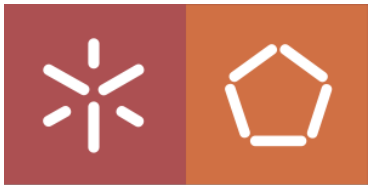


Universidade do Minho
Escola de Engenharia

SLiPaD - Smart Lighting with Parking Detection

Master in Industrial Eletronics and Computers Engineering

Embedded Systems

Authors:
Diogo Fernandes PG47150
José Tomás Abreu PG47386

Supervisors:
Prof. Dr. Tiago Gomes
Prof. Ricardo Roriz
Prof. Sérgio Pereira

January 1, 2022

Contents

List of Figures

List of Tables

Acronyms

AC Alternating Current

ACK Acknowledge

API Application Programming Interface

CAGR Compound Annual Growth Rate

CPS Cyber-Physical System

CRC Cyclic Redundancy Check

CSI Camera Serial Interface

CSS Chirp Spread Spectrum

DB Database

DC Direct Current

FSK Frequency-Shift Keying

GPIO General Purpose Input/ Output

GPS Global Positioning System

GUI Graphical User Interface

HTTP HyperText Transfer Protocol

I2C Inter-Integrated Circuit

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

IIO Industrial Input/Output

IoT Internet of Things

IP Internet Protocol

IP Ingress Protection

ISM Industrial Scientific and Medical

LED Light-Emitting Diode

LPWA Low Power Wide Area

LPWAN Low Power Wide Area Network

LTE-M Long Term Evolution for Machines

LVDS Low Voltage Differential Signaling

MIPI Mobile Industry Processor Interface

MISO Master In Slave Out

MOSI Master Out Slave In

NACK No-Acknowledge

NB-IoT Narrow-Band IoT

PGID Process Group Identifier

PID Process Identifier

PIR Passive Infrared

PWM Pulse Width Modulation

RF Radio Frequency

SCL Serial Clock

SDA Serial Data

SID Session Identifier

SPI Serial Peripheral Interface

TCP Transmission Control Protocol

UDP User Datagram Protocol

UNB Ultra-Narrow Band

XML eXtensible Markup Language

Chapter 1

Introduction

1.1 Problem Statement

Nowadays, the energy crisis is a constant theme because of the inflated energy prices [2]. Furthermore, huge energy consumption is a burden to the environment, as not all means of energy production are non-polluting. According to "Our World in Data"[3], in 2019, 63,3 % of eletrical energy production comes from fossil fuels. It is known that generally, street lamps are continuously switched on at night, most of the time unnecessarily glowing with its full intensity, in the absence of any activities in the street, leading to a great waste of energy. Furthermore, it is in cities where the consequences of using cars are most noticeable. An example of this is the search for a parking space. According to the RAC Foundation [4], in England, an average car is parked 95 % of the time, which explains how hard it can get sometimes when trying to find a parking spot. This struggle leads to an increase in carbon dioxide production as well as fuel and energy consumption.

With that in mind, this project aims the implementation of applications for a Smart City, regarding Smart Lighting and Smart Parking, in order to decrease the energy consumption in public streets, while improving the lives of citizens around the world. The solution will embrace a centralized system, composed by smart street lights capable of turning on only when they detect movement in the surroundings, at night time, and also, capable of detecting available parking spaces in the street post vicinity.

1.2 Problem Statement Analysis

This solution provides a network of street lamp posts, each implementing Smart Street Lighting and Smart Parking Detection, using Raspberry Pi 4B [5] has a controller. A gateway is needed to gather all the information from the street lamp posts, and store that in a remote system, needed to provide a way for a responsible entity manage the network.

When there is no activity detected in the area, the lamp post is at a predefined minimum light level, whereas when a car or pedestrian is noticed in the area, the light automatically activates at full brightness. To allow to dynamically turn on the lights of the following poles, each the street lamp post communicates with the neighbor lamp posts, indirectly, through the gateway. To detect movement in the vicinity of the pole, a motion detector is used. Since the lamppost will only light up during the night time, the motion detector will also only work during that period. To ensure this, a luminosity sensor is used, determining the ambient light conditions. In order to facilitate the maintenance of the pole, a system that determines the operating conditions of the lamp is also implemented. When this system verifies that the lamp is not in good working conditions, in other words, that it is broken or burnt, this information is transmitted to the entity responsible for the network of lamp posts, through a mobile app. This is also used by the person in charge, to manage all information on the pole network, such as the location and working conditions of each pole.

In order to detect empty parking spots, this system should only be used in an area where there are parking spaces nearby. For this, the lamp post has a camera, turned on all day, and, after Raspberry Pi processes the acquired information, it will be available on a website, so that a user, a car driver, can know where there are empty parking spaces.

Chapter 2

Market Research

2.1 Market Definition

As figure 2.1 shows, there are various applications of IoT technology for smart cities. In this project it will be created a solution that comprises Smart Lighting management and Smart Parking.

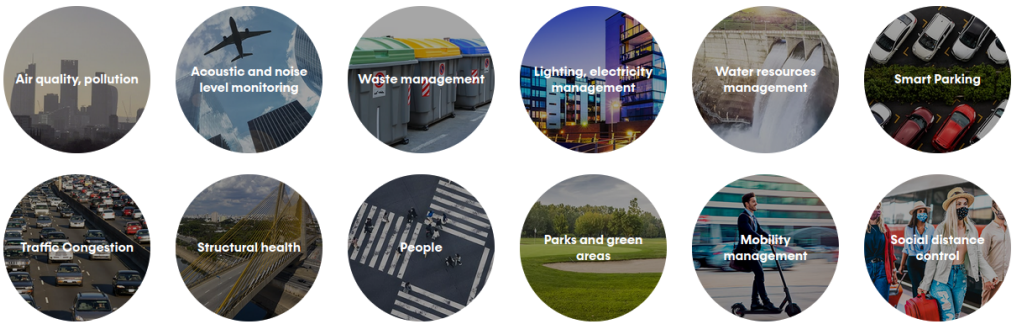


Figure 2.1: Applications of IoT technology for Smart Cities. [1]

The IoT starts with connectivity, but since it is a widely diverse and multifaceted realm, one certainly cannot find a one-size-fits-all communication solution. Next, one can identify these common types of IoT wireless technologies, through mesh and star topologies:

- **LoRaWAN** (LoRa from "long range") is a Low Power Wide Area Network (LPWAN) specification that targets key requirements of IoT, such

as secure bi-directional communication. LoRaWAN network architecture is deployed in a star-of-stars topology in which gateways relay messages between end-devices and a central network server. The gateways are connected to the network server via standard IP connections and act as a transparent bridge, simply converting Radio Frequency (RF) packets to IP packets and vice versa. The wireless communication takes advantage of the Long Range characteristics of the LoRa physical layer, allowing a single-hop link between the end-device and one or many gateways. [6]

- **Wi-SUN** (Institute of Electrical and Electronics Engineers (IEEE) standard 802.15.4g) is a RF mesh communication technology, which enables large-scale outdoor IoT networks including applications such as asset management, environmental monitoring, agriculture, structural health monitoring and much more. [7] Using the same IEEE standard, there is also **ZigBee**, a short-range, low-power, commonly deployed in mesh topology to extend coverage by relaying sensor data over multiple sensor nodes. [8]

- **Sigfox** is a cellular style communication technology that provides low power, low data rate and low communication costs for IoT applications. Sigfox employs Ultra-Narrow Band (UNB) technology, which enables very low transmitter power levels to be used while still being able to maintain a robust data connection, using unlicensed Industrial Scientific and Medical (ISM) radio bands. The simple and easy to roll-out star-based cell infrastructure has encouraged its current extended worldwide availability. [9]

- **Narrow-Band IoT (NB-IoT)** is a carrier-grade RF, narrowband communication technology, specially designed for the IoT. It connects devices more simply and effciently on already established mobile networks, and handles small amounts of infrequent 2-way data, securely and reliably. The special focus of this standard is on very low power consumption, excellent penetration coverage and lower component costs, deployed in GSM and LTE regulated frequencies. [10]

- **Long Term Evolution for Machines (LTE-M)** is a Low Power Wide Area (LPWA) technology standard published by 3GPP. It supports IoT through lower device complexity and extended coverage,

while allowing the reuse of the LTE installed base. Supported by all major mobile equipment, chipset and module manufacturers, LTE-M networks will co-exist with 2G, 3G, and 4G mobile networks and benefit from all the security and privacy features of carrier-grade networks. [11]

2.1.1 Smart Lighting

Smart Street lighting is a rapidly growing lighting market, with an expected Compound Annual Growth Rate (CAGR) of 20.4 % until 2026 [12], implementing a smart management of public lighting to optimize energy consumption according to lighting needs. This is boosted by regulatory policies that encourage energy efficiency, IoT convergence and the drop of Light-Emitting Diode (LED) prices. This new concept of smart light post is also growing, implementing not only the smart management of street lights, but also features that go from basic LED replacement control, to traffic and video monitoring, environmental monitoring, and others.

2.1.1.1 FLASHNET - inteliLIGHT

FLASHNET is a company focused on developing intelligent systems for smarter cities and better infrastructures and have created a solution that provides the right amount of light where and when needed to lighten the streets, the inteliLIGHT. [13] Using the existing infrastructure, this solution saves money and transforms the existing distribution level network into an intelligent infrastructure of the future.

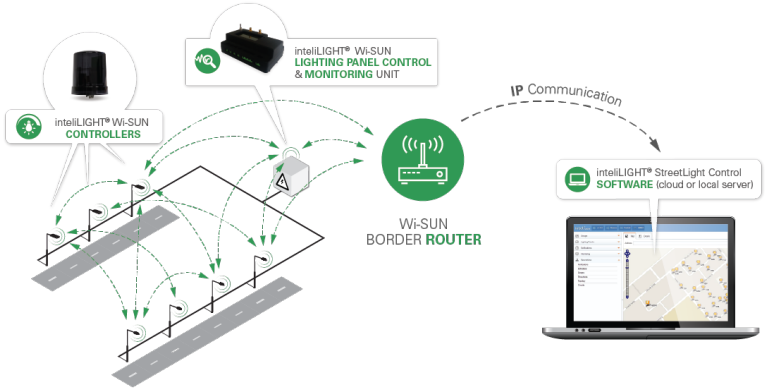


Figure 2.2: inteliLIGHT Communication Technology.

In figure 2.2 it is presented one of the many communication technologies that inteliLIGHT can provide in their smart street light solution. In this case, it is shown the use of Wi-SUN, a RF mesh street lighting communication technology. Furthermore, the system is integrated with major IoT platforms and provides Application Programming Interface (API) connectivity with City Management applications, ensuring compatibility with existing smart lighting and smart city initiatives.

2.1.1.2 Telensa - PLANet

Nowadays, Telensa is the market share leader in smart street lighting with more than ten years of experience.[14] PLANet is connected street lighting system that consists of wireless control nodes, an UNB wireless network and a Central Management System, as seen in figure 2.3. This system reduces energy and maintenance costs associated with street lighting and also improves quality of maintenance through automatic fault reporting.

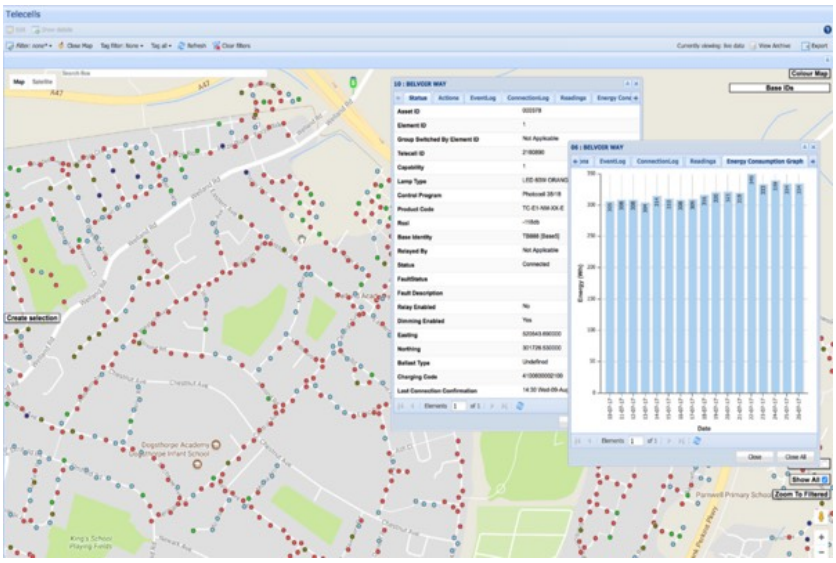


Figure 2.3: Telecells - PLANet's Central Management System.

2.1.2 Smart Parking

Smart parking, through the monitoring of parking spaces availability in the city, is also a growing market, expected to grow with a CAGR of 17.85% in the forecast period of 2021 to 2028.[15] The rise in investment in building driverless vehicles and an increase in the government's initiative in building smart cities across the globe, along with the demand and adoption of IoT technology, are the main driving factors for the growth of smart parking market.

