Technical Report

Project 4 - Mobility platform in Aveiro Tech City Living Lab Infrastructure

University of Aveiro

DETI

Computers and Informatics Engineering Project

Grupo 2

Pedro Lima, 97860

Nuno Cunha, 98124

Filipe Silveira, 97981

Bernardo Kaluza, 97521

Alexandre Gago, 98123

Ana Rosa, 98678

Contents

1 Introduction

As many of us may have already experienced, the public transports in Aveiro are outdated and suffer too many delays, as well as the problem with rush hours in the center of the city. The goal of our work is to improve these public transports, in specific the buses and the traffic in Aveiro in general.

Nowadays only the schedules of the buses are available to the public. So our idea is to develop an app that tracks the buses, presents the lines, stops and shows the expected arrival times of the buses, and shows a map with the density of traffic on specific roads.

To achieve this we already have a base work ATCLL - Aveiro Tech City Living Lab, that we will use, this work already provides us with the hardware and software that allow us to know the current position of the buses. ATCLL also provides us with stored data that allow us to improve our calculations for the expected arrival time and also possesses the tools (radars) for the detection of numerous vehicles which will be used to calculate de density of traffic.

For relevant work that is being used for the project, we are using Ana Filipa's thesis (ref. 1) about Using mobility data to estimate bus arrival time in a smart city.

After the introduction, there are 6 more chapters in this report. In chapter 2 we have the state of the art, which describes the current infrastructure of Aveiro and the sensors used throughout the whole city, as well as work done by a previous master's student in the subject of bus prediction and other applications in the public transport field. Chapter 3 presents the requirements elicitation and the system's requirements. The final chapter in the planning phase is chapter 4, which describes the system architecture planned for the beginning of the work.

The final 3 chapters are dedicated to explain the process behind the development. Chapter 5 explains how all of the project was implemented, divided in each main section of the work. Chapter 6 shows the results obtained, with screenshots of the finished product. Lastly we have the conclusion in chapter 7 to resume our accomplishments and some future work.

2 State of the art

2.1 Aveiro Tech City Living Lab

Aveiro Tech City Living Lab (ref. 2) is a large-scale advanced infrastructure throughout the city of Aveiro. This infrastructure integrates people, through their mobile phones, sensors, and vehicles, such as cars, bicycles in the city, and "moliceiros" in the Ria of Aveiro.

The technological laboratory uses mobile and geolocation sensors installed in vehicles, such as buses and municipal vehicles, bicycles, in the moliceiros of the city of Aveiro and designated spots in the city. In addition to the physical communication infrastructure, the Aveiro STEAM City team is implementing radars, LIDARs, video cameras, and IT units. The information coming from these devices provides insights into the flow of people, aiming to obtain new solutions in public transport, bicycle-sharing systems, critical safety systems, to identify problems and optimize mobility in the city.

In a nutshell, this infrastructure allows the development of an authentic tech city living lab for the experimentation of future services and applications in a real environment.

2.1.1 Infrastructure

To understand how Aveiro Tech City provides all this development, it is necessary to consider the infrastructures that comprise it. This innovative platform comprises a physical infrastructure and mobile units.

Physical Infrastructure

The Aveiro-open-lab infrastructure offers a wide range of multi-technological access points spread throughout Aveiro. Each access point supports various technologies to connect between distinct

types of users and terminals. The infrastructure also gives processing power to the edge and core of the network.

Furthermore, each station contains 16 km of fiber, and Radio access points will be placed in buildings (partnership with the City of Aveiro) and Smart Lamp Posts.

In terms of communication and communication, there are the following components:

- Software-defined Radio (SDR)
- 4G/5G C-RAN (future xG)
- Vehicular communication (IEEE 802.11p, C-V2X)
- Backscatter Communication (passive radar)
- LoRa
- Wi-Fi
- Lidars
- Radars
- Computing Units

These features referred to in the points above are available on smart lamp posts and murals. The equipment of the murals is very similar to the smart Lamp Post. Smart Lamp Posts integrate 4G/5G, vehicle communication to vehicle and infrastructure – ITS-G5, C-V2X, LoRa, and LoRaWAN long-range communication, and Wi-Fi, while the murals contain Wi-Fi, WAVE, flexible radio connectivity, based on Software Defined Radio technology including 4G and 5G and spectral probe.

Smart Lamp Posts (SLP) are posts placed all around the city that the Aveiro-open-lab infrastructure currently monitors and retrieves data from the various features they contain.

Figure 2.1 - Smart Lamp Post in Instituto de Telecomunicações (ref. 2)

The sensors that are being integrated into the testbed of these posts are environmental sensors, mobility sensors such as radars, and LIDARs.

All stations, integrated with the various communication technologies already mentioned, which are spread across the city of Aveiro, are interconnected by fiber.

This infrastructure is linked to the data center at the Instituto of Telecommunications, with computing and data aggregation units on the edge and in the cloud.

Regarding the murals, they integrate fiber-optic (point 3 of figure 2) access points that allow the connectivity of the various devices. It should be noted that the infrastructure has a 4G/5G SDR pilot testbed. The murals also contain antennas (point 1 of figure 2), a radio and processing unit (point 2 of figure 2), and a power management unit (point 3 of figure 2).

Figure 2.2 - Murals (ref. 2)

The connectivity between each point of access and the network core (implemented in the datacenter) is assured by point-by-point high debt dedicated optic fiber connection.

Mobile Units

Mobile units are divided into mobile sensing units and mobile communication units. The mobile sensor equipment installed in public transport is composed of DCU (Data Collection Unit) and OBU (On-Board Unit). Bearing in mind that the DCU is included in the mobile sensing units and the OBU is included in the mobile communication units.

OBUs (On-Board Unit) are small devices that register the kilometers driven on a payable toll road. Are placed inside buses and can communicate various data relative to their respective bus data. This device can, for example, estimate the number of people traveling inside moliceiros.

DCU integrates Wi-Fi and LoRa communication to send GPS location, speed and heading, temperature and sound, humidity, UV radiation, and much more.

OBU integrates Wi-Fi, WAVE, and cellular communication to establish the connection with the RSUs (Road-Side Units) and the other vehicles.

Figure 2.3 - Mobile Sensing Unit (ref. 2)

Figure 2.4 - Mobile Communication Unit (ref. 2)

2.2 Application of public transportation in Porto

2.2.1 Mobile Application using Data from Porto

The mobile application referenced was designed by Ana Filipa (ref 1) and documented in her thesis (ref. 1) about “Using mobility data to estimate bus arrival time in a smart city”. The app uses data from the city of Porto to run its various features and was developed using Flutter. The mobile application uses the Google Maps plugin to incorporate the map widget.

The first view of the application (Image 2.2.1a) allows the user to find nearby stops concerning his location or interact with the map to find bus stops on other locations. By clicking on a map marker, the user visualizes information regarding the selected stop as well as the lines that pass through this stop and an estimation of the time of arrival of each line at that stop (Image 2.2.1b).

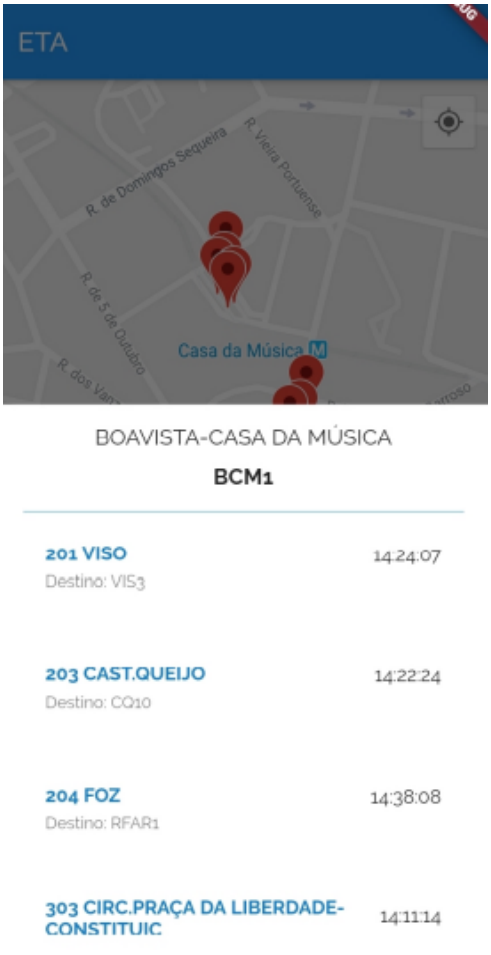


Figure 2.2.1 a): Map view (ref 1)

Figure 2.2.1 b): Detailed View (ref 1)

2.2.2 Bus Prediction API/AI

In Ana Filipa's thesis (ref 1), the Prediction API provides two endpoints to predict travel times between two bus stops, to serve predicted arrival times at bus stops given a context. The starting context is defined by four arguments that together specify a bus stop detection of a given line (line, direction, current-stop, current-stop-ts).

The fifth argument has a different purpose. It can return a prediction of the bus arrival time starting from the detection of a particular bus stop on a given route until reaching the provided destination stop, or it returns predictions in the given context, defining a starting point but forcing detection of the provided destination stop (since the bus line matching algorithm may miss consecutive stops) and making predictions for the remaining bus stops until the last one.

