change the Reserved state of the nodes depending on the users actions which will potentially influence the actuator system. This can be seen in Figure 5.

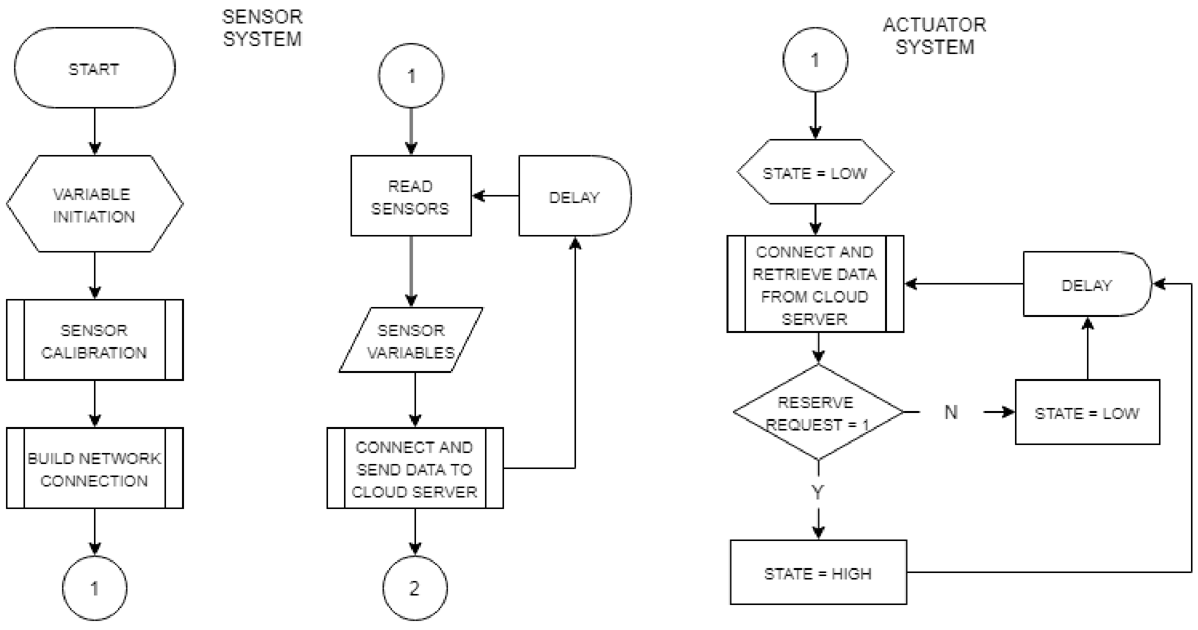


Figure 3: Sensor/Actuator System Flowchart

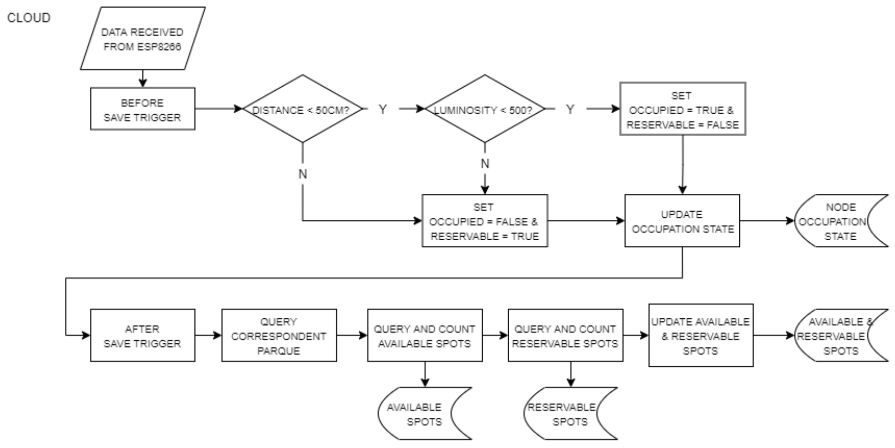

CLOUD
DATA RECEIVED FROM ESP8266
BEFORE SAVE TRIGGER
DISTANCE < 50CM?
Y
LUMINOSITY < 500?
Y
SET OCCUPIED = TRUE & RESERVABLE = FALSE
N
N
SET OCCUPIED = FALSE & RESERVABLE = TRUE
UPDATE OCCUPATION STATE
NODE OCCUPATION STATE
AFTER SAVE TRIGGER
QUERY CORRESPONDENT PARQUE
QUERY AND COUNT AVAILABLE SPOTS
QUERY AND COUNT RESERVABLE SPOTS
UPDATE AVAILABLE & RESERVABLE SPOTS
AVAILABLE & RESERVABLE SPOTS
AVAILABLE SPOTS
RESERVABLE SPOTS

Figure 4: Cloud Processing Flowchart

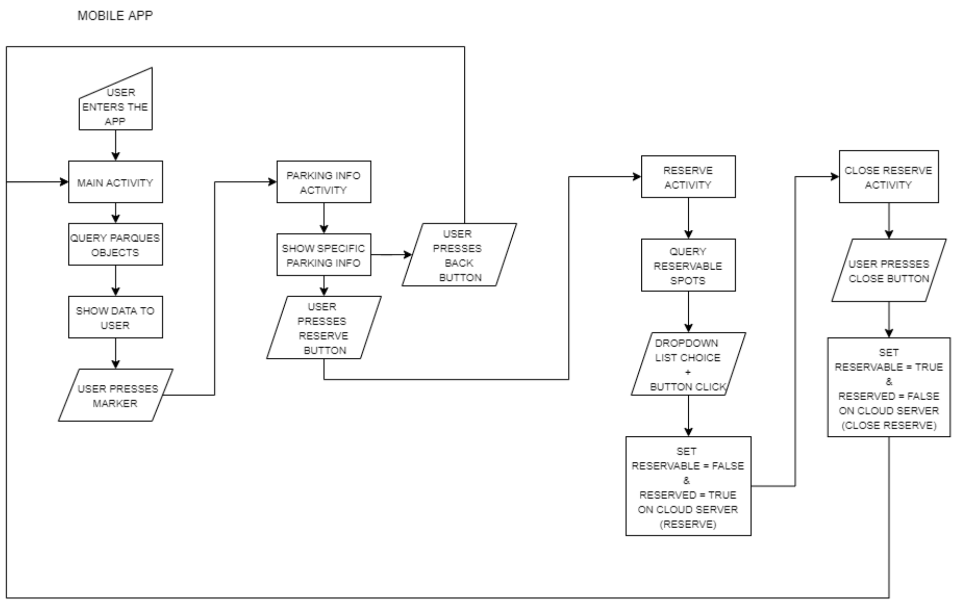

MOBILE APP
USER ENTERS THE APP
MAIN ACTIVITY
QUERY PARQUES OBJECTS
SHOW DATA TO USER
USER PRESSES MARKER
PARKING INFO ACTIVITY
SHOW SPECIFIC PARKING INFO
USER PRESSES BACK BUTTON
USER PRESSES RESERVE BUTTON
RESERVE ACTIVITY
QUERY RESERVABLE SPOTS
DROPDOWN LIST CHOICE + BUTTON CLICK
SET RESERVABLE = FALSE & RESERVED = TRUE ON CLOUD SERVER (RESERVE)
CLOSE RESERVE ACTIVITY
USER PRESSES CLOSE BUTTON
SET RESERVABLE = TRUE & RESERVED = FALSE ON CLOUD SERVER (CLOSE RESERVE)

Figure 5: Mobile App Flowchart

5 Cloud Server

For the Cloud Server of our project we've implemented a Parse Server [6] which is an open source Backend-as-a-Service(BaaS) framework. This Parse Server was implemented and deployed using the Back4App [1] service.

Since the sensor systems will be sending raw Data into the Cloud Server we will also need to do some Data processing inside the cloud. For this we will be using the Cloud Code [4] functionality which is built into the Parse Server.