2.1.2.1 intuVision - intuVision VA Parking

Regarding only to the detection of available parking spaces, there is a solution, by intuVision, named intuVision VA Parking, which provides parking lot analytics to determine vehicle count and security, and monitor parking space availability at all times, both for cities and for private parking lots, as one can see in the figure 2.4.[16]

Figure 2.4: intuVision Parking Lot Demonstration.

2.2 Why choose our product

This product aims to decrease power consumption associated with the traditional street light network, and also, using that infrastructure, contribute to the development of a smart city, detecting available parking spaces in the streets. This street lighting solution can be used in residential areas, public spaces or a large outdoor parking lot, feasible of being installed in existent lamp posts, requiring minimum changes to the original infrastructure. Although in this project it is not implemented, aside the parking spaces availability detection, this product can have the ability to monitor and to process various areas of interest using the camera built in, like for example, security purposes.

Chapter 3

System

3.1 System Requirements and Constraints

In order for the system to have the desired performance, these requirements and constraints must be respected:

3.1.1 Functional Requirements

- Sensors data acquisition;
- Motion detection;
- Control of a street lamp;
- Control a network of street poles;
- Wireless communication between local systems and gateway;
- Manage street poles network information through a mobile application;
- Empty parking spots detection;
- Add lamppost location through a mobile application;
- Access available parking spots location through a web site.

3.1.2 Non-Functional Requirements

- User friendly mobile application and web site;
- Ambient luminosity sensing;
- Lower power consumption than actual street lights;
- Soft Real-Time Embedded System.

3.1.3 Technical Constraints

- Buildroot
- C and C++
- Device Drivers
- Linux
- Raspberry Pi
- Cyber-Physical System (CPS)
- Makefiles
- Pthreads

3.1.4 Non-Technical Constraints

- Two members team
- Project deadline at the end of the semester
- Low budget

3.2 Network Architecture

To define the network architecture of the solution to be created, some aspects must be remembered. One is that there are various communication technologies that may be used, as presented previously in Market Research. Other important aspect to keep in mind is that this solution implements both Smart Street Lighting and Smart Parking, through the use of street lampposts. So, these must have parking spots nearby, in order to allow full use of the Smart Parking feature. In order to fulfill the city needs in street lighting, this lack of flexibility demands a creation of a similar solution, without the Smart Parking feature, that won't be approached in this project.

The data stream in the network will be very low since each lamppost will only communicate notifications on it's state. That is, if the lamp is light up, if it was detected a malfunction with the lamp, if it was detected an available parking space. Furthermore, since the street lampposts may be far apart from the gateway, there is the need to use a long range communication technology, to maximize the number of nodes connected to a single gateway.

In figure 3.1, one can see that LoRa is ideal for applications that transmit small chunks of data with low bit rates. Data can be transmitted at a longer range compared to technologies like Wi-Fi, Bluetooth, ZigBee or cellular communication technologies like Sigfox or LTE-M, as presented previously. These features make LoRa well suited for sensors and actuators that operate in low power mode. LoRa can be operated on the ISM license free sub-gigahertz bands, for example, 915 MHz, 868 MHz, and 433 MHz. It also can be operated on 2,4 GHz to achieve higher data rates compared to sub-gigahertz bands, at the cost of range. [17]

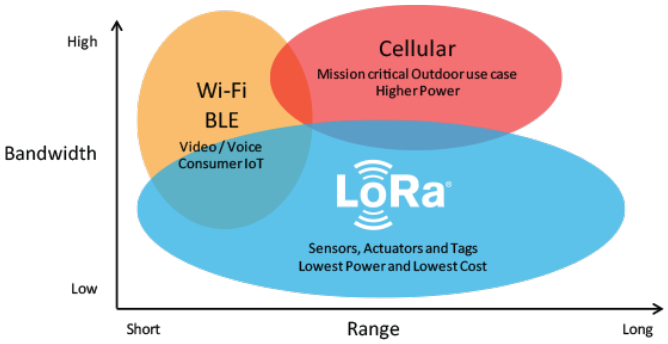


Figure 3.1: Communication technologies range vs bandwidth.

With that in mind, one can identify LoRa as a proper communication technology to use in this network.

In figure 3.2 one can see the network architecture diagram. This is a star topology, in which the gateway relay messages between each local system (lamppost) and a central network server, the remote system. This wireless communication takes advantage of the Long Range characteristics of the LoRa physical layer, allowing a single-hop link between the local system and the gateway. All communication modes are capable of bi-directional communication, and there is support for multicast addressing groups. The gateway is connected to the internet in order to store new information about the network in the remote system.

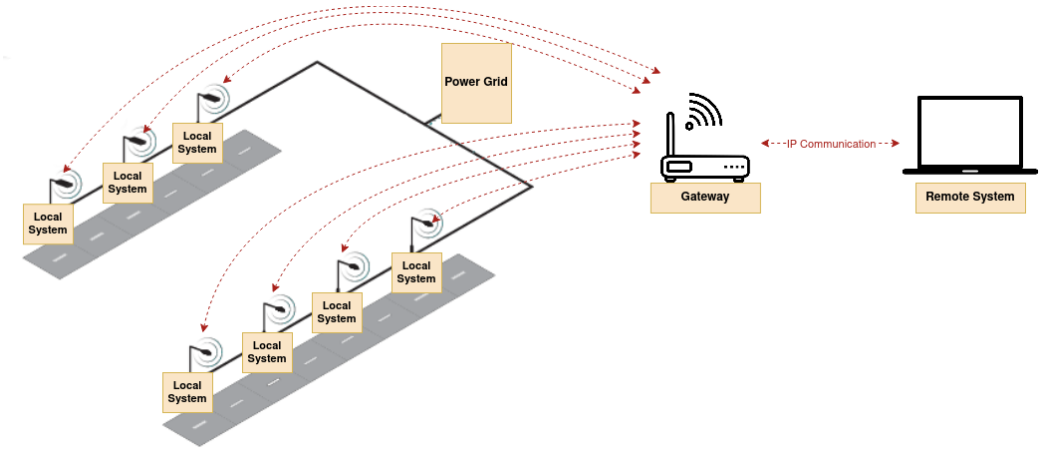


Figure 3.2: Network architecture.

LoRa end-devices serve different applications and have different requirements, that's why there are device classes, as one can see in figure 3.3.

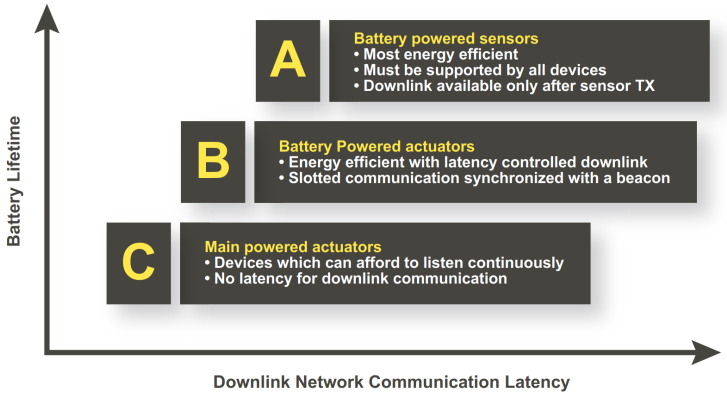


Figure 3.3: LoRa device classes.

Knowing that street lampposts have main power, from the power grid, one can classify the network nodes as class C. End-devices of class C are bi-directional end-devices with maximal receive slots, has they have almost continuously open receive windows, only closed when transmitting. That way, the downlink communication latency may be very low. [18]

LoRa provides long range communication, as LoRaWAN gateways can transmit and receive signals over a distance of more than 10 kilometers in rural areas and up to 3 kilometers in dense urban areas. It uses license free spectrum, so one doesn't have to pay expensive frequency spectrum license fees to deploy a LoRaWAN network. It is low cost, since it is a minimal infrastructure, low-cost end nodes and open source software.

To determine the maximum number of nodes that can be connected to a single gateway, one needs to evaluate gateway specifications, more specifically, the number of packets it can support. For instance, if there is a gateway supporting 1 million packets per day, and if the application sends 10 packet per hour, or 240 packets per day, then, more than 4000 nodes can be handled by that gateway. This is just a rough approximation, since it depends on the packet format/length, on the time needed for the node to send that packet, on the time needed for the gateway to process that packet, and many other constraints that at this point aren't fully identified.

3.3 System Overview

Through the system overview diagram, in figure 3.4, it is possible to identify the main modules of the system to be developed, and how they interact. We can divide the system into three subsystems: the local system, which represents a lamppost, the gateway, a device that links the lampposts network to the remote server, and the remote system, that stores information about the lamppost network and allows interaction with the system users by the use of remote client applications.

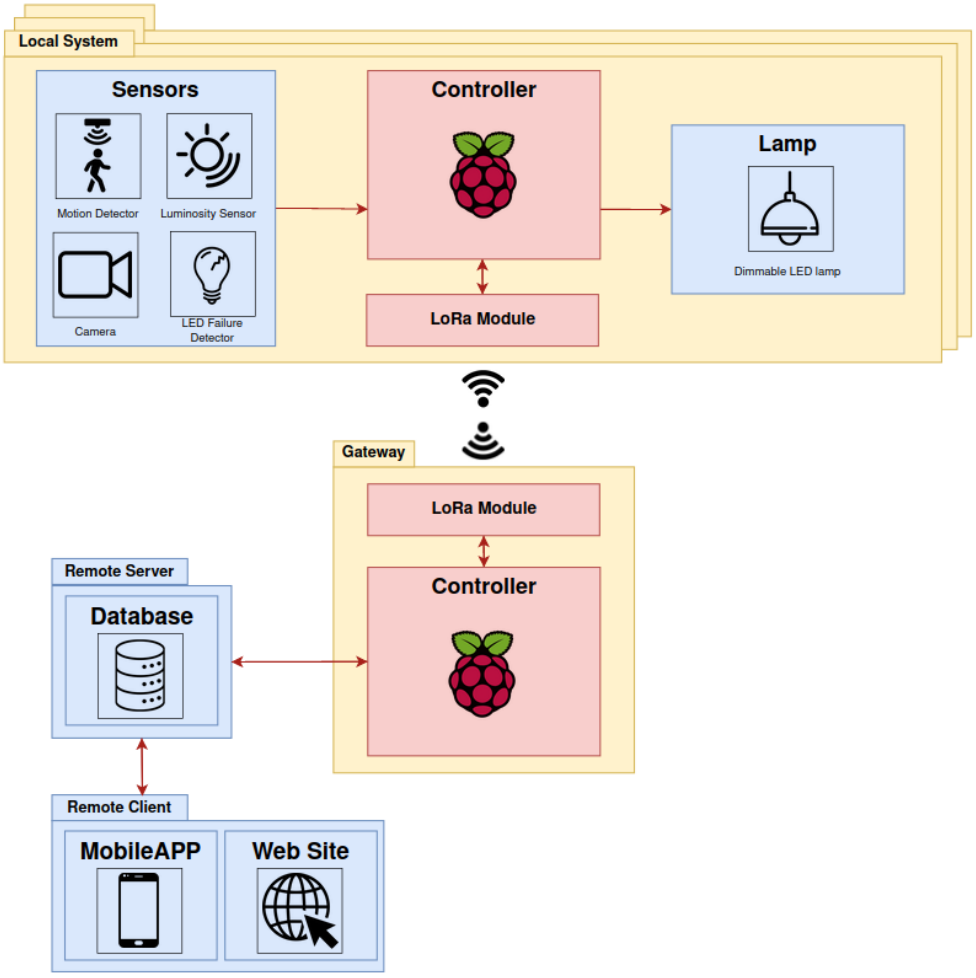


Figure 3.4: System Overview Diagram.

The local system is composed of sensors, a controller, a lamp and a wireless communication module, LoRa module. Regarding the sensors, there will be a motion detector, to allow the detection of movement in the vicinity of the pole, a luminosity sensor, to detect the light conditions of the pole's surroundings, a lamp failure detector to know if the lamp is working and a camera to detect empty parking spots in the lamppost vicinity. The controller of the local system is a Raspberry Pi, that uses the sensors information and communicates wirelessly with the gateway using a LoRa communication module. The gateway also communicates through internet with a remote server, so that network information is stored in the remote server, and to allow the dynamic control of local systems.

The remote system is composed by the remote server and the remote client. The remote server consists of a database that stores all information about each lamppost location and operating status. This information can be accessed through a mobile application by the operator in order to carry out the necessary maintenance of the lamp of each pole. Furthermore, the operator, when installing a new lamppost, can add its location to the database, using the mobile application. In addition, the database stores information on available parking spaces detected by the camera. When a user, a car driver, wants to know where there are empty parking places, he can access a website that informs him of the location of the empty parking spaces.

3.4 System Architecture

Using the system overview diagram information, one can describe the system in two different architectures: hardware architecture, as how the hardware modules interfaces with itself, what are the physical components of the system, and software architecture, which details how the information is processed among different software layers.

3.4.1 Hardware Architecture

3.4.1.1 Local System

In figure 3.5, one can see the diagram that represents the main hardware components of the system.