2.2.3 The current problems with the reuse of the AI

The problem that occurs with the data from Aveiro is that the bus lines aren't registered on our data, so we have no way of telling which line the bus is following. As the image below shows, in the center of the city, multiple lines overlap each other. Because of this, we have no way of telling what line the bus will continue after that overlap, that is before it reaches the next stop.

In the case of the mobile application for Ana Filipa's thesis (ref 1), the data used for the bus, referred to the line that it was on, so it was less troublesome to work with.

Figure 2.2.3: Aveiro city center Bus Lines (ref. 3)

2.3 Smart City projects

In this section we will present some smart city projects developed over the years in several countries.

The examples of smart cities presented in the sections are taken from the dissertation by André Mourato (ref 4), a former student at the University of Aveiro.

2.3.1 Building a Big Data Platform for Smart Cities: Experience and Lessons from Santander

SmartSantander is one of the largest smart city experimental testbeds in the world, developed in Santander, Spain. Within the SmartSantander testbed, more than 15,000 sensors (attached with around 1,200 sensor nodes) have been installed around an area of approximately 13.4 square miles in the city. These nodes are hidden inside white boxes and are attached to street infrastructure.

Communication among sensor nodes, repeaters and gateways are carried out through IEEE 802.15.4 interface, while gateways use Wi-Fi, GPRS/UMTS or Ethernet interfaces to connect with the SmartSantander backbone.

Like citymanager, but the real time and historic data is fetched from the same server, while the data in the CM is fetched from both the MQTT bridge (real time data) and from the TimescaleDB (historic data).

2.3.2 MLK Smart Corridor: An Urban Testbed for Smart City Applications

In this case, there is an Open Data Platform where researchers can access datasets generated by the testbed. In this smart city, data is only accessed in real time, and data is not stored from the sensors, but is used at that moment and discarded when updated.

2.3.3 Internet of Things for Smart Cities: Interoperability and Open Data

There is a line testbed in the city of Uppsala Municipality aimed at providing open data to all users.

The testbed, including the open data and open APIs, allows third parties to develop and experiment with new sensing products and services that could be exported to the international market.

2.3.4 Smart City architecture for data ingestion and analytics: processes and solutions

This smart city uses the concept of dynamic (real time) and static data. The input data comes from citizens, commuters, social media crawling, IOT sensors and city operators. The input data can provide detailed information about the city and its public services (available parking lots in any specific area, traffic flows and collapsing area, people flow, triage usage of hospitals, incidents in the streets).

Furthermore, it allows users (technical and non-technical) to detect early warnings about specific occurrences through the analysis of the historic data, and provides use cases for entertainment (booking tickets, etc.)

In this example, the Smart City API provides data to mobile, web and third-party services and there are 800 different datasets.

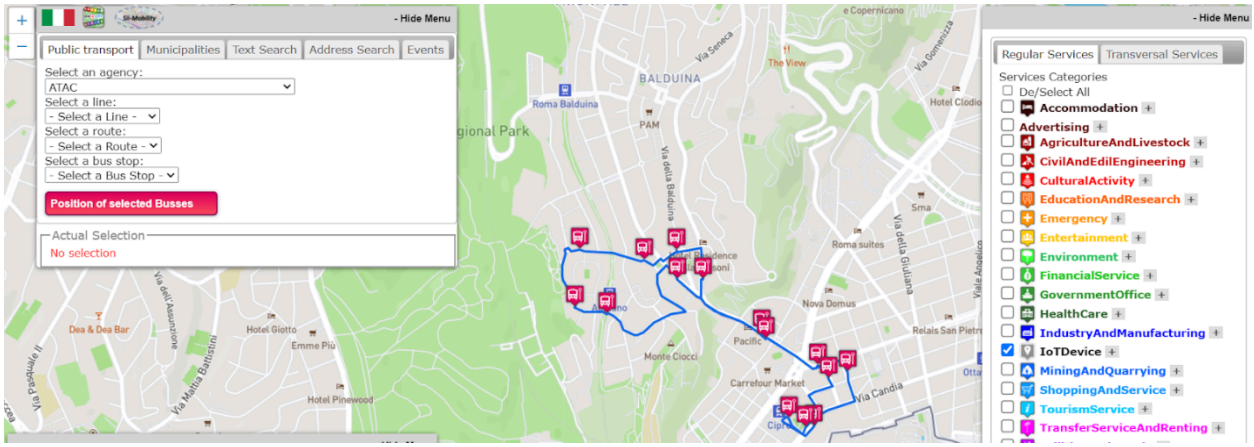


Figure 2.3.1 - Example of a smart city interface (ref 4)

3 System Requirements and Architecture

3.1 System Requirements

This chapter presents the system requirements specification. It has a description of the requirements elicitation, followed by a context description, actors, use cases, non-functional requirements, and the system's assumptions and dependencies.

3.1.1 Requirements Elicitation:

This specification process can be divided into three parts. The first one is the one in which it is determined the project's main goal, scope, boundaries, and usefulness to the public. The second one is the investigation of the state of the art, related works, studies, technology study, and understanding and the ways we can use those works to help speed up our work. These processes helped to improve the understanding of our project and its scope. By discovering works similar to ours, we can understand where things can go wrong, what is more difficult, what is essential to be done, what is already done, and in general, improve our system requirements. Finally, the third part is the functional requirements that are to be expected of each functionality present in the application. They go as follow:

- **Bus delay time:**

  For the user to get the delay of a bus it should be possible to visualize every bus on the map as well as each bus stop scattered around the city. This view of the city should be updated in real-time with data that comes from the servers. It should also be possible to have a menu to show all the circulating lines to select a bus on the map and see its course. Lastly, it should be possible to check how often a line gets late to the destination and if it's a long delay.

- **Live traffic of Aveiro:**

  Regarding the Live traffic of Aveiro, It should be possible to view Aveiro's map and get an estimate of how many people are passing on different crossroads as well as the traffic intensity on each road.

3.1.2 Context Description

This part suggests a description of how the system will be used by its different stakeholders. Firstly, a user should download the application via the Google Play Store or similar applications. Once the application is downloaded and installed, the user will be able to use it without the need for registration.

Once inside the application's interface, users can see Aveiro's map, with different buses, cars, and traffic intensity. The user may select different functionalities present in the app's interface: the traffic intensity, the buses delays. Once selected, the user may use them to help them get information about the topic selected and with that decide what they want to do. In the case of traffic intensity, the user may get information about the intensity of traffic in the streets of Aveiro. In the case of the bus delay tab, the user may get information about where the bus they are waiting for is located and an estimated time of arrival to the user.

3.1.3 Actors

The target user for the mobile application is any person in need of any service described above in Aveiro. The design is very accessible and there is almost no expected knowledge to be able to use the app properly.

The main actors are described as follow:

- **Users:**

  Any person who wishes to obtain information on the traffic intensity in Aveiro or obtain information about estimated bus arrival times can access the app to obtain that information. Users may also be able to view bus timetables.

- **Administrator:**

  Has access to all data, modifies the app, and fixes bugs.

- **Server**

  Will update the data necessary in the app periodically (real-time data and data from the database)