From Cloud Code we've used Save Triggers [7] such as the beforeSave for example to determine if a node is occupied, given it's luminosity and distance variables, before saving each entry into the database making it so that we always have the updated node state. We've also used the afterSave trigger in order to update the number of available spots and the number of reservable spots after updating the node state.

There are many benefits of having this data processing happening inside the cloud. For example, because the data from our sensors is being processed in the cloud instead of in our ESP boards we will save our sensor-systems' battery energy. Also, because of this, if we want to update the way the node's state is determined we only need to change the code inside the Cloud instead of changing it in every node where we might have implemented our project.

The combination of our cloud implementation together with the mobile app that we've developed allows the user to retrieve the information about the available parking spots (reservable or non-reservable) from the cloud server. It also allows the user's interaction with the actuator-system in order to reserve a parking spot. This last interaction is made by having the user send a request from the app to the cloud server in order to reserve the wanted spot which will update the data in the cloud. The actuator-system will then detect that update and act accordingly, lighting up the LED and reserving the wanted spot.

These interactions between the mobile app and the cloud server were done using the Parse SDK for Android [5]. That subject will be addressed more thoroughly in the next section of this report.

For the structure of our cloud server we have created two classes inside the Parse Server Dashboard. The "Parque" and the "Node" classes.

Inside the "Parque" class have the following columns: "name" (String); "coordinates" (GeoPoint); "available_spots" (Number);"reservable_spots" (Number); "pricing_1hour" (Number); "pricing_4hours" (Number); "pricing_1day" (Number); "pricing_reserve" (Number).

Here the "coordinates" column corresponds to the coordinates of the location of a given parking space. The "available_spots" and "reservable_spots" columns correspond to the available and the reservable spots of a given parking space. The "pricing_x" columns correspond to the prices applied in each parking lot.

In order to store a new parking space we create an object of the class "Parque" and store the values correspondent to each column.

Inside the "Node" class we have the following columns: "name" (String); "Luminosity"(Number); "Distance" (Number); "Occupied" (Boolean) ; "Reserved" (Boolean) ; "Reservable" (Boolean) ; "Type"(String) ; "ParqueID" (String);

In order to store a new node we create an object of the class "Node" and store the values correspondent to each column. After the creation of the Node objects the Luminosity and Distance values are updated by the sensor-systems. This update will trigger the beforeSave function which determines if the node parking spot is occupied ("Occupied" equal to true or false) and updates the "Occupied" value accordingly.

Subsequently the update of the node's occupation state will trigger the afterSave function which will update the "available_spots" and the "reservable_spots" parameters from the corresponding "Parque" object with "name" (from the "Parque" class) equal to the "ParqueID" (from the "Node" class).

These updates are better illustrated in Figure 4.

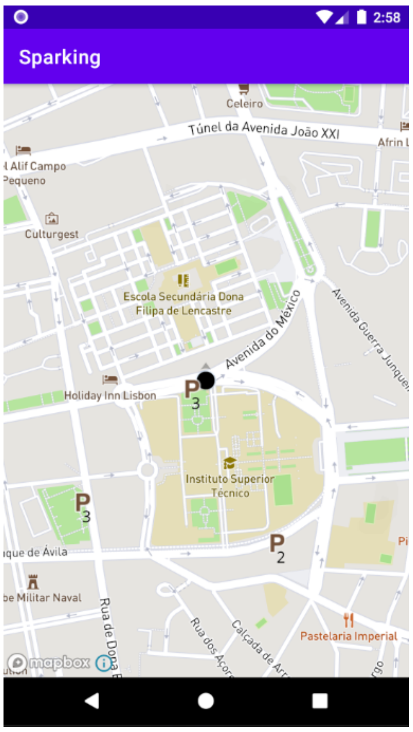
6 Mobile App

For this project we've created a mobile App for Android. This app was implemented by using Java throught the Android Studio SDK. In terms of libraries we've used the Parse SDK [5] for the cloud related operations, mapbox [3] for the map related functionalities and Glide [2] for image related functionalities .