The Raspberry Pi is the main component in the system, processing all the information given by the sensors and the camera. The Raspberry Pi is connected to a LoRa module, for the local system communicate with the remote system. In order to control the lamp brightness, a driver is used, taking a signal from the Raspberry Pi, and system power as inputs.

The local system is powered by the power grid, through a power module. This module contains an Alternating Current (AC)/Direct Current (DC) converter, and eventually a step-down converter, to power the Raspberry Pi and its associated sensors.

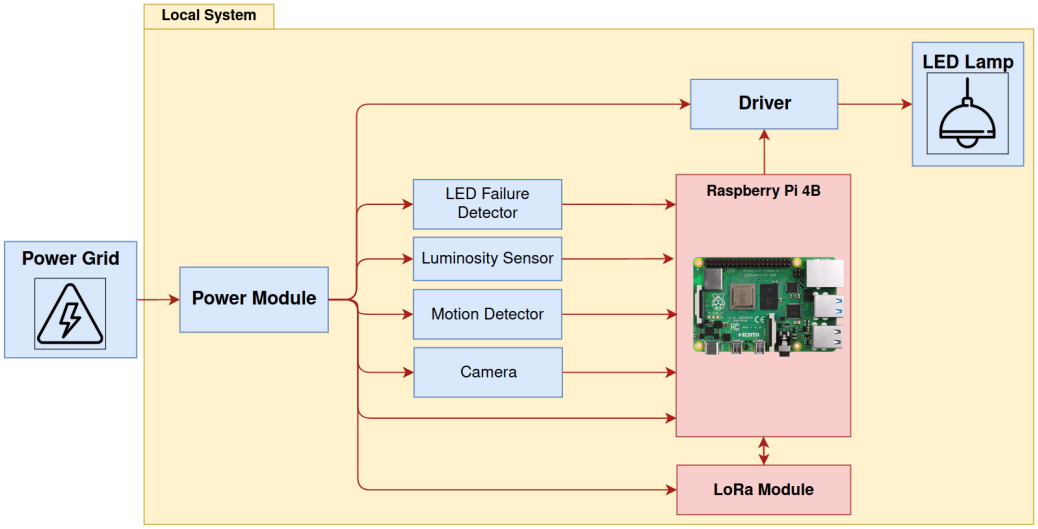


Figure 3.5: Local System Hardware Architecture Diagram.

3.4.1.2 Gateway

The hardware architecture of the gateway is shown in figure 3.6. The purpose of this device is to link the local systems to the remote system, so the hardware needed is only the LoRa communication module, as well as the Raspberry Pi to manage all communications.

As in the local system, there is a power module to power the gateway components, the Raspberry Pi and LoRa module.

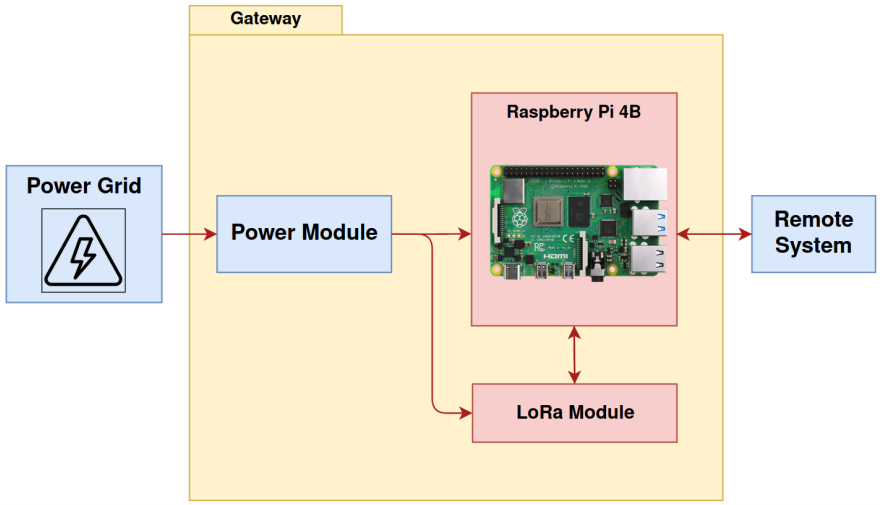


Figure 3.6: Gateway Hardware Architecture Diagram.

3.4.2 Software Architecture

The software architecture is divided into three layers:

- The **Operating System** layer, which is composed by the Operating System drivers and Board Support Packages;
- The **Middleware** layer, which includes software for abstracting the lower level layer packages. It works as a pipe since it links two applications, in different layers, so that data can be easily transmitted;
- The **Application** layer, where the core functionality of the program is built, with a resource for the API's in the lower level layers.

3.4.2.1 Local System

As shown in figure 3.7, the operating system layer is composed by the sensor drivers, such as the LED Failure Sensor, the Luminosity Sensor, the Motion Detector, the camera, and also the LoRa communication driver. In the middleware layer are the tools needed to process the images from the camera, to acquire data from sensors and to communicate via LoRa protocol with the gateway. The application layer manages the communication with the gateway.

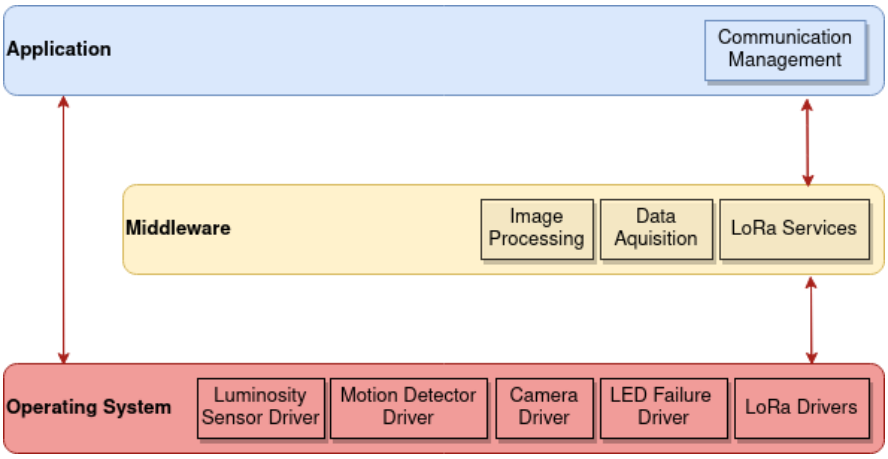


Figure 3.7: Local System Software Architecture Diagram.

3.4.2.2 Gateway

In figure 3.8 is shown the software architecture of the gateway device. The operating system layer is composed by the communication drivers, responsible of providing resources to establish a TCP/IP connection, with the remote system, and establish a LoRa connection with the local systems. The middleware layer deals with LoRa services and with TCP/IP framework. The application layer manages all communications between the street lampposts network and the remote system.

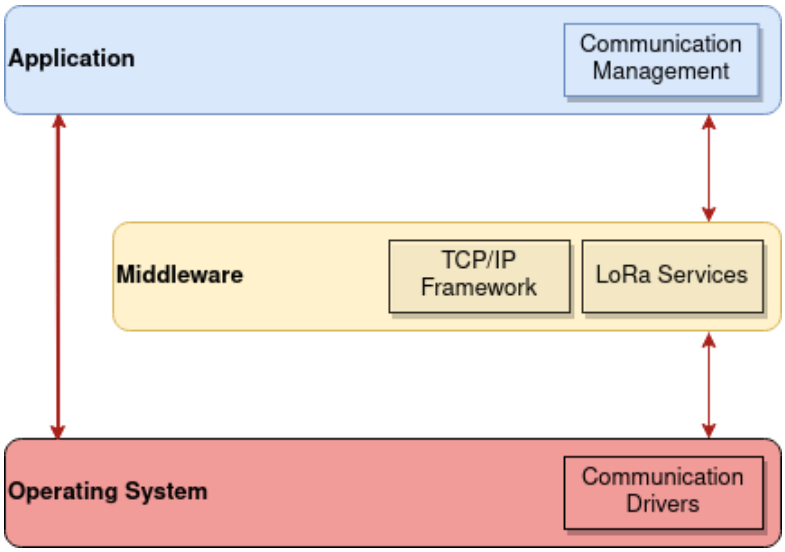


Figure 3.8: Gateway Software Architecture Diagram.

3.4.2.3 Remote System

In figure 3.9 is shown the software architecture of the remote system. The operating system layer is composed by the communication drivers, more specifically, TCP/IP, that are needed to establish a connection with the gateway. In the middleware layer, there is the TCP/IP framework. In the application layer there is the GUI, that allows the interaction with the user, through a mobile application and a web site. There are also the communication and database management, responsible of editing the database when required and supplying all the information requested by the GUI.

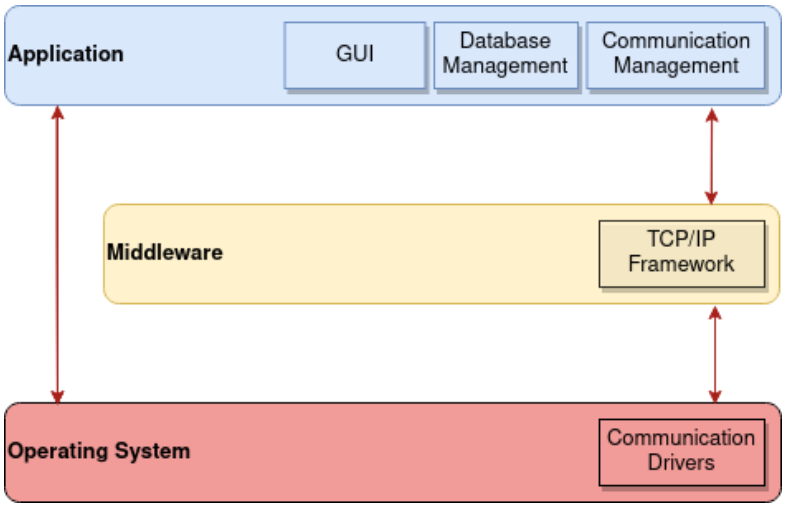


Figure 3.9: Remote System Software Architecture Diagram.

3.4.3 Database E-R Diagram

In the database will be stored all the information about each lamppost, its operators and parking spots. In the figure 3.10 one can see the Entity Relationship Diagram (E-R Diagram), displaying the relationships between the entities.

This database has five entities: "Lamppost", "Location", "Region", "Operator" and "Parking Space". A lamppost has a unique identification, his location coordinates and the status of the lamp (lamp at minimum bright level/ lamp at maximum bright level/ lamp OFF/ lamp broken). The location is defined by the coordinates, the post code associated and the street name. A region has multiple locations associated, that are defined by the post code. This entity has also information about parish, county, district of the specified region and the operator responsible for the lampposts in that region. The entity "operator" has the operator identification, the operator name and the operator pin code (used to login in the mobile application). A parking spot is defined by the entity "Parking Space" and has the attributes identification of the parking space, its location coordinates, the parking space type (normal park/ park for disabled people/ restrict park) and the park status (available/ not available).

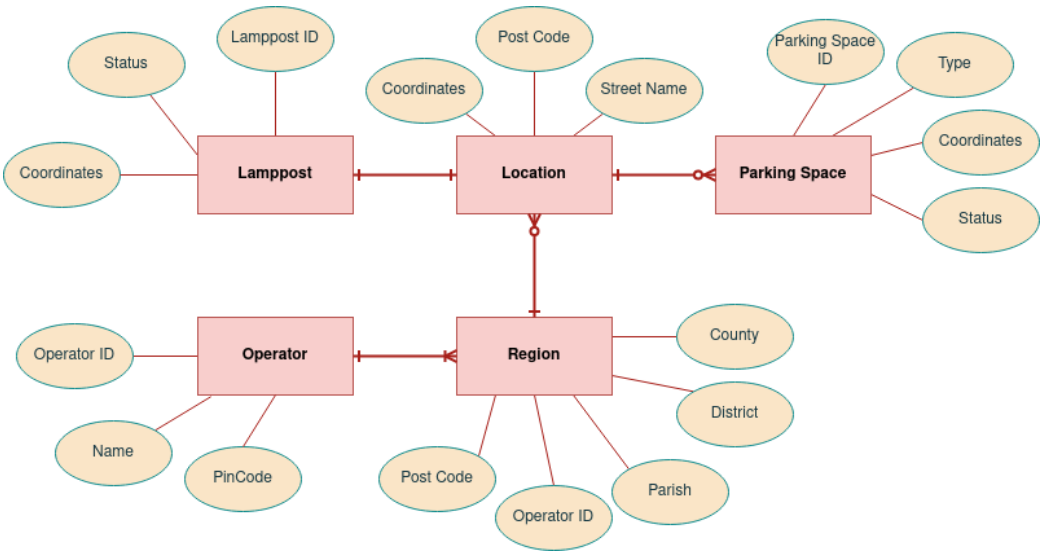


Figure 3.10: Database E-R Diagram.

Chapter 4

System Analysis

4.1 Local System

4.1.1 Events

To better understand the local system behavior, it is necessary to be aware of the events that may occur, defining how the system will respond to each one of them, as shown in table 4.1.