3.1.4 Use Cases

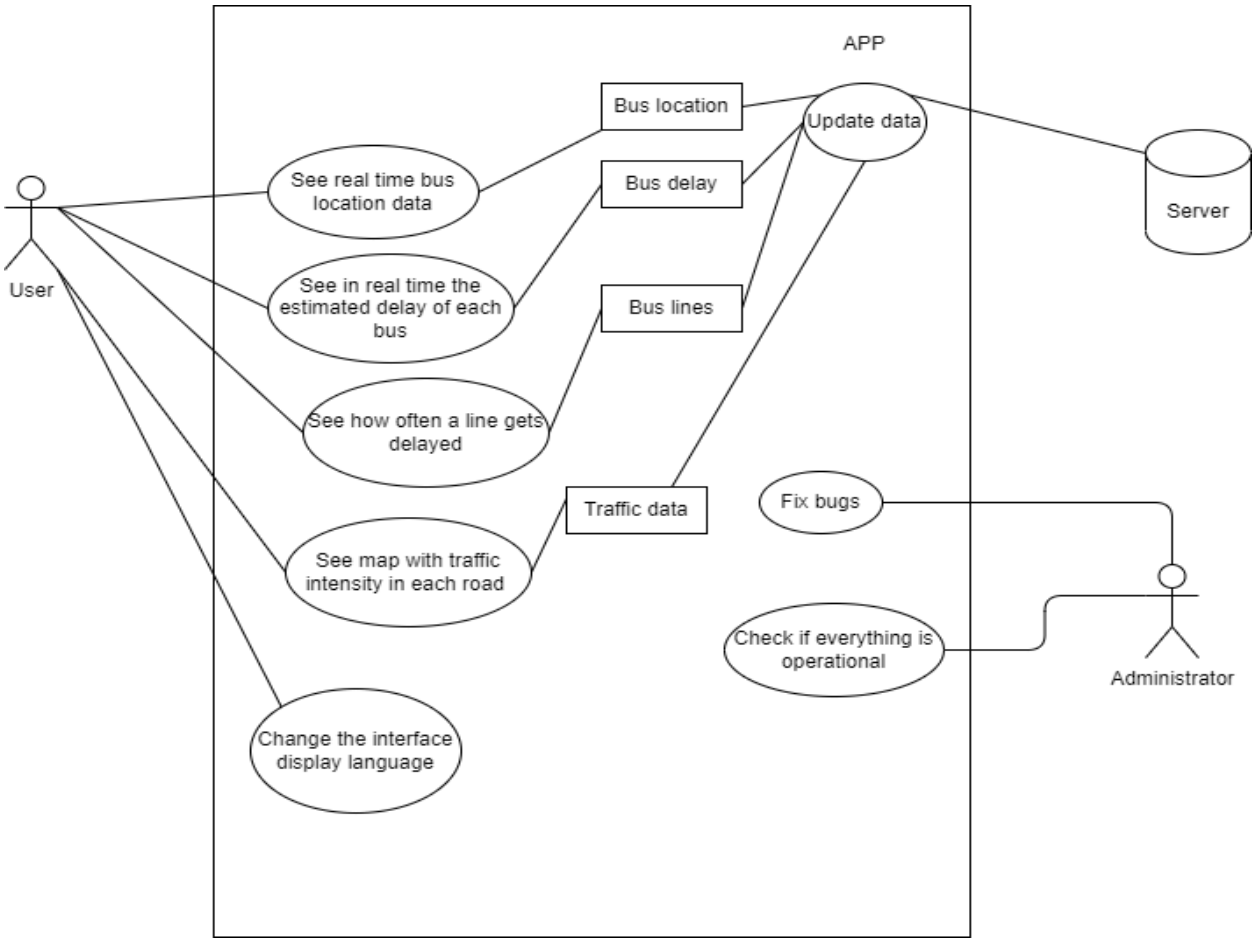


Figure 3.1: Use cases

Figure 3.1 shows a model of the architecture, as it is shown, both the user and the administrator interact with the application.

The user can perform the following operations:

- **See real-time bus location data:** A user will be able to see the current position of a bus detected in the infrastructure, for this the app will access the bus location data stored in the server.

- **See in real-time the estimated delay of each bus:** A user will be able to get an estimated time of arrival of a bus to a certain stop, as well as the delay compared to the timetable, for this, the app accesses the bus delay data calculated in the server.

- **See how often a line gets delayed:** A user will be able to get information about the average delay of buses doing a specific line, this is calculated with the bus line information on the server.

- **See map with traffic intensity on each road:** Users will be able to see a real-time map with color-coded traffic intensity for light, moderate and heavy traffic.

- **Change the interface display language:** Users can change the interface display language.

Administrators can perform the following operations:

- **Fix bugs:** Administrators will try to fix bugs with the service.

- **Check if everything is operational:** Administrators will check if the service is up and running at all times.

3.1.5 Non-functional Requirements

The following list presents requirements specifying non-functional requirements, i.e. a description of a property or characteristic that a software system must exhibit or a constraint that it must respect

- **Performance:**

  The mobile app will operate on a mobile device that nowadays has great power and resources. So, the app should give the data in real-time with little delay, it should be responsive and able to work in any device running android or IOS, it should load quickly, and lastly, the data processing should be done in a server to reduce the load in the device running the app. Priority: Medium

- **Usability:**

  Firstly, users should easily understand the interface of the app and it should be easy to understand and interact with. Secondly, the app's algorithms should be versatile in order to be used and implemented in other smart cities with similar sensors. Finally, there should be different language support. Priority: High

- **Security:**

  Given the fact that we are dealing with information that isn't harmful if stolen, we decided that security shouldn't be the focus, but it still would need to be considered. So, taking this into account, the app should be secure to keep each user's location private, the app should have little to no vulnerabilities and the data received by the app should only be the necessary data for the app. Priority: Medium

- **Documentation:**

  Given that there will be a new app with different algorithms, documentation should be created to help further improvements to the app, to help other projects if they need something similar to what we did, and to help to understand our work a little easier to those who need it. Taking all that into consideration, the documentation should be easy to understand, should state the sensors used and covered city spaces, should document the obtained data, the architecture, and

functionalities and finally it should have a user manual and a description of the functionalities.Priority: Medium

3.1.6 Assumptions and Dependencies

For the application to work as expected the following assumptions are made. Firstly, the user's phone should be Android, IOS, and they should have a GPS in them and have it be turned on. If this requirement is not met, it is impossible to know where the user is on the map and, as such, the user could only see the bus delay times by selecting the stops on the map. Secondly, the user's phone must have an internet connection, may it be wi-fi or mobile data in order to receive the data from the server. If this requirement is not met, It will be impossible for the user to get any data and as such, impossible to select any service offered by the app.

4 System Architecture

This chapter describes the overall structure and system architecture of the project, describing the domain model, the physical model, and the technological model.

4.1 – Domain Model

The domain model presented in Figure 4.1 describes the two main entities, the server (which will receive the data from the sensors and calculate the estimated arrival time) and the mobile app (which will receive the data from the server and show it), and the interactions they have with each other.

First, in our APP we have 2 main classes: UI and Coms_server.

The Coms_server class communicates with the server API making requests (about the estimated arrival time of a specific bus to a destination stop, and about the line a specific bus is on a certain day) and parsing the data received in JSON to the corresponding class. It will also make requests to receive the data about the current traffic on each road in a periodic way to keep the map updated.

This class will need a TOKEN, which is necessary for receiving real-time data.

The UI class will be the class that will contain all the different tabs used in the APP. Each tab will be a different method in the APP plus a method to ask permissions to the user (such as GPS localization). In the main map shown in the APP, it will be possible to visualize all the buses in range, stops, and corresponding lines in circulation, and in another map it will be possible to visualize the traffic in each road, represented by a different color for each intensity of traffic (light,moderate,heavy). These 3 entities (bus,stops and lines) ,as well as another to represent a road, are defined in classes to better organize the types of objects in the APP and will also be used in the server.

Second, on the Server-side we have 3 main classes, the API, AI and Traffic classes.

The API class will be used to communicate with the app, responding to the requests made by the app and giving a JSON dictionary with the data requested (such as the current line of a bus and the estimated time of arrival of a bus).

The AI will be responsible to receive the data from the sensors and calculate the estimated arrival time with that data. Periodically, this class will store values in a database used to improve the calculations with the most relevant data (method UpdateAIvalues). The last method is CalculateLines, which will serve to identify which line a bus is making on a certain day (because it's not guaranteed that a specific bus does the same line every day).

The Traffic class will be responsible for receiving the data from the sensors (mainly radars combined with the speed of the buses OBU) and differentiate all the roads detected, and their coordinates. After that, it will use that data and calculate each road's traffic intensity, to later send to the app.

The real-time data is received periodically from the sensors and is sent directly to the APP to improve the update of the positions of buses on the main map.

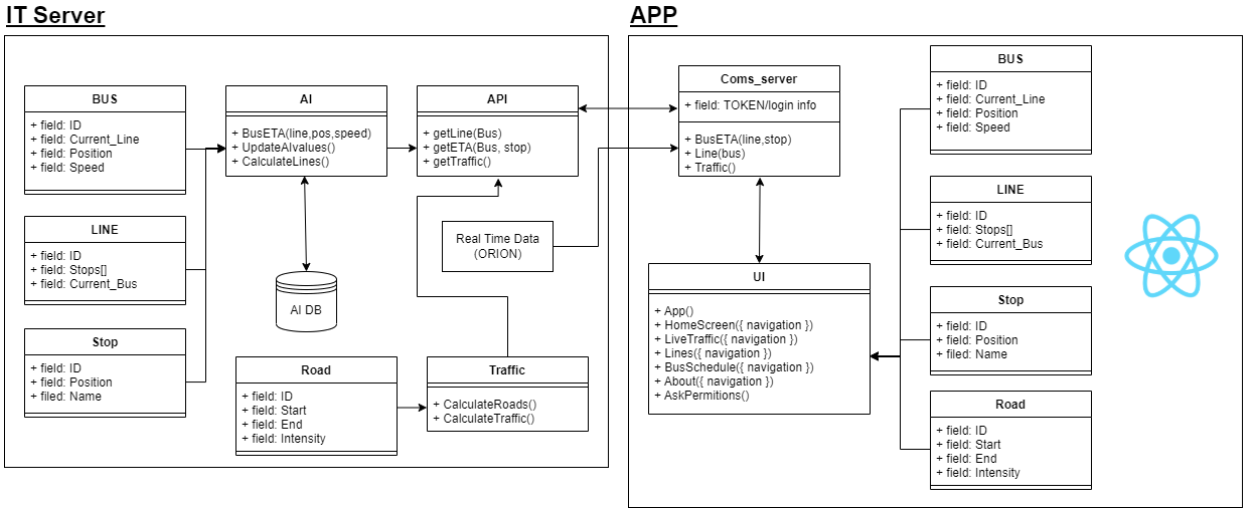


Figure 4.1: Domain Model

4.2 – Physical Model

The physical model of the ATCLL system is presented in figure 4.2, which describes how the physical components interact with each other. The model was divided into 3 groups: the app, the server/backend, and the sensors/RSU's edge.