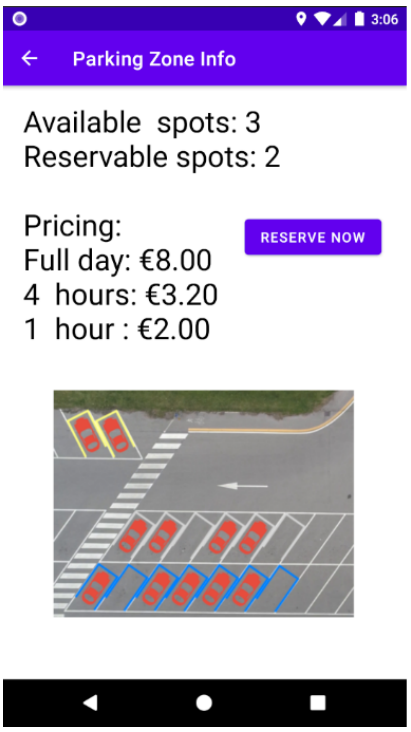
In terms of main functionalities our mobile app provides the users with the ability to check the locations of available parking spots and the ability to reserve a parking spot.

For the first functionality our app has the main activity layout (MainActivity.java) which contains a map that can be navigated manually by the user. This map contains the parking lots in the location area the user is viewing and their corresponding number of available parking spots. An example of this layout can be seen on Figure 6a.

Still inside the first layout, the user can click on each parking lot icon. Doing so will take the user to our next layout, the "Parking Zone Info" activity layout (parkingInfo.java). This layout can be seen in Figure 6b.



(a) Main Activity Layout


3:06

Parking Zone Info

Available spots: 3
Reservable spots: 2

Pricing:
Full day: €8.00
4 hours: €3.20
1 hour : €2.00

RESERVE NOW

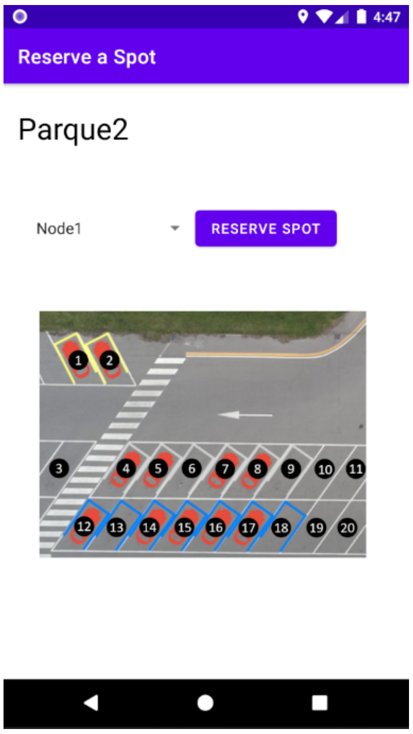
(b) Parking Zone Info Activity Layout

Figure 6: Activity Layouts

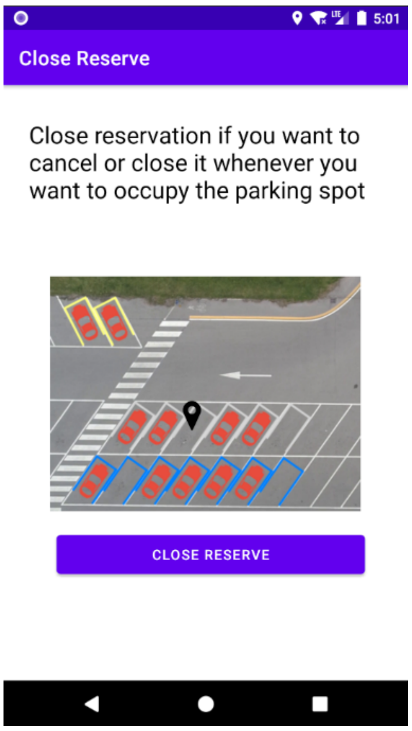
In this second layout the user can see the number of available and the number of reservable spots, the practiced prices and a map of the corresponding parking lot. In this layout the user can choose to press the "Reserve Now" button, if there are reservable spots available, which will take him to the "Reserve a Spot" activity layout (reserveActivity.java). This layout can be seen in Figure 7a.