Event	**System Response**	**Source**	**Type**
Low luminosity detected	Put lamp at a predefined minimum bright level	Environment	Asynchronous
Motion detected	Put lamp at maximum bright level	User	Asynchronous
LED failure detected	Notify remote system	Local system	Asynchronous
Requested to turn on the lamp	Put lamp at maximum bright level	Remote system	Asynchronous
Camera sample period	Acquire camera frame and do image processing	Timer	Synchronous
Sensors data acquisition	Sample sensor values	Timer	Synchronous
Update system information	Send data to remote system	Local system	Asynchronous

Table 4.1: Events: Local System.

4.1.2 Use Cases

The local system use cases are presented in figure 4.1. A street passerby, a car or a pedestrian, can interact with each local system by moving in the vicinity of the lamppost, triggering it's motion detector, or by clearing a parking space.

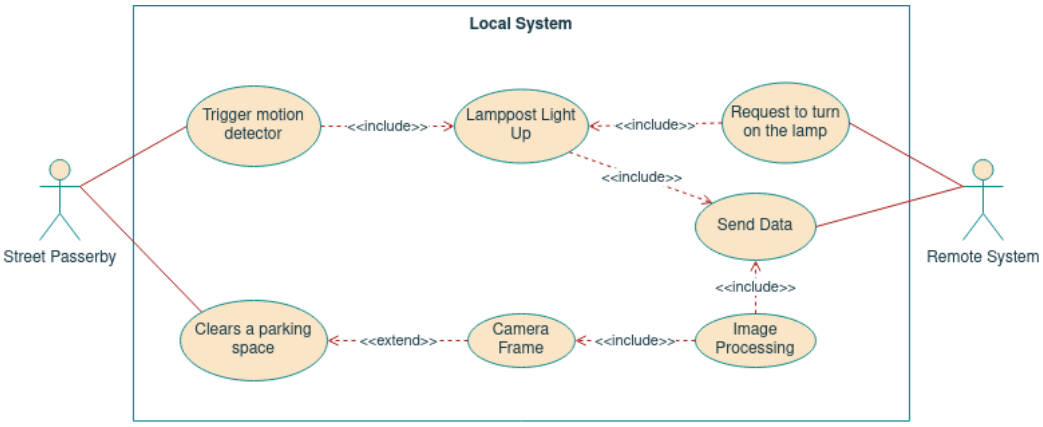


Figure 4.1: Use Cases: Local System.

When movement is detected, the lamp is put at maximum bright level. The system then informs this occurrence to the remote system, through the gateway, in order to turn on the neighbor lampposts. The opposite can also happen, when a neighbor local system, with the lamp already on, requests this local system to turn on its lamp, being this request handled by the remote system.

Moreover, the local system is periodically doing image processing after the capture of camera frames. So when the street passerby clears a parking space, the system will detect that, and will send that notification to the remote system.

4.1.3 State Chart

In figure 4.2 its represented the state chart of the local system.

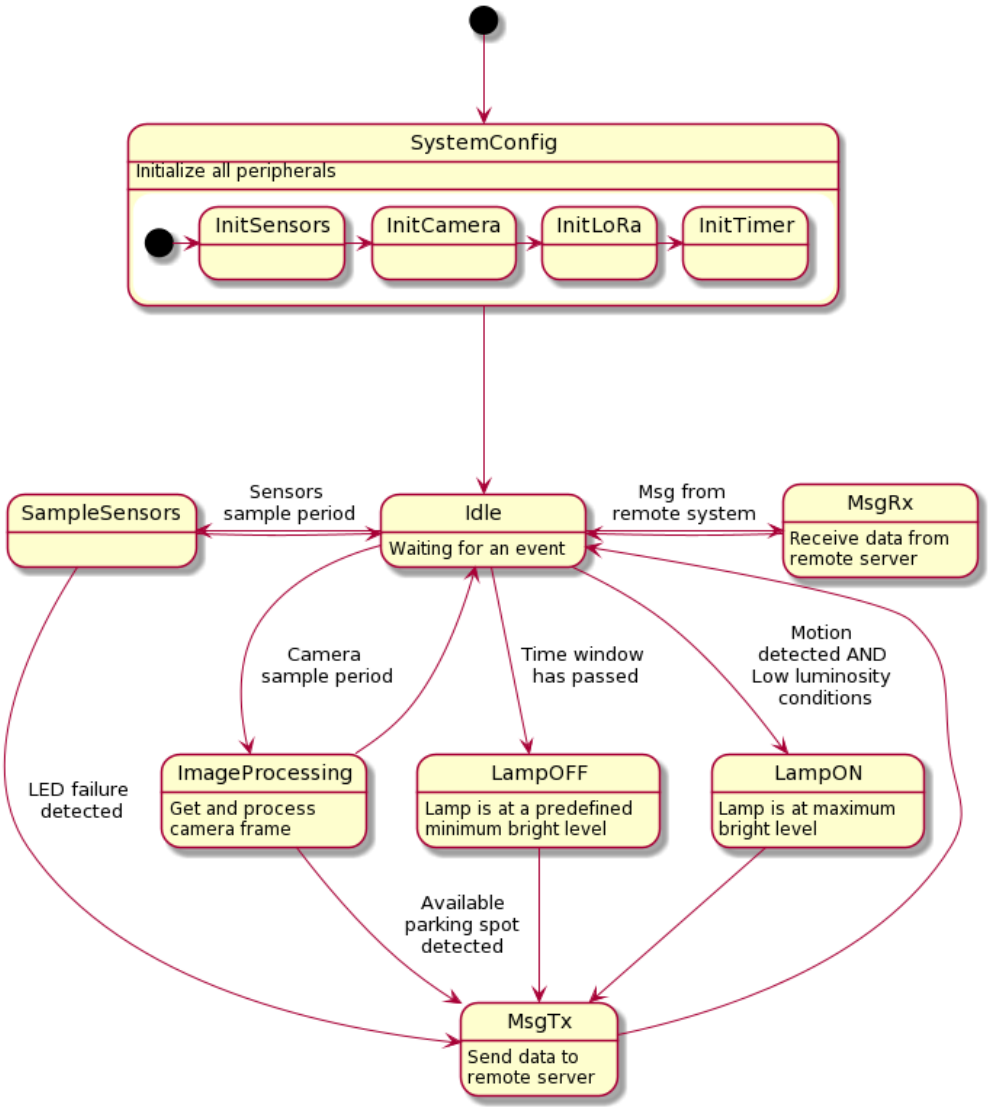


Figure 4.2: State Chart: Local System.

It initiates with the system configuration, initializing all peripherals from this system, including the sensors, the camera, the LoRa communication module and the timer for sensors data acquisition. The timer is used to

sample sensor values periodically and to acquire image frames through the camera. After that, the system enters an idle state.

When motion is detected and the luminosity sensor detects low luminosity conditions, for example, during the night, the lamp is put at its maximum bright level, for a predefined time window, and this information is sent to the remote server. When this time window passes, the lamp is put at a predefined minimum bright level, and again, informs the remote system of this event. Every time motion is detected, when the lamp is already on, the time window for the lamp being on is restarted.

To do the sensors data acquisition, including the camera frame sampling, are used sample periods, that periodically triggers the sampling of the sensors. In "SampleSensors" state, the system checks the luminosity sensor levels and if the LED failure detector hasn't detected a failure on the lamp. If a LED failure is detected, that information is sent to the remote system. To do the image processing, it is also used a sample period to get image frames through the camera. If there is an available parking space detected, that information is sent to the remote server.

When the local system detects that the remote system is sending a message, either to turn on the lamp of the local system, or other reason, the system goes to a dedicated state to receive that message.

4.1.4 Sequence Diagram

In figure 4.3 it is shown the local system sequence diagram. When a street passerby triggers the motion detector, the local system turns on its lamp, and, at that moment, using the communication management of the system, that information is sent to the remote system. If, no more movement is detected, the lamp turns off after a predefined time (turn off time), and again, the lamp status is updated in the remote system.

An alternative of an interaction with the local system is when the remote system requests the local system to turn on its lamp, this being processed in a similar way to the previous example, when the user triggers the motion detector. The remote system does this requests to local systems to dynamically turn on the lampposts as a passerby is walking down a street.

Note that, regarding the lamp control, one may use "turn on" to represent the lamp bright transition from minimum bright level to maximum bright level, and use "turn off" to represent the opposite, the transition from maximum bright level to minimum bright level. The lamp is only off when

low luminosity conditions are not verified, i.e, during the day.

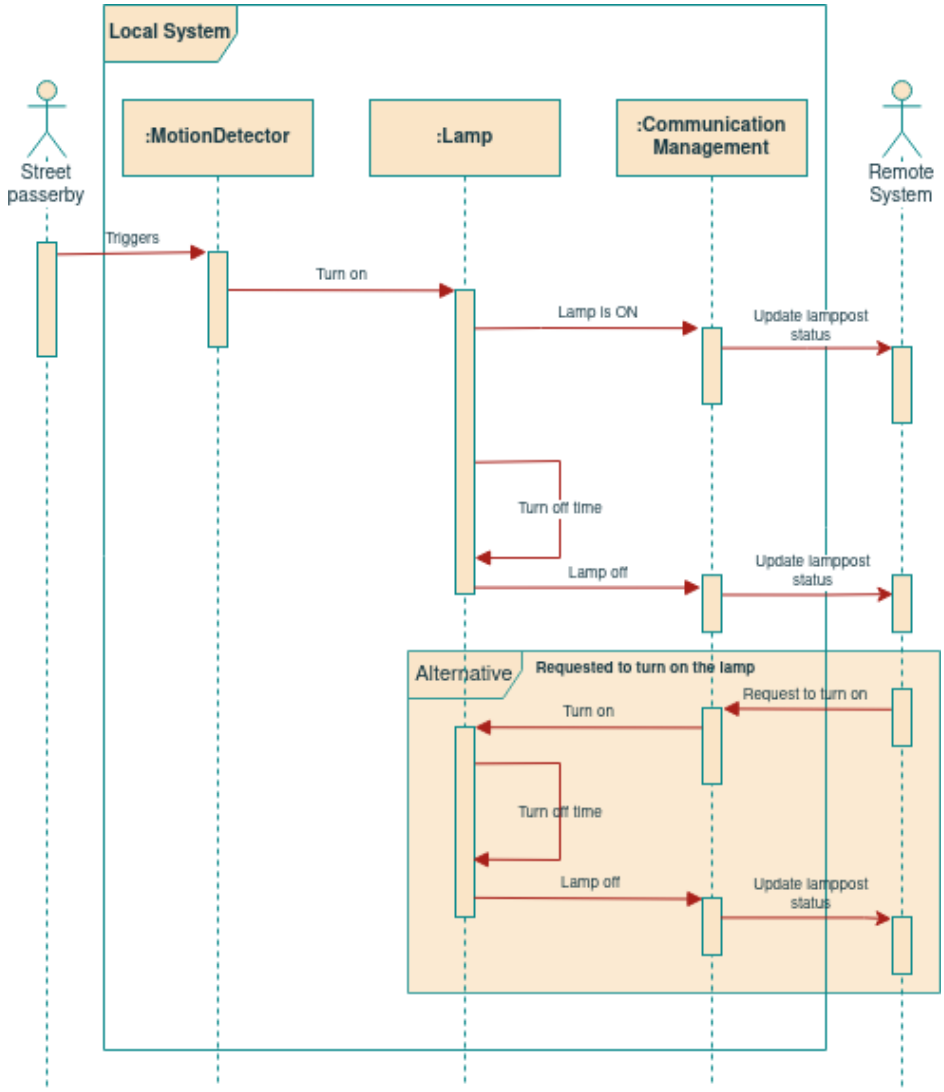


Figure 4.3: Sequence Diagram: Local System.

In figure 4.4 it is shown the local system data acquisition sequence diagram. A timer is used to trigger two different samplings, one to sample sensor values, other to sample image frames from the camera.

When the sensors sample period occurs, the sampling is started, gathering values from the luminosity sensor and from the LED failure detector. If a LED failure is detected, that is sent to the remote system through the communication management system.

When the camera sample period occurs, one gets an image frame from the camera and does image processing, in order to avail if there is an available parking spot. If that comes true, it's communicated to the remote system, like in the previous example.