The user's device will interact with the backend to receive the data needed for the app and on the other side, the backend/server will process the data received from the sensors (RSU's edge) and store it in a database for later use in the AI.

Lastly, the OBU's, when in range of an RSU, will send its stored data to the RSU and will continue to send in real-time data while in range.

Our part in this infrastructure is depicted in the blue circled "Service PECI".

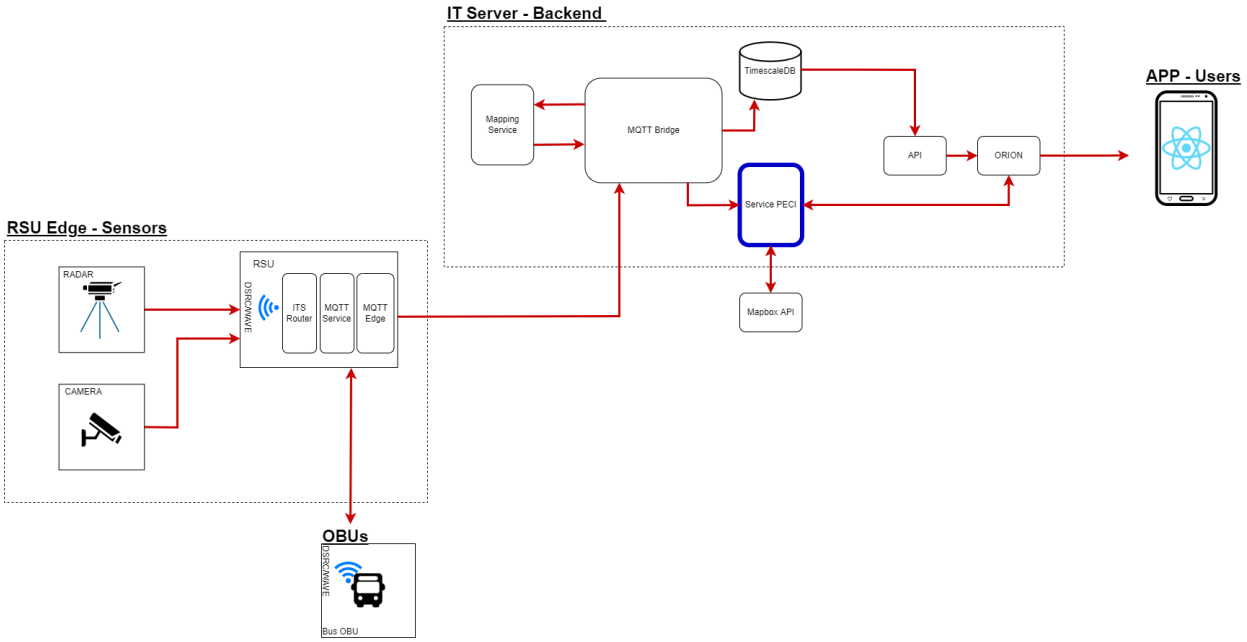


Figure 4.2: ATCLL's Physical Model

The physical model of the PECI service iis presented in figure 4.3, which describes how the developed models interact with each other. Starting with the Line Detection Model, it consumes data from MQTT about what buses are being detected in real-time and it consumes from ORION data about the history of real-time detected buses. It outputs the assigned line and direction to the bus to the Stop time Prediction module that uses the MapBox API which is a routing service to estimate arrival times to the next stops. This information is then saved in a Mongo Database and published to ORION to be later consumed by the app.

The other module is Live Traffic which consumes from MQTT real-time information of the radars in the city and based on the information reported by them, mainly the vehicle count and average speed, it outputs a color in a green to red gradient that indicates how congested is traffic in Aveiro. It outputs this color by comparing the real-time information to the daily average that is also calculated by this module.

This info is also stored in a Mongo Database.

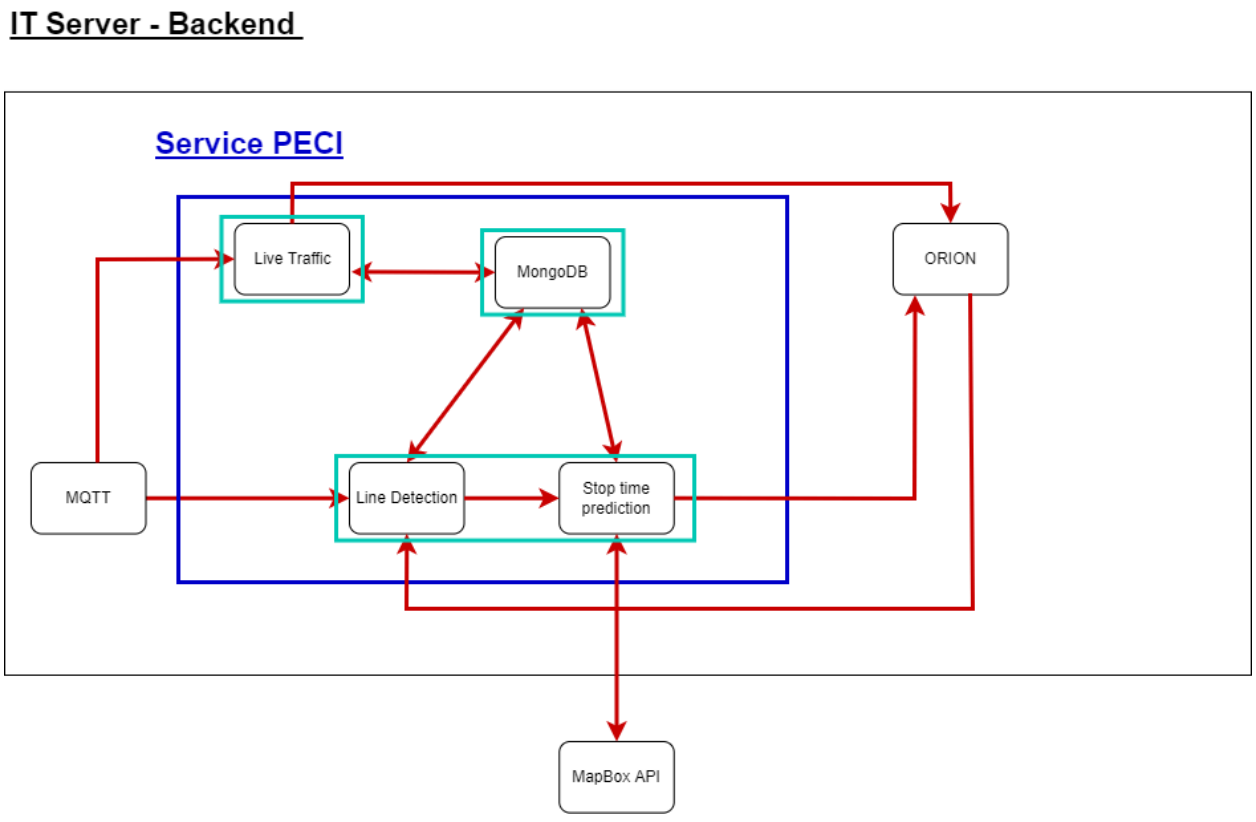
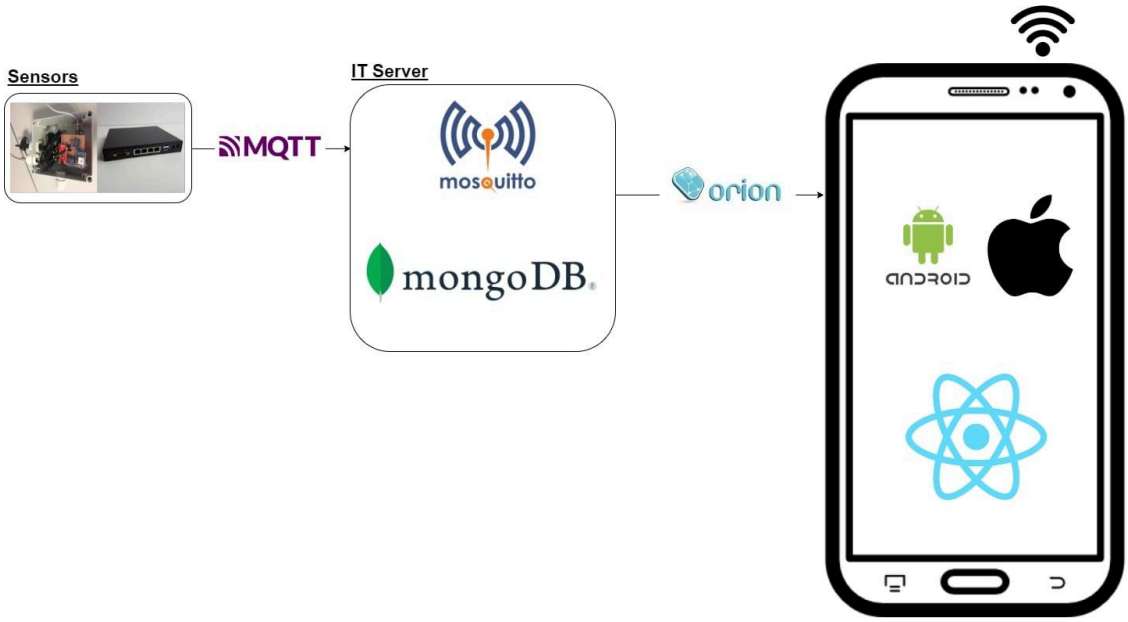


Figure 4.3: Service PECI's Physical Model

4.3 – Technological Model

Figure 4.4: Technological Model



The model represented in Figure 4.3, shows an overview of the technologies used within the system. The smartphone used by the user operates in Android or IOS and communicates to the IT Server using Orion to obtain the data calculated from the AI running in IT. The real-time data is retrieved using Orion, which permits the data to be received from outside of the IT network.