In the "Reserve a Spot" activity layout the user will have a dropdown list with the available spots' names and will be able to choose one from that given list. Once the user chooses a spot, he/she can press the "Reserve Spot" button which will send the Reserve request to the cloud, changing the value of "Reserved" to true and the value of "Reservable" to false which will light up the LED, signaling that the spot as been reserved.

By pressing the "Reserve Spot" button the user will be taken to the "Close Reserve" activity layout. In this layout the user has an image of the map of the parking lot with the reserved parking spot indicated. The user also has indications in order to proceed with the reservation functionality. Still in this layout the user has the "Close Reserve" button which will take him to the main activity layout and close the Reservation changing the value of "Reserved" to false which will turn down the LED, signaling that the spot is no longer reserved. This layout can be seen in Figure 7b.



(a) Reserve a Spot Activity Layout



Close reservation if you want to cancel or close it whenever you want to occupy the parking spot

(b) Close Reserve Activity Layout

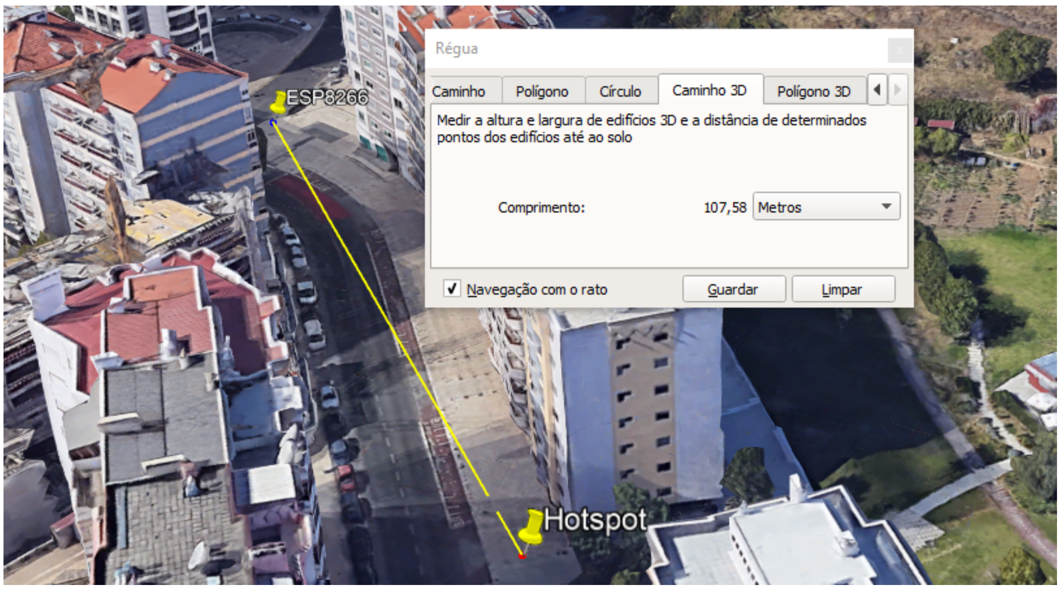
Figure 7: Activity Layouts

7 Results of the Tests

In order to test the performance requirements proposed on the third chapter of this report, we started by making in-door tests, in order to verify if the three main systems of our smart parking application were working correctly, and then we decided to take our system outside, to real parking lots, so that we could produce data, and check the performance of our system under a more realistic environment.

First of all, we confirmed that the delay bound between the moment the sensors read the state of a parking spot and the moment the new data was inserted in the database was always around the previously defined 5 seconds, 2 of which were introduced as the delay value inside the loop of the ESP8266's software and the other 3 resulted from the GET and PUT requests. In our in-door tests, we could always guarantee this delay requirement, however, unfortunately we could only guarantee this 5 second delay consistently for the first 20 meters, for our out-door tests. In the worst case scenario, when the ESP8266 was near it's maximum connection range with the hotspot, we got delay values of up to 25 seconds, which is far from optimal. This might have been caused by the attenuation or the distortion of the signal due to, for example, the reflections with other buildings, cars and the ground.