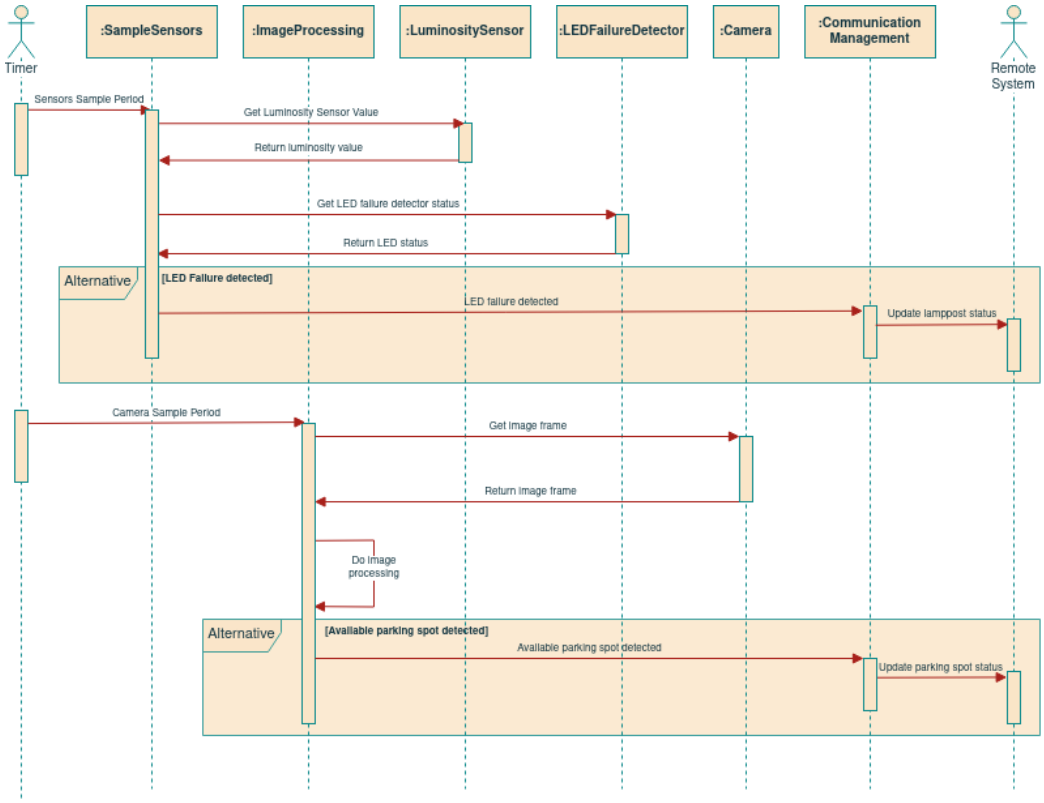


Figure 4.4: Sequence Diagram: Local System Data Acquisition.

4.2 Remote System

The remote system is composed by the remote server and remote clients: mobile application and web site. In this section are shown the remote system events, state charts and sequence diagrams.

4.2.1 Remote Client

4.2.1.1 Mobile Application

Events In order to better understand the system, it is necessary to identify the events that may occur, how the system will respond to the event, what caused the event and what type of event it is. For the remote client mobile application, the events that may occur are presented in table 4.2.

Event	System Response	Source	Type
Login	Show application main screen if successful	Operator	Asynchronous
Obtain geolocation	Request device geolocation	Mobile device	Asynchronous
App notification	Notifies the operator about the lamppost status	Remote Server	Asynchronous
Register operator	Add operator information to Database (DB)	Operator	Asynchronous
Modify lamppost	Update lamppost information on DB	Operator	Asynchronous
Add new lamppost	Add new lamppost information to DB	Operator	Asynchronous
Remove lamppost	Remove lamppost from DB	Operator	Asynchronous

Table 4.2: Events: Remote Client Mobile Application.

Use Cases The mobile application use cases diagram are represented in figure 4.5, showing that the main actors in the system are the operator, the remote server and the mobile device. The operator can perform operations such as login, logout, register, modify information about a lamppost, remove a lamppost and register newly installed posts. For this, register a new

installed post, the remote client acquires current device location from the mobile device and sends it to the remote server.

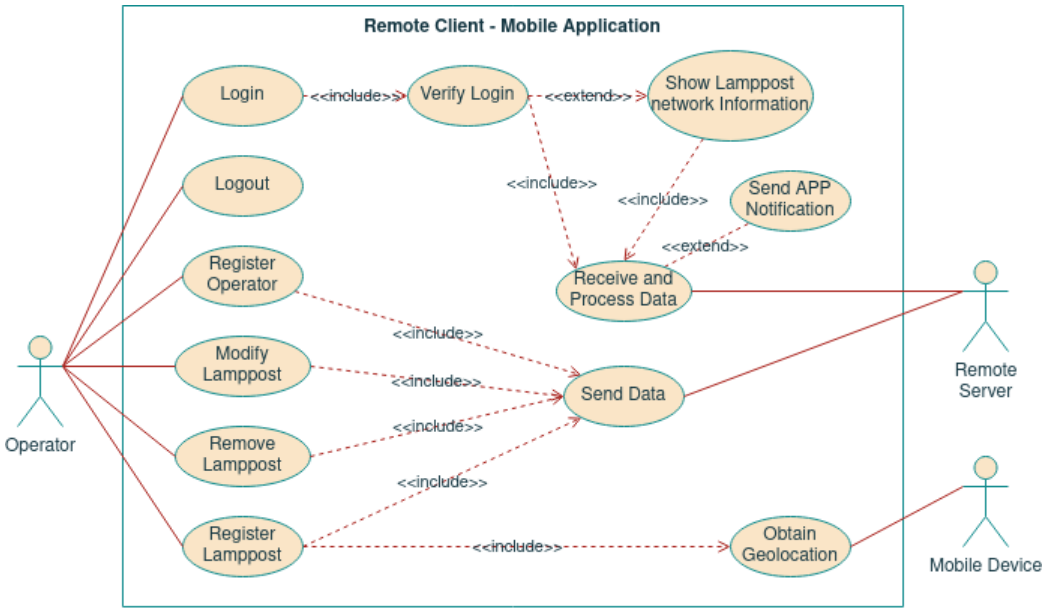


Figure 4.5: Use Cases: Remote Client Mobile Application.

State Chart In the figure 4.6 is represented the state chart of the mobile application. It initiates with the system configuration, showing a home screen that allows the operator, the application user, to log into the system or register himself, if he doesn't have login credentials. After a successful login, the system will show information about the lampposts associated to the logged in operator and he can do operations like register lampposts, remove lampposts, modify lampposts information and logout of the system.

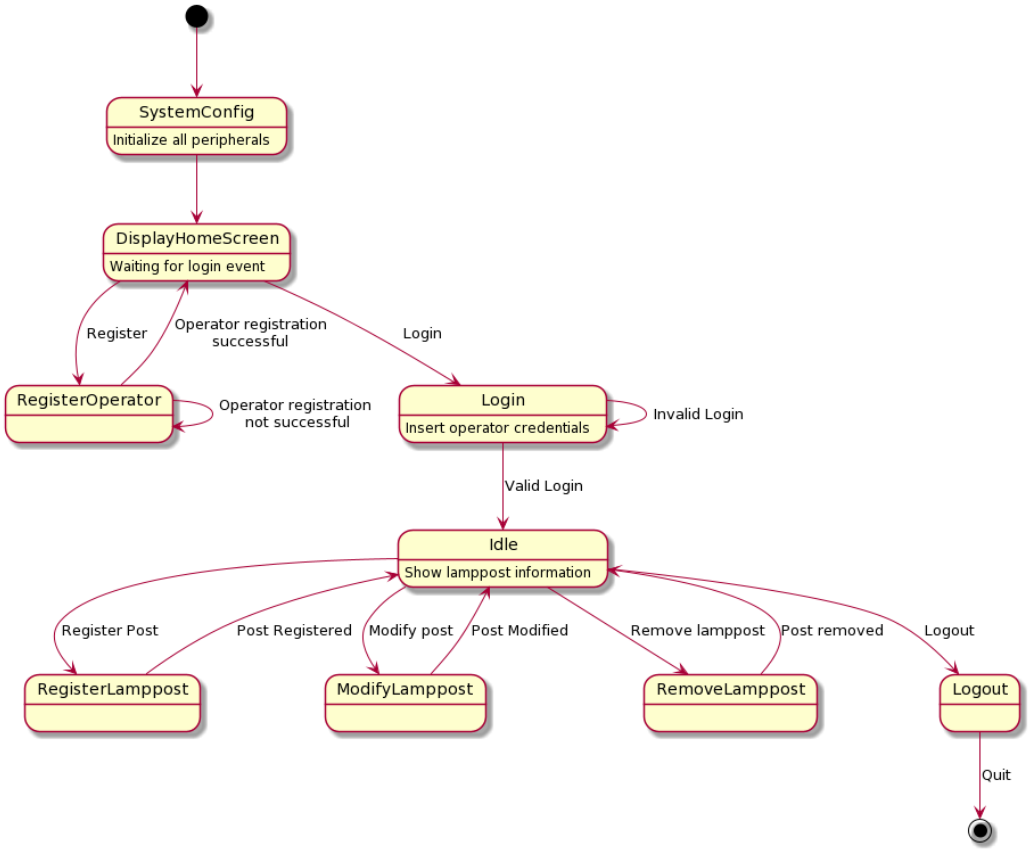


Figure 4.6: State Chart: Remote Client Mobile Application.

Sequence Diagram In figure 4.10 is represented the sequence diagram of the mobile application. Most of the actions are triggered by the operator, starting with the registration or login operations in the application. If the login is valid, information about the network of lampposts will be shown and, depending on the interaction with the operator, the application may have different execution flows: registration of a lamppost; changing information about a lamppost or removing a lamppost; verification of the existence of damaged posts, notifying the operator if so. If the login is invalid, the application returns an error to the operator.

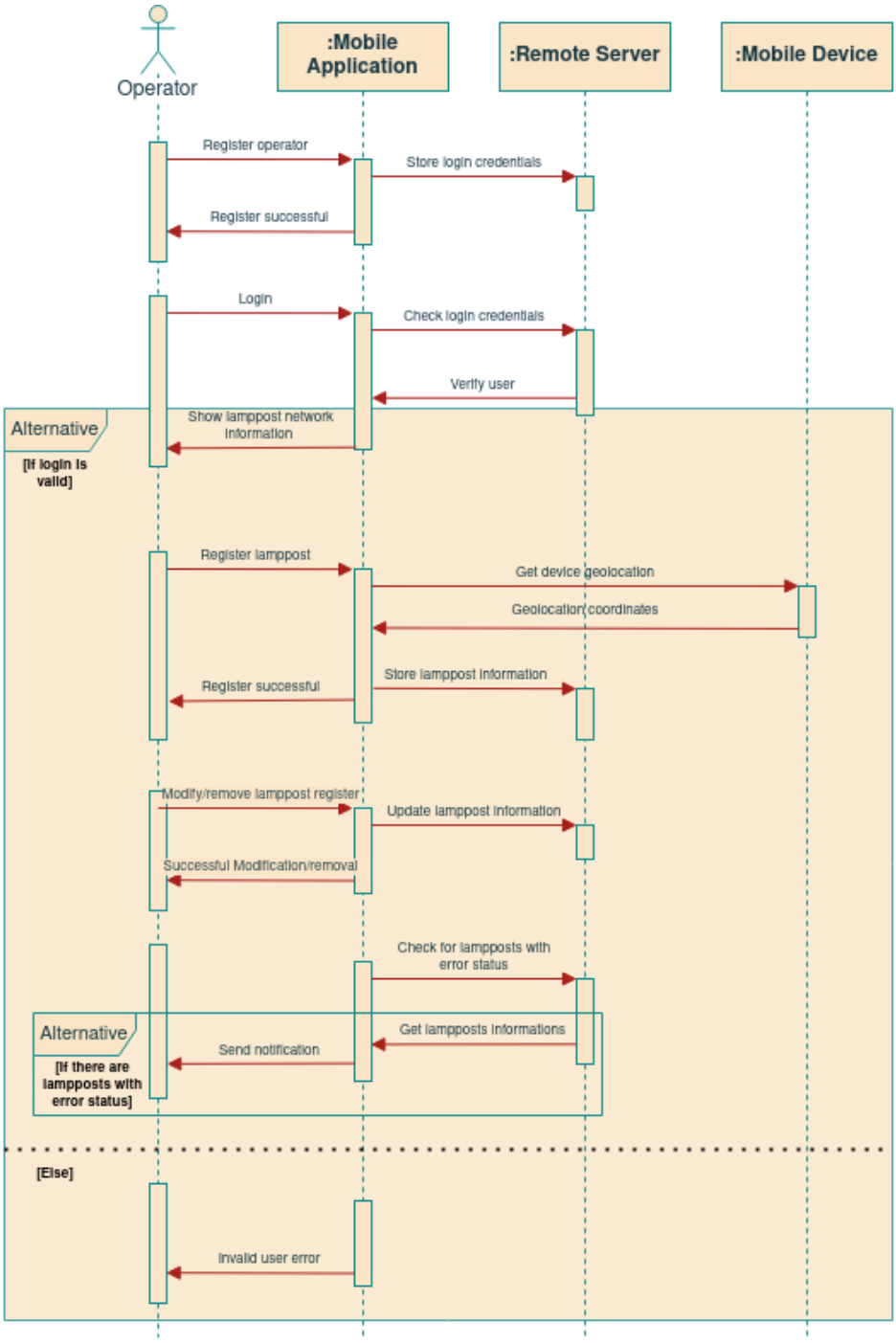


Figure 4.7: Sequence Diagram: Remote Client Mobile Application.

4.2.1.2 Web Site

Events In order to better understand the remote client web site, it is necessary to identify the events that may occur, which are presented in table 4.3.