The data collected by the sensors is sent to the IT server using MQTT, this data is then processed by the MQTT broker(mosquito) and saved in MongoDB.

5 Implementation

5.1 Communication with the app and Main IT server

5.1.1 Introduction

The server service is written in Python3 and is constantly running on a virtual machine in IT that is used as our personal service. It is responsible for processing and accessing the data provided by the IT services. The server is also responsible for returning the data that will be handled by the Line Detection and Bus ETA Detection.

5.1.2 Detection and Data filter

For the Detection and Data filter, firstly we get the data from the MQTT and ORION services provided by IT. These services are used in order to get the data needed for the Line Detection and Bus ETA calculations.

MQTT Service path	Resources
/apu/cam	Information related to the detection of buses

ORION Service path	Resources
aveiro_cam	Information received from the SmartLampPosts
aveiro_radar	Information about the radars installed
transdev	Information about the buses OBU's

Table 5.1.1: Summary of topics used for data collection

5.1.2.1 ORION historical data

In our code we subscribe to the ORION topic of transdev because its historical data will provide us with the ID of the bus and its location within a predetermined time window. This is done by creating a URL using the start time and end time of our time-window, the type of data that we are subscribing to and the attributes needed, as represented by the example below.

```
start_time = [integer]

end_time= [integer]

type =
aquatic|noiselevelobserved|vehicle|airqualityobserved|weatherobserved|obugp
s

attribute =
location|speed|obuid|rsuid|dateobusent|datersusent|heading|drivingbehaviour
|drivingbehaviouroutside|acceleration|road

Example:"https://api.atcll-data.nap.av.it.pt/history?type=obugps&start=1656
085025000&end=1656088625000&attribute=location"
```

In the example before, there's the URL related to the buses detected, on the 24th of June between 16:37 and 17:37 (1656085025000 - 1656088625000), that will give us the data with each bus and its location on specific time-stamps demonstrated below.


```
{
   "urn:ngsi-ld:obuGPS:transdev:87":{
      "time_index":[
      "2022-06-24T15:55:58",
      "2022-06-24T15:55:59",
      "2022-06-24T15:56:00",
      "2022-06-24T15:56:01",
      ...
      ],
      "long":[-8.651342533,-8.6513817,-8.65138545,-8.6513962,...],
      "lat":[40.640020583,40.640052167,40.640086083,40.640106017,...]
   },
   "urn:ngsi-ld:obuGPS:transdev:51":{
```

```
      ...
   },
   ...
}
```

5.1.2.2 MQTT Data

In order to detect the buses as soon as possible, we resorted to the MQTT broker provided by the IT infrastructure. With this, our MQTT client can subscribe to the topics with real-time information to be treated and filtered.

For instance, to detect all the buses when they first appear in range of the lamp-posts, a filter was created in order to reduce the number of requests. This filter consists in a global dictionary that stores the buses when they first appear and removes them when the timestamp is outdated. The filter also checks if a bus is being detected by more than one lamp-post at the same time.

```
j_son = json.loads(msg.payload)
    if j_son["stationType"]==6: # Station Type (6) means it's a bus
        if j_son["stationID"] in y.keys(): #if it's in the dictionary, it
means it already past by a lamp-post
            if j_son["receiverID"] in y[j_son["stationID"]].keys(): #if the
bus already past by the lamp-post that is sending the message
            #Compare the timestamp of the bus in the dictionary and the
current timestamp detected by the lamp-post
                if j_son["timestamp"] >
y[j_son["stationID"]][j_son["receiverID"]]: # if the current timestamp is
bigger than the dictionary timestamp, that means the bus is still moving in
range of the lamp-post
                    y[j_son["stationID"]][j_son["receiverID"]] =
j_son["timestamp"] # update the timestamp in the dictionary

            else: #first time the bus is passing by that lamp-post
                IAjson=make_IA_request()
                                if IAjson != {}:
                  Line_detection(IAjson)
                  y[j_son["stationID"]][j_son["receiverID"]] =
j_son["timestamp"]
        else: # first time that bus is detected by any lamp-post
            IAjson=make_IA_request()
```

```
            if IAjson != {}:
                Line_detection(IAjson)
                y[j_son["stationID"]] = {}
                y[j_son["stationID"]][j_son["receiverID"]] =
    j_son["timestamp"]
```

5.1.3 App Communication

The communication with the app is being processed with two ORION topics, bus_line and aveiro_livetraffic. The bus_line topic is being used to send the data related to the Line Detection and Bus ETA Prediction which sends the ID of the bus, the timestamp and the busData. Meanwhile, the aveiro_livetraffic topic is sending the ID of the radar, the timestamp and the radarData, which sends the road lane, color, average speed and lastly the average count of vehicles detected.

ORION bus_line topic

```
{
  "busData": {
      "metadata": {},
      "type": "dict",
      "value": {
      "bus_id": "52",
      "day": "11/06/2022",
      "line": 1,
      "paragem": "2834104313",
      "prediction": {
      "1482460601": "14:48:26",
      "1721651331": "14:55:39",
      "1799473677": "14:40:47",
      "3414738003": "14:51:01",
      "4852045623": "14:43:22",
      "4852088188": "14:44:09",
      "4873436913": "15:10:37",
      "4873464186": "15:06:34",
      "5174144701": "14:49:34",
      "5396335661": "14:59:34"
      },
      "time": "14:34:50"
      }
  },
```

ORION aveiro_livetraffic topic

```
{
  "dateSent":{
     "metadata":{

     },
     "type":"Datetime",
     "value":"2022-06-11T15:47:52.000Z"
  },
  "id":"p33",
  "radardata":{
     "metadata":{

     },
     "type":"dict",
     "value":{
        "average_count":1,
        "":25.643076923,
        "cor":"#d6ff00",
        "faixa":"1"
     }
  },
  "type":"Radar"
}
```

<pre>"dateSent": {
"metadata": {},
"type": "DateTime",
"value": "2022-06-11T14:22:38.000Z"
},
"id": "urn:ngsi-ld: Bus:peci_bus:52",
"type": "Bus",
"dateObserved": {
"metadata": {},
"type": "DateTime",
"value": "2022-05-30T20:52:30.000Z"
}
}</pre>	

Table 5.1.2: Examples of each ORION topic

5.2 Line Detection and Bus ETA Prediction

5.2.1 Introduction

Line detection and Bus ETA Prediction are written in Python3 and are constantly running on a server in IT. It is responsible for assigning a line to a bus and calculating an Estimated Time of Arrival (ETA) to the next stops of the bus using only historical positional data stored in a database. It is also responsible for storing the output in a Mongo Database.

5.2.2 Configuration

In this section, it is shown how to run this module.

To run the docker module you type the command: docker-compose up

This command starts the 3 docker containers that receive, process and send the needed data, for the mobile app to consume, to Orion.

The 3 docker containers are:

- mongo - docker containing four mongo databases.
    - Bus_lines : contains the assigned line to each bus and deletes the old data.
    - MapBoxTimeStampsPrediction : contains the assigned line to each bus but doesn't delete the old data
    - LiveTraffic : contains the average of the previous day for each radar
    - LiveTrafficSave : contains the average of each day for each radar
- app-lines - docker container that includes the line detection algorithm.
- live-traffic - docker container that includes the live-traffic calculation.

5.2.3 Method used

To make an algorithm that could work it needs various data about the lines, so several Json files were created. First, a file that contains all stops in all lines was created, to do this an intersection of the stops that AveiroBus uses with a dataset available for us was made, this process was manual and double-checked. Then starting from that file, other Json files were created with other info such as stops per line and lines per stop.

When a bus is detected in real-time, this module is automatically called and receives the info about the history in the last hour of the bus as a JSON object (the JSON object contains coordinates stored by the *OBUs* in the bus) and the id of the bus detected. This data runs through a filter function that tries to find any stop that would indicate the bus changed lines. If such a stop is found the data chronologically before the stop is discarded.

Then, for each coordinate in the Json object, the coordinate is passed to a function that determines the nearest stop to the coordinates and returns its ID. For this, we developed the *Distance* module that uses the *geopy* library to determine if that coordinate is near a stop. If the coordinates are near a stop it returns the distance to it, if not it returns *None*. The distance of the check can be easily adjusted.