Speaking of the maximum connection range, from all the out-door tests we did, the largest distance between the hotspot and the ESP8266 module in which we could still consistently communicate with Back4App web server was a distance of 107.58m, as we can see in Figure 8.



Medir a altura e largura de edifícios 3D e a distância de determinados pontos dos edifícios até ao solo

Comprimento: 107,58 Metros

Figure 8: Maximum connection range obtained for the ESP8266 sensor/actuator system.

Considering the proposed value in the requirements chapter, we can conclude that the range of our system was far from optimal (250m). However, we have to consider that in our system, we did not use any external antennas for the ESP8266 board, while many of the manufacturers did, in order to allow for a greater coverage. Besides that, our hotspot, or gateway, was not optimal, because when testing our project we used our smartphone's hotspot functionality in order to connect to the Back4App web server. Maybe if a dedicated router was used, with a stronger signal and range, we could've obtained better results in both the maximum connection range and also regarding the delay bounds.

As for the required message rates, we assured that the ESP8266 system only sent data to the

Back4App web server when the occupation status of a parking spot was changed. This was done by registering the previous luminosity and distance values and comparing it with the current values. If the current value was equal to the previous, we chose to not update the database, however, if it was different, we would update the previous value with the current and we would also send it to the database with a HTTP PUT message. If by any chance our system spends 15 minutes without updating the Back4App web server, it sends the sensor's data to the web server, no matter their value, in order to signal the database that the ESP8266 is still operational. The message sizes were also only composed of 19 bytes, which met the performances requirements proposed (maximum of 20 bytes of content).

Regarding the requirements surrounding the sensor's data, we chose to use the highest resolution possible on both sensors, in order to insure the highest data accuracy possible. In order to test the data accuracy we used two cars with different ground clearances: the Dacia Sandero and the Dacia Duster. As for the Dacia Sandero, our sensors registered in average a ground clearance of 16.71cm which differs 3.4% from the real ground clearance value (of 17.3cm). On the other hand, the Dacia Duster, which was registered as having 21.6cm of ground clearance by our sensors, differed 5.37% from the real value of the ground clearance (20.5 cm). Although we could not hit the desired 99.7% data accuracy which is promised by other smart parking devices on the market, we considered the results to be satisfactory in order to detect the occupation state of a parking spot. We were also able to conclude that the system was able to detect the presence of vehicles in a range of up to 50cm by doing in-door tests with obstacles, and confirming the occupation status of our parking spot in the mobile application. Unfortunately we were not able to test the data accuracy given by the luminosity sensor.

Finally, as for the autonomy and maintenance system requirements, unfortunately we were really far from the 5 year battery life proposed in the third chapter. Considering that our battery had a milliamp-hour capacity of 3750mAh per mA, and that the average current of the system was 100.5 mA (70mA for the ESP8266, 40mA for the Ultrassonic Distance Sensor and 0.5mA for the Luminosity Sensor), we came to the conclusion that our system had approximately 34 days of autonomous battery life, by dividing the capacity by the average current. Perhaps another battery with a greater capacity could've been considered, or other components with less consumption could've also considered, in order to increase our system's battery life.

8 Changes relative to the Intermediate Report

There were several differences between the proposed project in the intermediate report, and the project presented in this final report, the biggest of which is related to the hardware. In the final project, the ESP32 Wi-Fi Camera sub-system was unfortunately scrapped from the smart parking project. As for the rest of the hardware, we opted to use a breadboard power supply instead of using voltage regulators, capacitors, and other components in order to obtain the desired voltage in each sensor.

In terms of changes on the Cloud Server instead of using the Firebase's Realtime Database service together with Cloud functions, because those services now require providing credit card data, we've opted on implementing and deploying a Parse Server using the Back4App service. For the cloud processing we used the Parse Server's Cloud Code functionalities.