Event	System Response	Source	Type
Insert location	Show parking spots	User	Asynchronous
Obtain geolocation	Request device geolocation	Mobile Device	Asynchronous

Table 4.3: Events: Remote Client Web Site.

Use Cases The web site use cases are shown in figure 4.8. The main actors are the user, the remote server and a mobile device. In order to know if there are available parking spaces in a certain location, the user can enter a location (inserting the street name, for example) or, as in the mobile application, use his mobile device to obtain the location automatically. The remote server lets the user know where there are empty parking spaces.

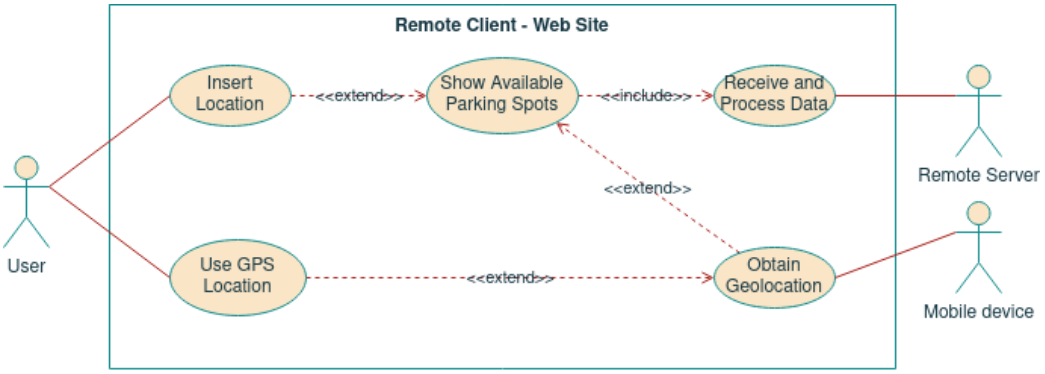


Figure 4.8: Use Cases: Remote Client Web Site.

State Chart In figure 4.9, one can see the web site state chart. The system initiates with the system configuration. Then the user can use his mobile phone's location (through the GPS tracking system) or type manually the location address. If the user enters a location manually (the street name, for example), it will be checked and, if valid, the available parking spaces will be displayed.

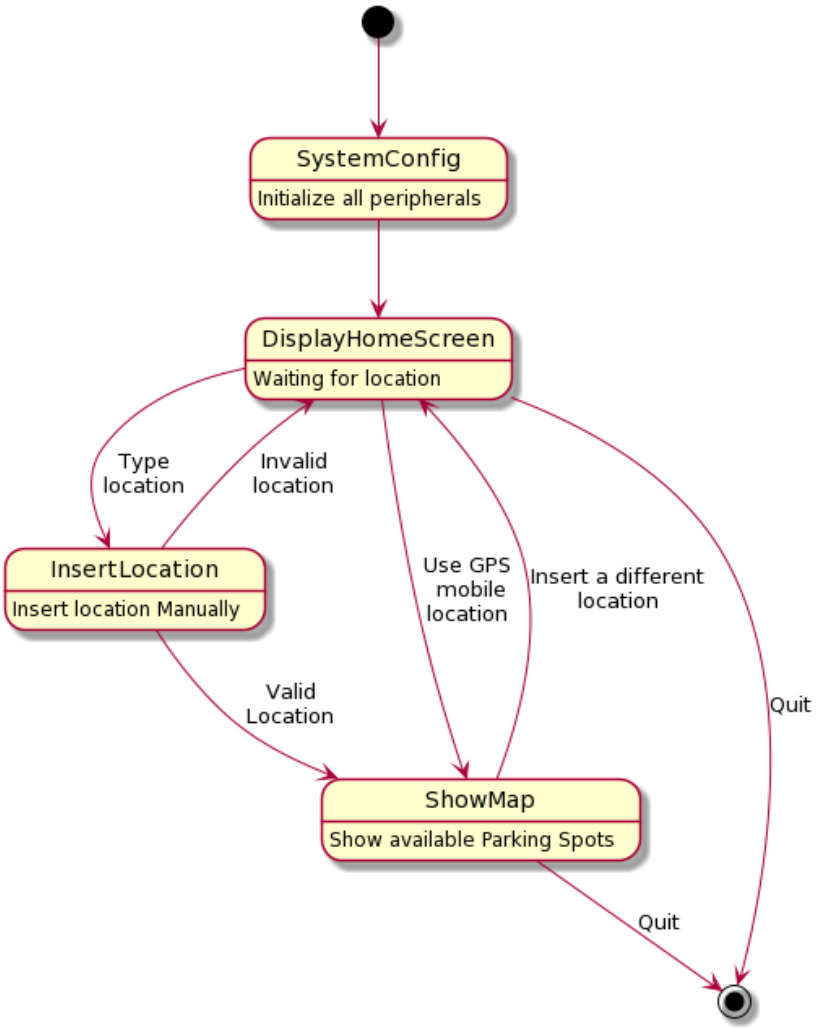


Figure 4.9: State Chart: Remote Client Web Site.

Sequence Diagram The sequence diagram of the web site is shown in the figure 4.10. To know where there are empty parking spots, the user can insert manually a location or use his Global Positioning System (GPS) location. In both cases the web site asks the remote server if there are available parking spots near the location and displays them to the user. However, in the case of using the GPS location, the web site has to get the GPS coordinates from the mobile device running the web site.

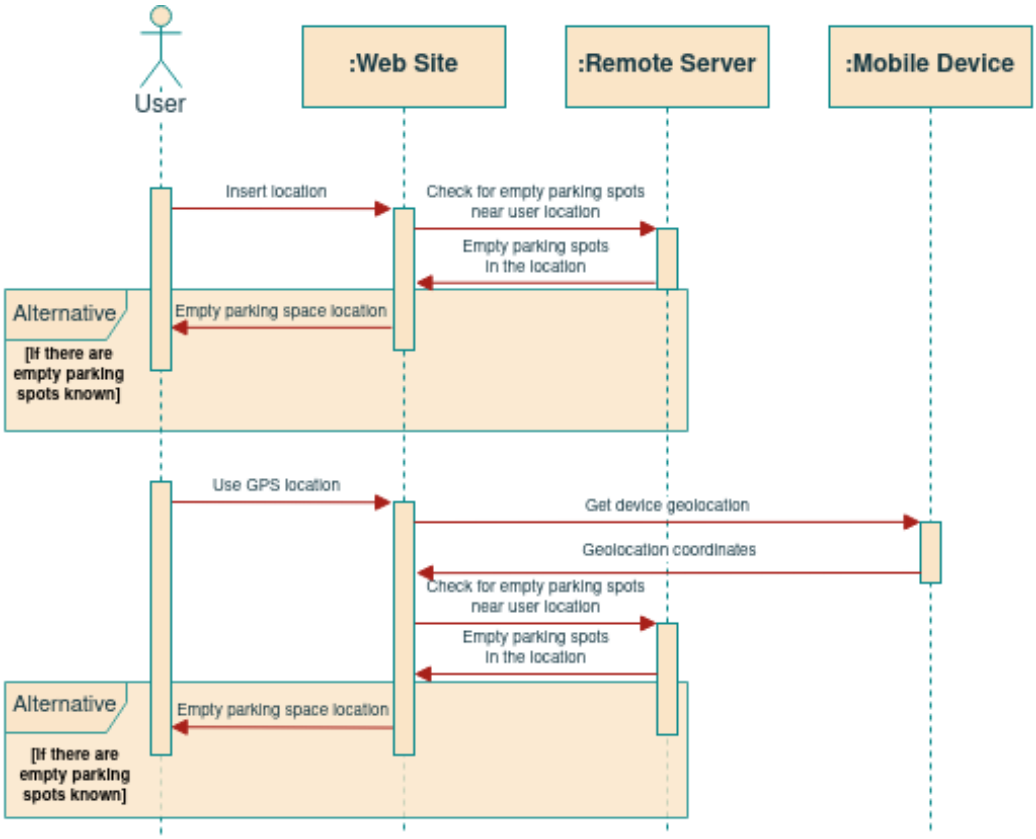


Figure 4.10: Sequence Diagram: Remote Client Web Site.

4.2.2 Remote Server

Events In order to better understand the remote server, it is necessary to identify the events that may occur, which are presented in table 4.4.

Event	System Response	Source	Type
Connection Request	Accept/ decline connection	Remote client	Asynchronous
Check Log-in Credentials	Validate username and pin-code	Remote client (Mobile App)	Asynchronous
Add new lamppost	Add new lamppost info to DB	Remote client (Mobile App)	Asynchronous
Remove lamppost	Remove lamppost from DB	Remote client (Mobile App)	Asynchronous
Update lamppost information	Update lamppost info on DB	Remote client (Mobile App); Local System	Asynchronous
Check for lampposts with error status	Read and send lampposts with error status to Mobile App	Remote client (Mobile App)	Asynchronous
Check for empty parking spots	Read and send available parking spots to Web Site	Remote client (Web Site)	Asynchronous

Table 4.4: Events: Remote Server.

Use Cases The remote server use cases are shown in figure 4.11. The remote(s) client(s) can interact with the remote server in various ways, as previously seen. This system only executes the demanded commands, from the remote clients or from the local systems, by accessing database information. The database interacts with the remote server by providing read, modify, remove or add register functions.

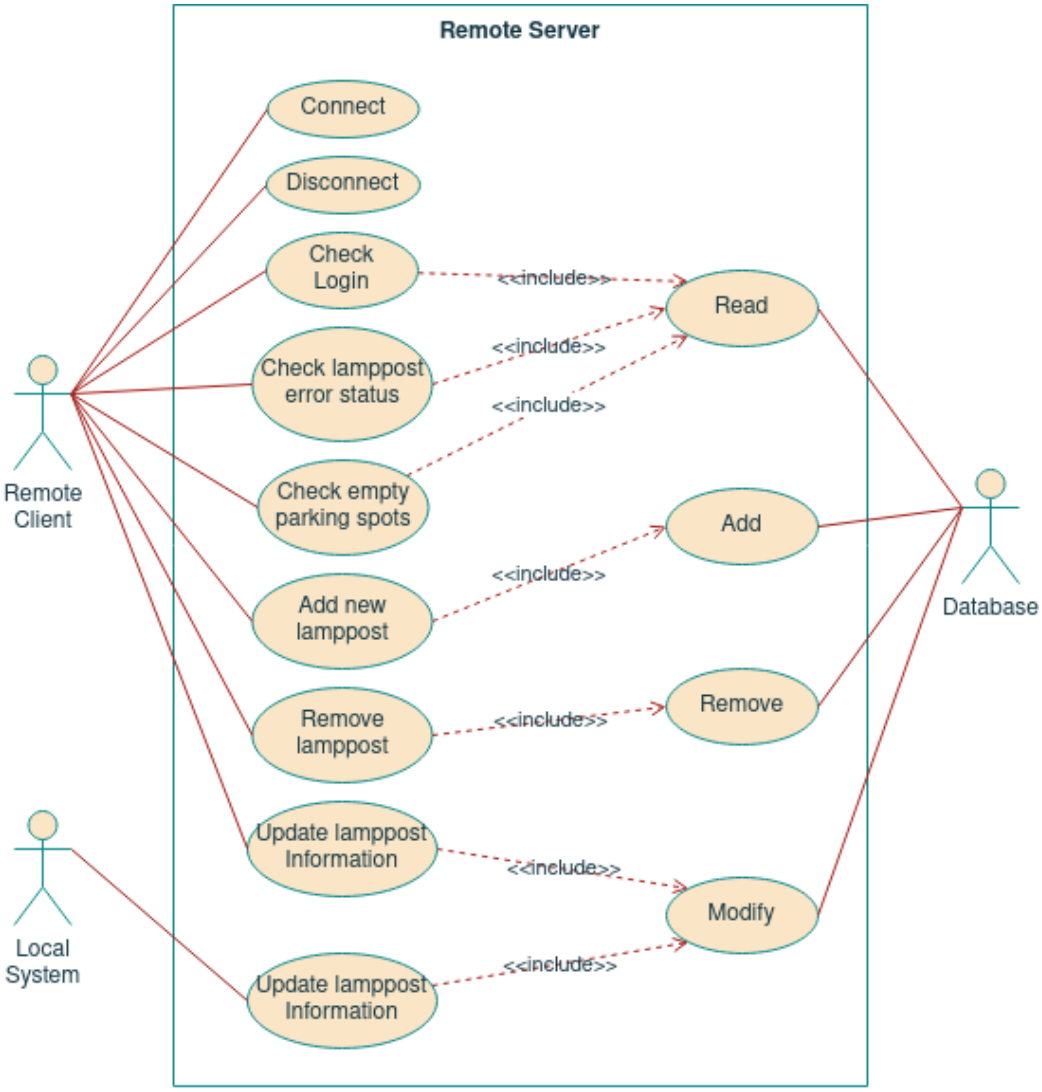


Figure 4.11: Use Cases: Remote Server.