In the next part we had to refactor the algorithm. At first the decision made was that after discovering the nearest stop, the algorithm would check if the stop only has one bus line going through

it, if that's the case then the algorithm would attribute that line to the bus and stop, else, the lines belonging to that stop would be intersected with the lines calculated before (if this is the first coordinate then it would be intersected with a list containing all lines). After processing all coordinates, the successive intersections would narrow the possible lines of that bus to 1. There was a problem in this approach as shown in the figure:

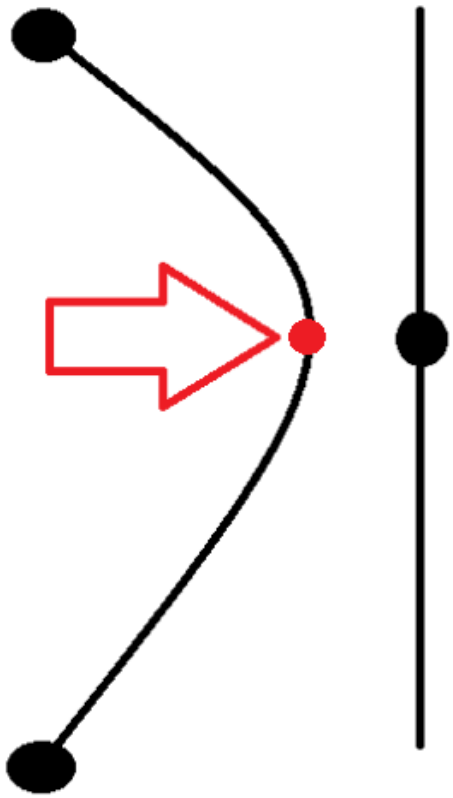


Figure 5.2.1 *-Ilustração do Problema de "Paragens Únicas"*

Suppose the coordinates of the bus are in the red dot and suppose the black dot in the straight line is a stop that contains only one line, using the approach described above the algorithm would determine the nearest stop is the one on the straight line and then a wrong line would be assigned to the bus.

To solve this issue and other issues created by the intersections the algorithm was refactored so that each line would have an internal counter to count how many stops belonging to it were detected. So

that when all the coordinates were processed, the line with the highest counter would be assigned to the bus.

The next task of this module is to determine the direction of the bus. For this, a function filters through the stops detected to try and find a stop that would indicate the direction of a bus:

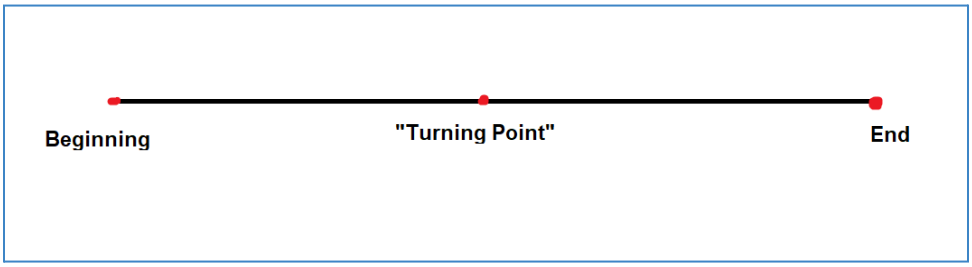


Figure 5.2.2 - Direction choosing Illustration

If a stop that is the beginning of a line, the end of a line, or the “turning point” of the line, is found then the direction is determined.

The next task of the module is to determine the ETA to the next stops of a bus, for this, a module called ETA_MapBox was developed. This module receives the current stop of a bus (where it is in real time), the line assigned to it, and the direction. As a failsafe, if the function responsible for assigning a direction fails, this module will count how many times the current stop appears in the line assigned to the bus, if the count is 1, the direction isn’t needed to know which stop is next. If this fails, the prediction is aborted. If all requirements are met, the module will know which stops are next in line for the bus as shown in the figure below:

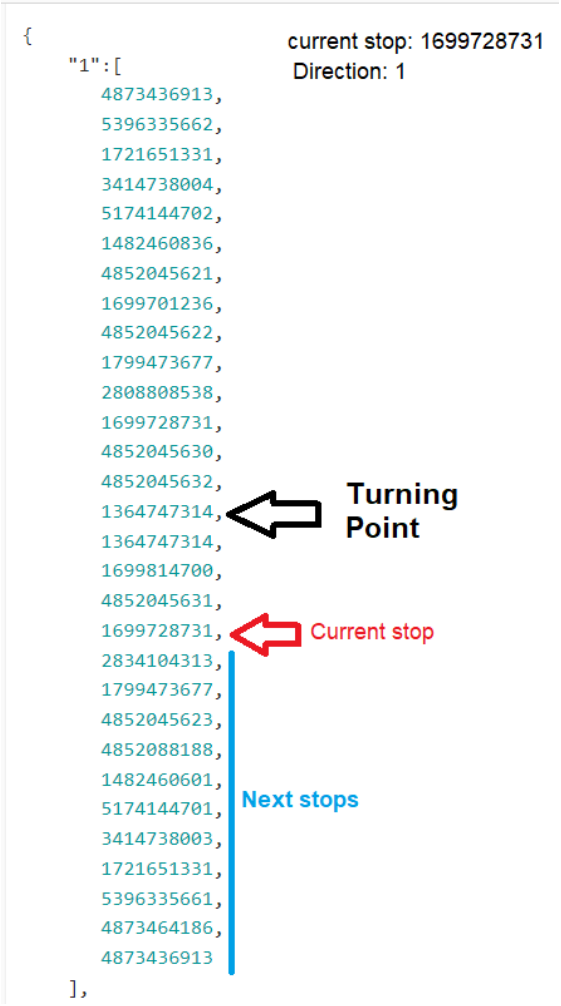


Figure 5.2.3 - *Current and next stop Explanation*

In this example, the current line is 1, the current stop is “1699728731” and the Direction is “1”, which means the module will search for the current stop after the Turning Point.

After determining the next stops, the module makes a request to the MapBox API with the coordinates of the next stops as waypoints in order, MapBox is a routing service like Google Maps. The response then is formatted and as a result, a JSON object is produced that contains the id of the stops paired with the Estimated Time of Arrival calculated by MapBox.

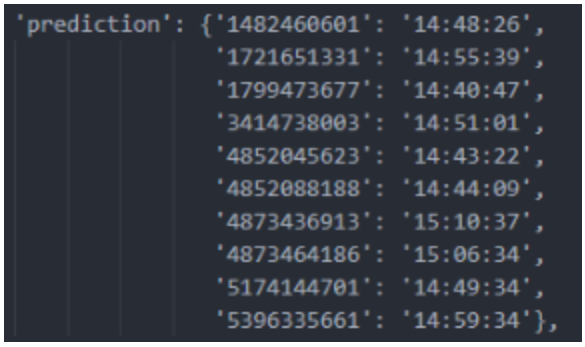


Figure 5.2.4 *-Example object produced by this module*

In the end, the data produced goes into two different mongo databases:

- Bus_lines: This database stores the line, day, timestamp,stop and next stop time predictions to the mongodb database. It is organized by the bus identifier and when new data comes to a certain bus, it deletes what was previously stored in a bus when it gets new data unlike the next database.
- MapBoxTimeStampsPrediction: This database stores the line, day, timestamp,stop and next stop time predictions to the mongodb database. It is organized by the bus identifier and when new data comes to a certain bus, it does not delete what was previously stored in a bus when it gets new data unlike the next database.

5.3 Live Traffic

5.3.1 Introduction

Live Traffic is written in Python3 and constantly runs on an IT server. It is responsible for processing radar information from the ATCLL Infrastructure in real-time and assigning each radar one color so this info can be drawn on the screen by the mobile app.

5.2.3 Method used

This module is constantly listening to new information from radars and when a message is received, it uses the information received to calculate the number of vehicles and the average speed detected by the radar. After this it compares the processed data with the average of the previous day so a color ranging in a green to red gradient can be assigned. This color then is published to Orion to be

retrieved by the app. The Average speed calculation and the vehicle count calculation is shown in the code snippet below:

```
average_speed = (j_son["speedHeavy"] + j_son["speedLight"])/2 if
(j_son["speedHeavy"] and j_son["speedLight"]) else (j_son["speedHeavy"] or
j_son["speedLight"])

average_speed = (average_speed) *(18/5)

vehicle_count = float(j_son["vehicleHeavy"] + j_son["vehicleLight"])
```

5.4 Mobile Application

5.4.1 Introduction

When developing the Mobile application, the main goal was to provide a platform in which the user could have access to information about the location of the Buses of the city of Aveiro, as well as its estimated time of arrival to the stops of the line that it's currently operating and the traffic of vehicles in roads where we have radar coverage. To do that, the data was first collected and then displayed in real-time.

5.4.2 Framework used

For the framework, we chose React Native, or more specifically Expo, which is a set of tools built on top of React Native, as this framework enabled us to produce a mobile app that would work not only in Android devices, as well as IOS devices with the same code.

We decided to use Expo specifically because this CLI gave us a lot of tools to make our development of the mobile app easier and quicker. Some of these advantages were: No need to link pages, lots of libraries to choose from, better running to test the application with each line of code

introduced, and easier upgrades to new versions. Although it also had some disadvantages, such as: Some of the libraries we encountered for base React Native were not compatible with Expo, and others didn't support both Android and IOS at the same time, which limited some libraries we could use.