As for changes on the Mobile App, instead of using the Firebase SDK we've used the Parse SDK which makes sense because we are no longer implementing a Firebase Server. We've also added the use of the Glide library for managing and loading images.

Still in the mobile App topic there are some differences between implementations in terms of layouts, those differences can be easily seen in Figures 6 and 7.

We have also made changes regarding the logic of the software implementations (mobile app and cloud processing software). These changes can be checked in Figures 3, 4 and 5.

9 Conclusions

In terms of objectives we've implemented most of the proposed functionalities for our project, even though unfortunately we had to let go of the idea of implementing the image related system using the ESP32 Wi-Fi CAM.

Despite this, we still managed to build a solid mobile App with an easy to use interface and a fairly easy to maintain Cloud Server. The fact that we are processing the Data inside the Cloud Server through the use of the Cloud Code's Save Triggers we make our project more easily scalable. This means that if we want to expand our project to more nodes we only have to program each of those nodes once because the nodes' interaction with the cloud server will always be the same (send the variables data and retrieve the "Reserved" state).

The implementation of the cloud server, even thought it may look simple it was a lot harder than it looks to implement. Mainly because the Parse Server functionalities are not the most documented nor the most troubleshooted ones.

In terms of software, the way we've determined the "Occupied" state was by comparing the values of the Luminosity and Distance variables with fixed predetermined values. This makes sense for the Distance variable, but since the Luminosity value will vary throughout the day, we've considered the possibility of having one or two nodes (for redundancy) that would provide a default Luminosity value which we would then compare with each node's Luminosity variable to determine the node's occupation.

In terms of hardware, we concluded that it is very difficult, especially in a real context, to get the standard performance for each component of our system, especially regarding the communication range of our nodeMCU board. We concluded that, due to phenomenons such as free space attenuation, obstacle attenuation, reflections, or other factors, it's really hard to obtain a range big enough to cover a parking lot. This phenomenons, combined with the fact that the system is to be placed under the car, difficults tremendously the connection between the gateway, and the sensor/actuator system. Because of this, an external PCB is extremely recommended. Another factor of great importance is the battery life of our system, and because of that the consumption of each component must be taken into account when designing an application of this kind. We also concluded that nowadays, due the new virtualization technologies, the hardware of an IoT system should focus on only generating real world data or actuating upon data that is given to it. In order to allow for a scalable solution, the hardware should be the simplest possible, allowing for the cloud to process the data.

Even though we've developed what we consider to be a pretty complex project, there is still room for improving it by refining some implementation details and adding possible functionalities.

We conclude by observing that this project provided us with a good general knowledge of what it takes to build an IoT system. From the hardware development in order to obtain data from the real world, to the communication with a web server in order to store and process that data (in this specific case a cloud server), to the communication of the mobile App with the web server in order to visualize that given data and close the cycle by interacting with the hardware's actuator-system.

References

[1] Back4App. *Back4App: Low-code backend to build modern apps.* URL: `https://www.back4app.com/`. (accessed: 06.06.2021).

[2] Glide. *Glide: a fast and efficient open source media management and image loading framework for Android.* URL: `https://github.com/bumptech/glide/blob/master/README.md`. (accessed: 06.06.2021).

[3] mapbox. *Maps SDK: a toolset for displaying maps inside of your Android application.* URL: `https://docs.mapbox.com/android/maps/guides/`. (accessed: 06.06.2021).

[4] Parse. *Cloud Code: code ran in the cloud server.* URL: `https://docs.parseplatform.org/cloudcode/guide/`. (accessed: 06.06.2021).

[5] Parse. *Parse Android SDK: Parse for Android.* URL: `https://docs.parseplatform.org/android/guide/`. (accessed: 06.06.2021).

[6] Parse. *Parse Server: an open source backend.* URL: `https://docs.parseplatform.org/parse-server/guide/`. (accessed: 06.06.2021).

[7] Parse. *Save Triggers: cloud code triggered by save events.* URL: `https://docs.parseplatform.org/cloudcode/guide/#save-triggers`. (accessed: 06.06.2021).