State Chart In figure 4.12, one can see the remote server state chart. After power on, the remote server does the system configuration, by initializing communication and database management services, and after that, the system executes concurrently these services until system's power off. These services give to the remote server the capability of processing received requests/commands, from the remote clients or from the local systems, and execute them, by accessing to database information.

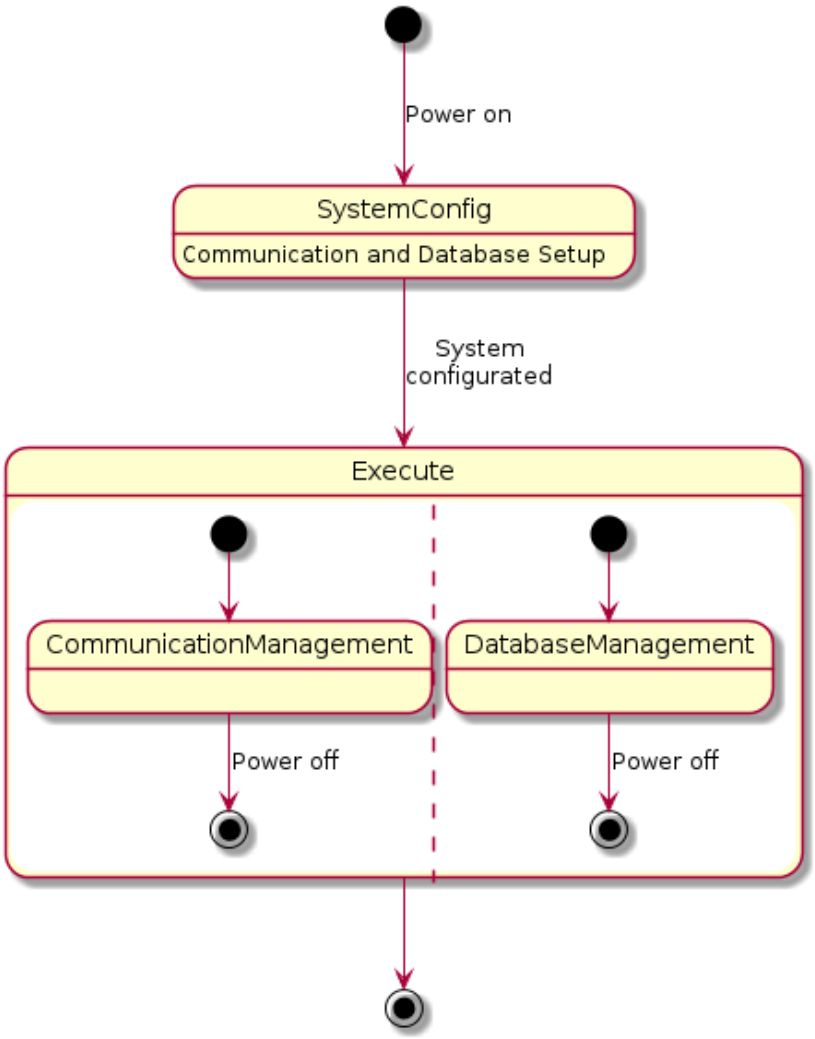


Figure 4.12: State Chart: Remote Server.

Sequence Diagram The sequence diagram of the remote server is shown in the figure 4.13. As seen previously, the remote client and the local system can execute several different actions. That is only possible if the remote server accepts the connection made from the remote client. After that, the remote server processes all received communications and, if needed, accesses the database to do so.

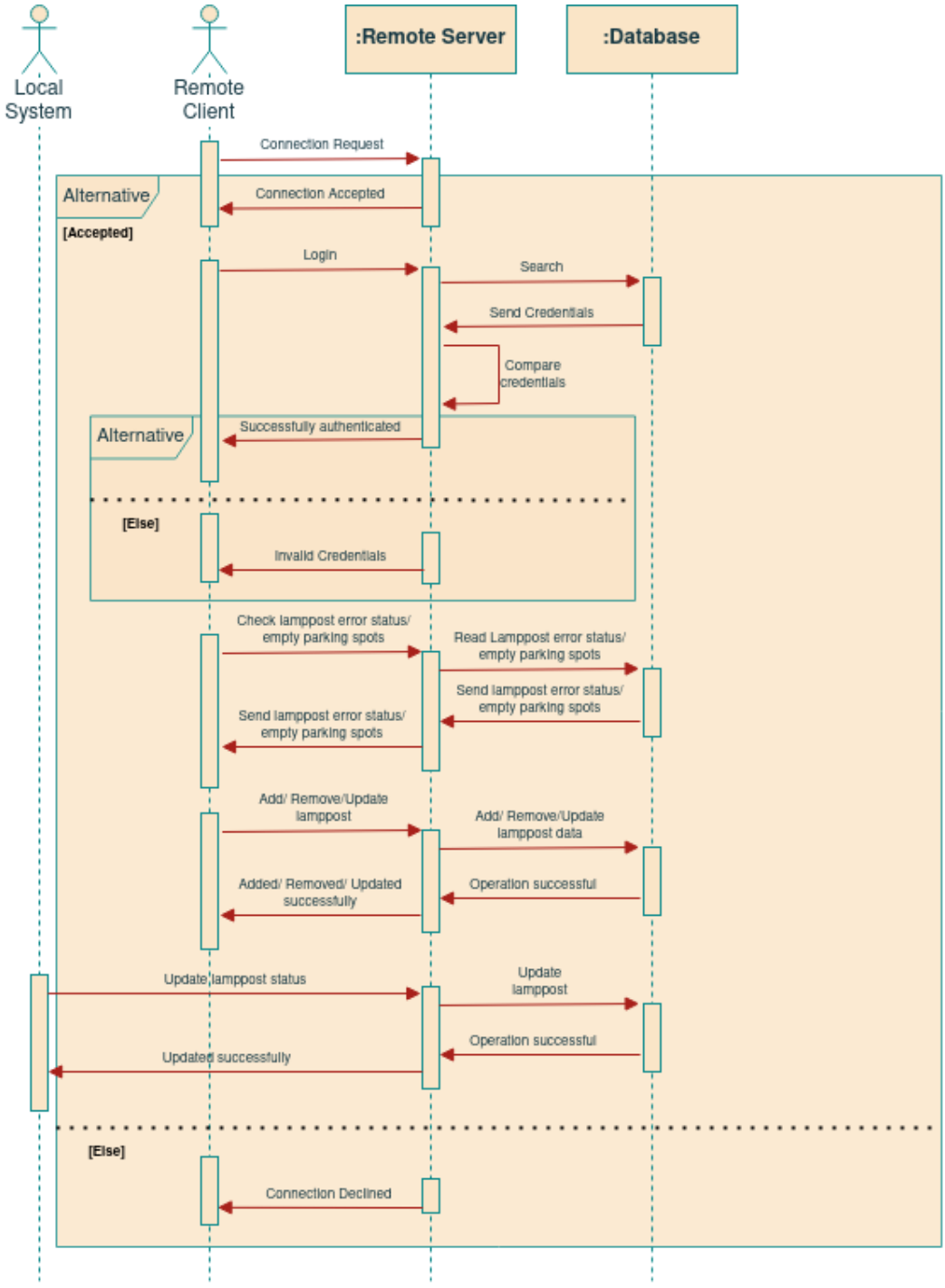


Figure 4.13: Sequence Diagram: Remote Server.

4.3 Estimated Budget

In table 4.5 is shown the estimated budget of one local system and a gateway, excluding their external casing.

Device	**Product**	**Price(€)**
Gateway	Raspberry Pi 4B	63,50
	Lora Module	7,90
	Power Module	12,95
	Gateway Cost	**84,35**
Local System	Raspberry Pi 4B	63,50
	Lora Module	7,90
	Power Module	12,95
	Video camera	15,95
	Motion detector	4,60
	Luminosity sensor	4,90
	LED lamp	3,63
	Driver (MOSFET)	1,00
	Basic Eletronic Components	5,00
	Local System Cost	**119,43**
	Total	**203,78**

Table 4.5: Estimated budget.

4.4 Task Division and Gantt Chart

In figure 4.14, is represented the Smart Street Lighting project schedule in form of a Gantt chart.

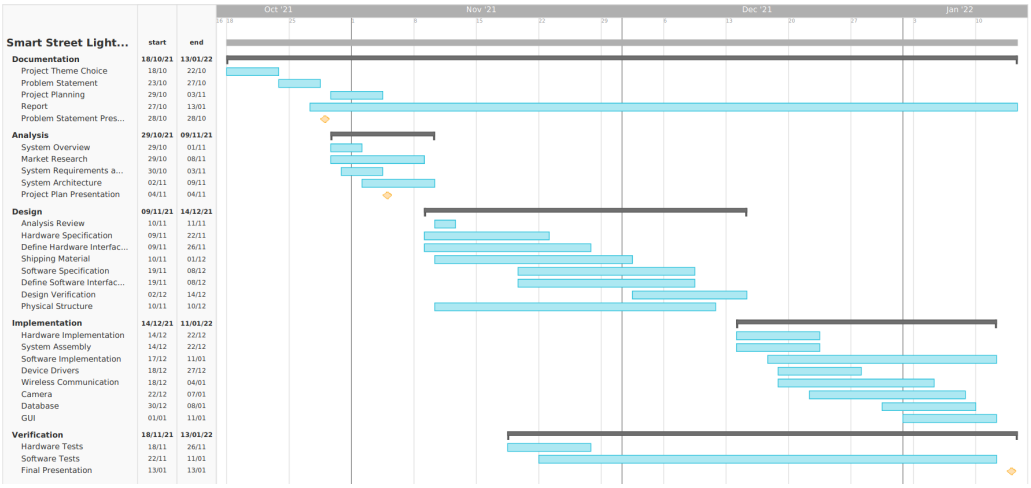


Figure 4.14: Gantt chart.

Chapter 5

Theoretical Foundations

In this chapter the theoretical foundations for the development of this project are presented, creating a solid groundwork for the design and implementation process.

5.1 Communication Protocols

In order to establish communication between the peripherals and between different systems, there is the need to use communication protocols, depending on what are supported by these devices and what are the best in the context of the application.

5.1.1 LoRaWAN

LoRa (Long Range) is a radio modulation technology for wireless LAN networks in the category of LPWA network technologies. LoRa was developed by Cycleo and later acquired by Semtech, the founding member of LoRa Alliance, an open, non-profit association that supports the LoRaWAN protocol. LoRaWAN is a network (protocol) using LoRa. [19]

LoRa technology uses the unlicensed frequency band (ISM), like 433 MHz, 868 MHz (in Europe), 915 MHz (in Australia and North America) and 923 MHz (in Asia). LoRa is the physical layer or the wireless modulation utilized to create a long range communication link, which may cover more than 10 km in line of sight. Many legacy wireless systems use Frequency-Shift Keying (FSK) modulation as the physical layer because it is a very

efficient modulation for achieving low power. LoRa is based on Chirp Spread Spectrum (CSS) modulation, which maintains the same low power characteristics as FSK modulation but significantly increases the communication range, robustness to interference. LoRa is the first low cost implementation for commercial usage.

The LoRaWAN specification is a LPWA networking protocol that targets key IoT requirements such as bi-directional communication, end-to-end security, mobility and localization services, whose baud rates range from 0,3 kbps to 50 kbps. LoRaWAN network architecture is deployed in a star-of-stars topology in which gateways relay messages between end-devices and a central network server. Each gateway is connected to the network server via standard IP connections, acting as a bridge by converting RF packets to IP packets, and vice versa. The gateway only performs the forwarding of the data packets without any security protection. The end device communicates with one or more gateways by a single-hop LoRa or FSK. LoRaWAN the protocol stack shown in figure 5.1, remembering also figure 3.3. [18]

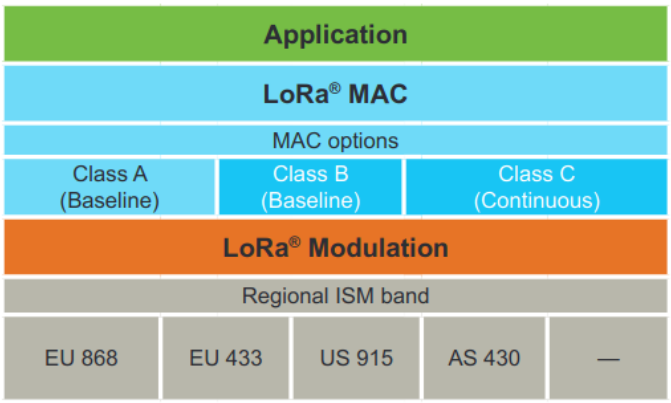


Figure 5.1: LoRaWAN protocol stack.

LoRa technology uses ISM bands, which brings vulnerability to the network. The attacker can listen on the address of the legal terminal and generate forged packets to the gateway to cause congestion. The attacker can also use his own LoRa device to send the maximum length preamble to occupy the channel maliciously. LoRaWAN considers network security issues in its design. LoRaWAN's security policy is to encrypt data from the end device node to the network server and the application server. The former ensures that the legal node can access the network, authenticate the data packet, and

perform integrity verification, and the latter ensures the end-to-end security of the application through the encryption of application data. [20]

LoRa Frame Structure

The LoRa frame structure is shown in figure 5.2, which comprises a preamble, an optional header and the data payload. LoRa employs two types o packet format: explicit and implicit. The explicit packet includes a short header containing information about the number of bytes, coding rate and whether a Cyclic Redundancy Check (CRC) is used in the packet.