5.4.3 Scripts used

Because React Native is a open-source JavaScript framework open-source, it enabled us to make scripts to get all the data needed to show in the front end from the server hosted in IT, as well as have some of the data about the lines and stops in functions so we are able to get them when needed.

For data we have the 3 different javaScript files (LinesData.js , StopsData.js, RoadsData.js).

These files each contain a function to return the respective data. In the LinesData we have all the coordinates, id and name of each stop for each line and upon receiving the requested line, the script returns the data of the stops included in that line. For StopsData, it returns all the stops in the city to show in the main screen of the app. Lastly, in the RoadsData function, it receives the radar, as an argument, that detected the traffic and the function returns the coordinates for the road detected by that radar.

Then we have the scripts that enable the communication with the IT server. These are subscribe_orion, subscribe_PECI, subscribe_traffic and lastly token_generate.

All these scripts create a request to a different URL of the Orion service currently hosted in the IT server, except the token_generate script, which performs a direct request to the API of ATCLL to receive a token to later make the Orion requests.

```
import axios from "axios";

export async function generate_token (){

            const   user_and_password={"username":   "[insert   your   username
here]","password": "[insert your password here]"}
```

```
    const res = await axios({
      method:'post',
      url: 'https://api.atcll-data.nap.av.it.pt/auth',
      headers:{
          'Content-type': 'application/json', 'Accept-Charset': 'UTF-8'
      },
      data:user_and_password
    })

  return res.headers.authorization;
```

Listing 5.1: token_generate script

Each subscribe script makes a request to a different Orion topic. For subscribe_orion, we receive the current positions of each detected bus in our infrastructure. For subscribe_PECI, we receive the calculated line through the Line Detection algorithm, for each bus when they enter the sensors range, as well as a prediction of arrival of each bus to the next stops of the line detected. Lastly for the subscribe_traffic, we receive the name of the radar, the road lane, the color that show be depicted in the road lane and the average speed and number of vehicles at that given time.

For an example of a subscribe script, here is the subscribe_orion script:

```
export default function sub(token){

    const axios = require('axios')

    const services = [
        {
            service: 'aveiro_cam',
            types: [
                {
```

```
                    type: 'Values',
                    attrs: ['speed', 'location', 'dateObserved','stationID'],
                    timeIndex: 'dateObservedFrom'
                },
            ]
        },
    ]

    let myPromise = new Promise(function(myResolve) {

        services.forEach(({service, types}) => {
            types.forEach(async ({type, attrs, timeIndex}) => {
                const res = await axios({
                    method: 'GET',
                                                                    url:
'https://orion.atcll-data.nap.av.it.pt/v2/entities?entityId=urn:ngsi-ld:Values:
aveiro_cam:42&type=Values&attrs=stationID,location,speed,dateObserved',
                    headers: {
                        "FIWARE-Service": service,
                        "authorization": token
                    }

                })

                myResolve(res);
            })
        })
    });
    return myPromise;
}
```

Listing 5.2: subscribe_orion script

All the subscribe scripts had to be implemented with the use of promises to ensure that after the request was made, the app would only read the result of the request after the data arrived and not

before that, otherwise the app would have multiple errors when trying to read the requested data that didn't arrive yet.

5.4.4 App Screens

Main Screen

On the main page of the app (named Map Screen), it's displayed a map containing three types of markers, one for the buses circulating, one for the stops of the city, and another one to represent the current location of the user, if the user gives permission for the use of GPS location, which was possible through the expo-location library. The map was implemented with the intent of making it more intuitive for the user to know the current location of the buses to navigate easier between them. It was implemented with the help of the MapView component from the react-native-maps library, which enabled us to use the google maps map for the android version and Apple Maps in the IOS version.

The location of the buses is received through the subscribe_orion script and when a bus marker is clicked, a menu appears in the lower part of the screen, showing the line in circulation by the bus, it's current speed, estimated time of arrival to the next stop, and the name of the next stop. This information is received through the subscribe_PECI script.

To illustrate how we used the data received, the next code shows how we create the markers in the app using bus data:

```
sub(this.state.token).then(
                res => this.setState( {buses:filter_orion(res)} )
            );

                function filter_orion(value) {
                    var data=[];

                    for (let i=0;i<value.data.length; i++){
```

```
                        var obj={
                            id:value.data[i].stationID.value,
                            lat:value.data[i].location.value.coordinates[1],
                            lon:value.data[i].location.value.coordinates[0],
                            speed:Math.round(value.data[i].speed.value*18/5),
                            linha: [0],
                            eta: 0,
                            next_stop: "NaN"
                        }

                                                                var   timeOfData=
Date.parse(value.data[i].dateObserved.value)
                        var now=Date.now()

                           var timeToDeleteBus = 5*1000; //! 5 sec para deletar
bus

                        if ( timeOfData > (now- timeToDeleteBus) ){
                            data.push(obj);
                        }

                    }

                return data;
            }
```

Listing 5.3: Map Screen code snippet

The bottom-sheet menu was made as a separate component (named BusInfoMenu) and was implemented with the help of the RBSheet component from the react-native-raw-bottom-sheet library.

To reduce lag due to the excessive amount of requests being made to Orion, the calculated data for each bus (data like the line and ETA calculated in our server) is requested in this component.

The next code snippet illustrates how we use the calculated data in the BusInfoMenu component:

```
sub_PECI(this.props.token).then(
                res => {this.setState(filter_peci(res));}
            )

            function filter_peci(value){
```

```
                const d = new Date();

                var item = {
                  linha:"Nan",
                  eta:"Nan",
                  next_stop:"Nan"
                };

                for (let i=0;i<value.data.length; i++){
                    if (value.data[i].busData.value.bus_id == newID){

                      var now=Date.now()
                      var min=999999999
                      var tmp_obj={
                          stop: "Nan",
                          eta: 0
                      }


Object.entries(value.data[i].busData.value.prediction).forEach(([key,  value2]) => {
                          var arr_stop= value2.split(":");

                                                                  var  date_str= d.getMonth()+1+"/"+d.getDate()+"/"+d.getFullYear()
                          var tmp=Date.parse(date_str)
                                                                var  timeOfStop= tmp+(parseInt(arr_stop[0])*3600+parseInt(arr_stop[1])*60+parseInt(arr_stop[2]))*1000

                          var tmp=timeOfStop-now
                          if (tmp>0 && tmp<min){
                              min=tmp
                              tmp_obj.stop=key
                              tmp_obj.eta=Math.round(tmp/1000/60)
                          }

                      })

                       var stop_idx = StopsData().findIndex(x => parseInt(x.id) === parseInt(tmp_obj.stop));

                      item = {
                        linha: value.data[i].busData.value.line,
                        eta: tmp_obj.eta,
```

```
                        next_stop: StopsData()[stop_idx].name
                      };


                }
            }

            return item;
        }
```

Listing 5.4: Bus Info Menu code snippet

Lines Screen

On this screen, we can consult the several lines that aveiro bus provides, for each line its displayed a map with all the stops, buses in the corresponding line and also the user location. It also shows the prediction of the arrival time of all the stops of the line. This map was implemented with the help of the MapView component from the react-native-maps library as explained above.

The location of the bus is given by the orion service using the subscribe_orion script.

```
sub(this.state.token).then(
                res => this.setState( {buses:filter_orion(res)} )
                );

                function filter_orion(value) {
                    var data=[];

                    for (let i=0;i<value.data.length; i++){
                        var obj={
                            id:value.data[i].stationID.value,
                            lat:value.data[i].location.value.coordinates[1],
                            lon:value.data[i].location.value.coordinates[0],
                            speed:value.data[i].speed.value,
                            linha: [0],
                            eta: 0,
                            next_stop: "NaN"
                        }

                        var timeOfData=
Date.parse(value.data[i].dateObserved.value)
                        var now=Date.now()

                        var timeToDeleteBus = 5*1000; //! 60 sec para deletar
```

```
bus

                        if ( timeOfData > (now- timeToDeleteBus) ){
                            data.push(obj);
                        }

                    }
                return data;
            }
```

Listing 5.5: Lines code snippet

To reduce lag due to the excessive amount of requests being made to Orion, the calculated data for each line (data like the line and ETA calculated in our server) now is made in the LinesTable component and the request it's only made once when the line is selected.

The information of the stop's estimated arrival time of the stops and line is given by the Orion service using subscribe_PECI script.

```
sub_PECI(this.props.token).then(
        res => {this.setState({tableData: filter_peci(res)});}
      )

      function filter_peci(value){

        var date_sent=0;
        var items = [[null,"No bus in this line"]];

        for (let i=0;i<value.data.length; i++){
          if(value.data[i].dateObserved != null){continue}

          if (value.data[i].busData.value.line == line &&
Date.parse(value.data[i].dateSent.value) > date_sent ){

            date_sent = Date.parse(value.data[i].dateSent.value);
            var dict=value.data[i].busData.value.prediction

            if (Object.keys(dict).length === 0){
              return [[null,"No prediction available"]]
            }

            items = Object.keys(dict).map(
```

```
              (key) => {
                var stop_idx = StopsData().findIndex(x => parseInt(x.id) ===
parseInt(key));
                return [dict[key], StopsData()[stop_idx].name]
              });

            items.sort();

          }

        }
        return items
      }
```

Listing 5.6: Lines Table code snippet

Bus Schedule

On this screen, we can view all 13 schedules for each line currently active in the AveiroBus system. To implement this we first thought to use the react-native-pdf library, but unfortunately this library was not compatible with the Expo cli we are using. So we decided to use the links for the schedules and create iframes and using javascript code so the iframe was purely the PDF and did not have any links to other sites or redirections.

Live traffic

The objective for this screen is to show the roads of the city of Aveiro. The roads that are currently being monitored are the ones which have one of the radars from the ATCLL infrastructure nearby. At the time of writing we have 5 roads being monitored.

To draw the roads in the map, we used the Polyline component present in the react-native-maps library.

The next code snippet exemplifies how we create the Polylines in the map:

```
<MapView
                ref={(map) => { this.map = map; }}
```

```
                style={styles.map}
                customMapStyle={[
                    {
                      featureType: "administrative",
                      elementType: "geometry",
                      stylers: [
                      {
                          visibility: "off"
                      }
                      ]
                    },
                    {
                      featureType: "poi",
                      stylers: [
                        {
                          visibility: "off"
                        }
                      ]
                    },
                    {
                      featureType: "road",
                      elementType: "labels.icon",
                      stylers: [
                        {
                          visibility: "off"
                        }
                      ]
                    },
                    {
                      featureType: "transit",
                      stylers: [
                        {
                          visibility: "off"
                        }
                      ]
                    }
                  ]}
                >
                <Polyline
                    coordinates={RoadsData("p1_1")}
                    strokeColor={this.state.dict.p1_1.color}
                    strokeWidth={5} //->pontos | strokeWeight={5} //->linha
                    lineDashPattern={[0]}
                    tappable={true}
                    onPress={() => {
```

```
                    this.RBSheet.open()
                    this.setState({
                        road_clicked_id: 11
                    })
                    }
                }
            />
```

Listing 5.7: Live Traffic code snippet

Having the roads depicted in the map, it's possible to click on each transit route to see its name, average speed and average number of vehicles. This information, as well as the color depicted in the road, which symbolizes how much transit the road has at a given time, is received through the subscribe_traffic script previously mentioned, that collects the data from the radars which was processed by our server in IT. The colors follow a gradient, from pure green to pure red, the former meaning no traffic and the latter meaning the road has a lot of traffic compared to the usual.

6 - Results

In the end of our project we achieved all the goals that were set in the beginning of the project.

Now we will show the final result (the app that was made to show the data produced) divided between the two objectives.

6.1 Bus tracking and Arrival Prediction

For the first objective, we decided to have two screens, from which the implementation was described before. The first screen in the app contains a map that shows all the stops and the buses being currently detected by the infrastructure's sensors. When clicking on a bus it shows the received/calculated information as follows:

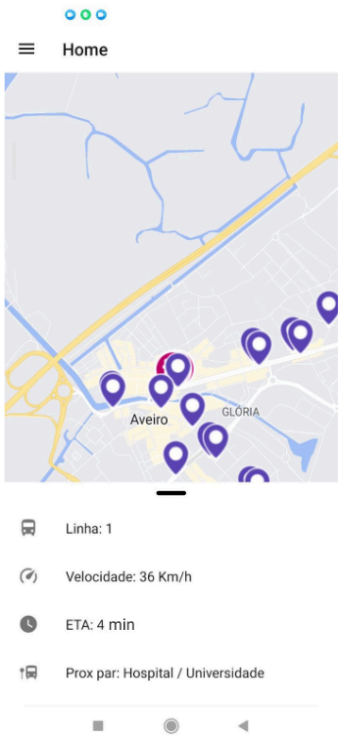


Figure 6.1:Map Screen with Bus information

To show a more detailed and complete set of data for the prediction made for a bus we use the Lines screen. In this screen it's possible to select the line desired to see it's stops, buses that are currently

doing that line as well as a list of the next stops the bus is going to go through and the expected time of arrival calculated before:

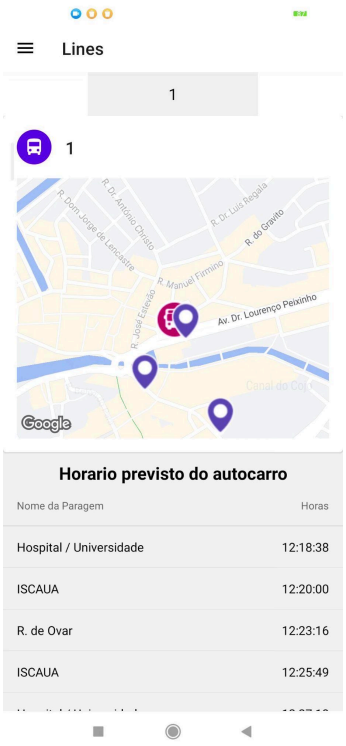


Figure 6.2:Lines Screen with line 1 selected

6.2 Live Traffic

For the second objective we have the live traffic screen, from which the implementation was described before. In this screen it’s shown the traffic in the streets where we have radars installed. These radars provide information about the average vehicle's speed, way and the global count of vehicles in that street.

In the app we receive average speed and vehicle count from the radars through IT servers and receive the color of the street that is calculated by our service running in IT.

The name of the street is in the app files, so there is no need to be sent by the servers.

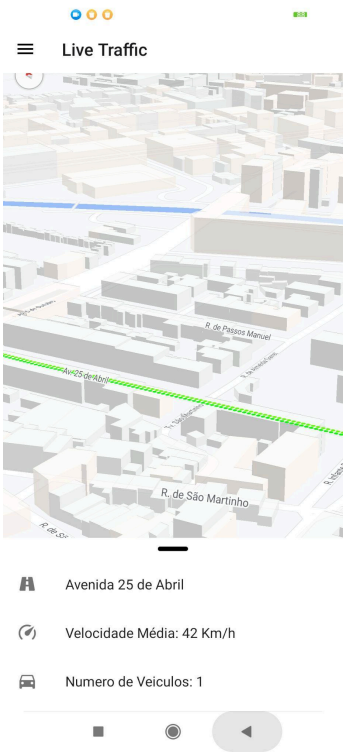


Figure 6.3:Live Traffic with Street information

6.3 Bus Schedule

As a small extra, we decided to have a screen in our app to show the schedules of the bus lines from the AveiroBus website.

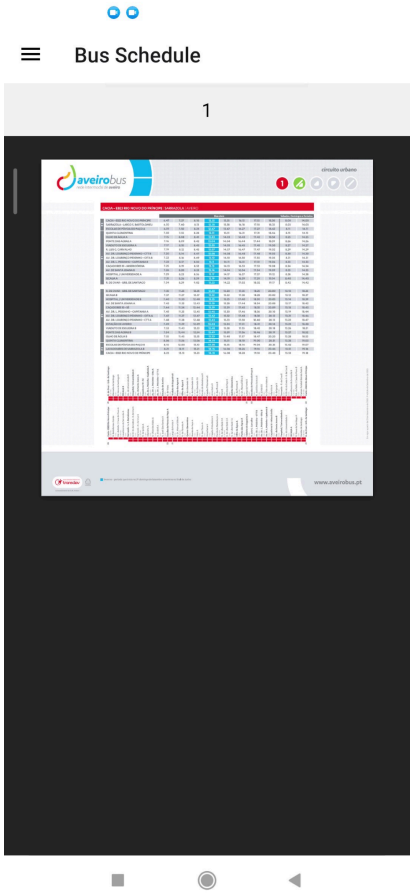


Figure 6.4:Bus Schedule with line 1 information

7 - Conclusion

Our project, Aveiro Tech City Living Lab (ATCLL) Mobile, has as its main goal to enrich the city of Aveiro in order to improve the daily life of our citizens. This way, we thought about two day-to-day situations that could be improved. First, we took into account the citizens who use public transportation and use the static bus timetables available. The problem is that bus schedules, being static, do not take into account delays. So we wanted to provide a service to citizens where they could have access to better timetable results by predicting the arrival of buses. In addition, we also thought of creating a service that would allow the citizens of Aveiro to be warned about the busiest roads, so that they could avoid those roads.

Both services were developed, the first functionality has the official name of bus tracking and arrival prediction and the second functionality has the name of Live Traffic. It should be noted that for the development of the functionalities we used data provided by the ATCLL infrastructure.

Our product can be used through an Android/IOS smartphone by any citizen of the city of Aveiro that uses buses as the main means of transport or wants to know the current traffic in the city.

For future work several features can be developed and implemented in our project. Firstly, line detection will become an API provided by Transdev and for that detection we would have more active smart post lamps, which would improve the existing prediction and we could also improve the prediction algorithm.

In addition, the live traffic feature will also be improved by installing more radars, but unlike the other features, this will need an application update to add the coordinates for the new monitored roads.

References

1 Using mobility data to estimate bus arrival time in a smart city, Ana Filipa Ferreira Tavares, UA-DETI em 2019

2 https://aveiro-open-lab.pt/ consulted on 03/01/2022

3 https://www.aveirobus.pt/ consulted on 04/01/2022

4 City Manager, dissertation of André Mourato, UA-DETI em 2021