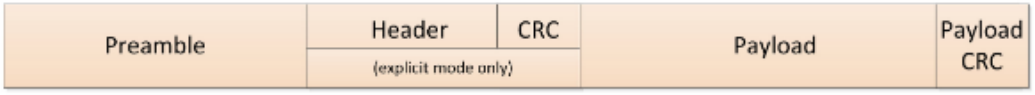


Figure 5.2: LoRa frame structure.

The preamble is used to keep the receiver synchronized with transmitter. The packet payload is a variable-length field that contains the actual data coded at the error rate either as specified in the header in explicit mode or in the register settings in implicit mode. An optional CRC may be appended.

5.1.2 I2C

The Inter-Integrated Circuit (I2C) protocol is a synchronous, half-duplex, serial communication bus that uses a master-slave strategy, where the master is the device that clocks the bus, addresses slaves and writes or reads data to and from registers in slaves. I2C interface uses two-wire to allow inter-IC communication. In figure 5.3 one can see a generalized I2C connection diagram. [21]

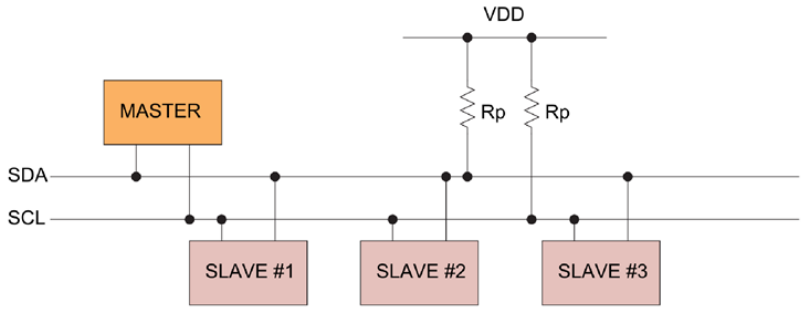


Figure 5.3: I2C configuration with master and slave devices.

I2C is implemented using two lines:

- **Serial Data (SDA)**: used to send and receive data, bit by bit, between the master and slave;
- **Serial Clock (SCL)**: carries the clock signal, which is generated by the master device.

I2C compatible devices connect to the bus with open collector or open drain pins which pull the line LOW. When there is no transmission of data, the I2C bus lines idle in a HIGH state, being passively pulled high. Transmission occurs by toggling the lines by pulling LOW and releasing HIGH. Bits are clocked on falling clock edges. Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7 bit address, 128 (2^7) unique address are available. Using 10 bit addresses is uncommon, but provides 1024 (2^{10}) unique addresses. To connect multiple slaves to a single master, one should wire them as shown in figure 5.3, with 4.7 KΩ pull-up resistors, R_P, connecting the SDA and SCL lines to V_{DD}. [22]

With I2C, data is transferred in messages. Messages are broken up into frames of data. Each one are composed by the following:

- **Start condition**: SDA line switches from HIGH to LOW state, before the SCL line does the same;
- **Stop condition**: SDA line switches from LOW to HIGH state, after the SCL line does the same;
- **Address frame**: 7 to 10 bit sequence that uniquely identifies the slave when the master wants to talk to it;
- **Read/Write bit**: bit that specifies whether the master is sending data to the slave (bit 0) or requesting data from it (bit 1);
- **Acknowledge (ACK)/No-Acknowledge (NACK) bit**: states if the address or data frame was successfully received. When successful, an ACK bit is sent, by pulling the SDA line LOW for one bit. If not, a NACK bit is sent, by leaving the SDA line HIGH.

Data Transmission

The slaves are devices that respond only when interrogated by the master, through their unique address.

1. The master sends the start condition to every connected slave;
2. The master sends each slave the address frame of the slave it wants to communicate with, along with the read/write bit;
3. Each slave compares the address sent from the master to its own address. If they match, the slave returns an ACK bit. Else, the slave leaves the SDA line HIGH;
4. The master sends or receives the data frame;
5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender, to acknowledge successful receipt of the frame;
6. To stop the data transmission, the master sends a stop condition to the slave.

I2C has advantages comparing to other communication protocols. I2C uses only two wires, and supports multiple masters and multiple slaves and ACK/NACK bit gives confirmation that each frame is transferred successfully. However, this protocol has slower data transfer rate than SPI, for example, the size of the data frame is limited to 8 bits and has more complicated hardware to implement than SPI.

5.1.3 SPI

The Serial Peripheral Interface (SPI) protocol is a synchronous, full-duplex, serial communication interface that uses a master-slave strategy, where a microcontroller takes the role of the master. SPI interfaces can have only one master and can have one or multiple slaves. SPI interface can be either 3-wire or 4-wire interface. One will focus in the popular 4-wire SPI interface, as one can see in figure 5.4. [23]

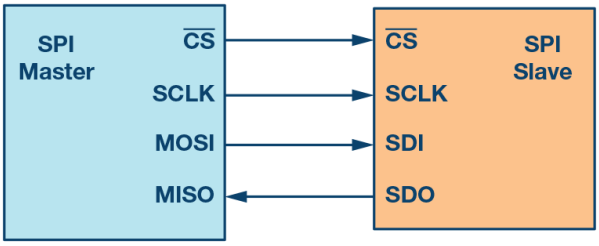


Figure 5.4: SPI configuration between master and slave devices.

4-wire SPI devices makes use of four signals:

- **Clock (SPI CLK or SCLK)**: generated by the master device, and used to synchronize the data transmitted between the master and the slave;
- **Chip Select (CS)**: signal that comes from the master and is used to select the slave. When multiple slaves are used, an individual CS for each slave is required from the master;
- **Master Out Slave In (MOSI)**: data line to transmit data from the master to the slave;
- **Master In Slave Out (MISO)**: data line to transmit data from the slave to the master.

Data Transmission

In this protocol both master and slave devices have an internal shift register, controlled by the clock signal generated by the master.

The steps of SPI data transmission are the following:

1. To begin SPI communication, the master must send the clock signal;
2. The master selects the slave by enabling the CS signal, by a low voltage state;
3. The master sends the data one bit at a time to the slave, using the MOSI line. The slave reads the bits as they are received;
4. If a response is needed, the slave returns data, one bit at a time, to the master, using the MISO line. The master reads the bits as they are received.

SPI has advantages comparing to other communication protocols. It doesn't have start and stop bits, do the data can be streamed continuously without interruption. It is full-duplex, having separate lines for receiving and sending data. Has higher data transfer rate and has a simpler slave addressing system than I2C. However, this protocol uses four wires, in contrast to two used by I2C, has no acknowledgment that the data has been successfully received and only allows for a single master. [24]

5.1.4 CSI

The Camera Serial Interface (CSI) is a specification of the Mobile Industry Processor Interface (MIPI) Alliance. It is a widely adopted, simple, high-speed protocol primarily intended for point-to-point image and video transmission between a camera and a host processor. The latest active interface specifications are CSI-2 v3.0, CSI-3 v1.1 which were released in 2019 and 2014 respectively. [25]

The CSI-2 is the most widely used camera interface in mobile and other markets. CSI-2 consists of a unique physical bus that contains a differential clock, source synchronous, and from one to four differential data lanes, as one can see in figure 5.5. This interface is called a D-PHY. This standard takes advantage of Low Voltage Differential Signaling (LVDS) to send data through multiple lanes from the camera controller to the MIPI. The LVDS standard is used to reduce electromagnetic interference and cross talk between lanes.

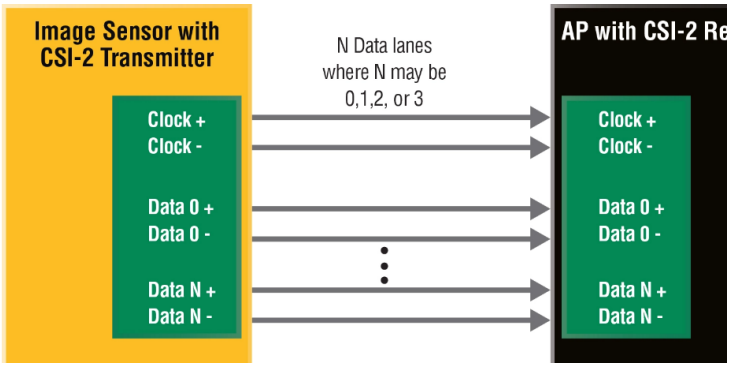


Figure 5.5: CPI-2 sensor interface.

Each data lane transmits 8-bit serial data. The higher the image sensor resolution and frame rate, the more data lanes and higher speed for each will be required. The practical limit for a CSI-2 interface is less than 1 Gbits/s data rates, but often it is less than 700 Mbits/s. [26]

5.1.5 TCP-IP

Transmission Control Protocol (TCP)/Internet Protocol (IP) is one of the most commonly used protocols for internet communication. TCP is the component that collects and reassembles the packets of data, while IP is responsible for making sure the packets are sent to the right destination. These are part of the OSI model, as shown in figure 5.6.

Figure 5.6: OSI model.

The TCP provides services to the transport layer, providing a reliable and sequenced delivery of data. One can compare TCP to while the User Datagram Protocol (UDP) provides data transportation without guaranteed data delivery or acknowledgments, which may be faster, but unreliable on the other hand. The IP part of the TCP/IP protocol, provides services to the network layer, and is used to make origin and destination addresses available to route data across networks.

Creating a TCP Server

In order to create a TCP server, one needs to perform the following steps:

1. Create a TCP socket, using *create()*;
2. Bind the created socket to a server address, using *bind()*;
3. Using *listen()*, put the server socket in a passive mode, waiting for the client to approach the server to make a connection;
4. Establish a connection, between client and server, using *accept()*; Both can now transfer data;
5. Go back to step 3, to listen for more client connections.

Creating a TCP Client

In order to create a TCP client, one needs to perform the following steps:

1. Create a TCP socket;
2. Connect newly created client socket to server, using *connect()*;

5.1.6 HTTP

HyperText Transfer Protocol (HTTP) is a protocol for fetching resources such as HTML documents. It is the foundation of any data exchange on the Web and it is a client-server protocol, which means requests are initiated by the recipient, usually the Web browser. A complete document is reconstructed from the different sub-documents fetched, for instance, text, layout description, images, videos, scripts, and more. [27]

Clients and servers communicate by exchanging individual messages (as opposed to a stream of data). The messages sent by the client, usually a Web browser, are called *requests* and the messages sent by the server as an answer are called *responses*. It is an application layer protocol that is sent over TCP, though any reliable transport protocol could theoretically be used.

Each individual request is sent to a server, which handles it and provides an answer called the response. Between the client and the server there are numerous entities, collectively called proxies, which perform different operations and act as gateways or caches, for example, as shown in figure 5.7.

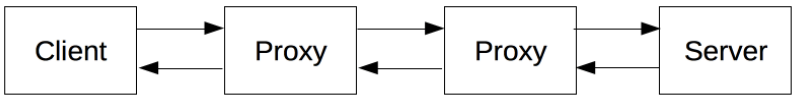


Figure 5.7: Components of HTTP-based systems.

Client-Server Communication

When a client wants to communicate with a server it performs the following steps:

1. Open a TCP connection, which is used to send a request or to receive an answer;
2. Send an HTTP message;
3. Read the response sent by the server;
4. Close or reuse the connection for further requests;

If HTTP pipelining is activated, several requests can be sent without waiting for the first response to be fully received.

HTTP Request methods

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Below are listed some of the HTTP request methods. [28]

- `GET`: requests a representation of the specified resource;
- `HEAD`: asks for a response identical to a `GET` request, but without the response body;
- `POST`: submits an entity to the specified resource;
- `PUT`: replaces all current representations of the target resource with the request payload;
- `DELETE`: deletes the specified resource;

HTTP Messages

HTTP messages are human-readable, having two types of messages, requests and responses, each with its own format.

Requests consists of the following elements:

- HTTP method, that defines the operation the client wants to perform: usually a verb like `GET`, `POST` or a noun like `OPTIONS` or `HEAD`.
- Path of the resource to fetch: the URL of the resource stripped from elements that are obvious from the context;
- The version of the HTTP protocol using;
- Optional headers, conveying additional information for the servers;
- A body for some methods like `POST`, which contains the resource sent.

Responses consists of the following elements:

- The version of the HTTP protocol using;
- Status code, indicating if the request was successful or not, and why;
- Status message: a non-authoritative short description of the status code;
- HTTP headers;
- Optionally a body containing the fetched resource.

5.2 Daemons

A daemon is a process that runs in the background and has no controlling terminal. The lack of a controlling terminal ensures that the kernel never automatically generates any job-control or terminal-related signals (such as SIGINT, SIGTSTP, and SIGHUP) for a daemon. [29]

A daemon is often created at system startup and runs until the system is shut down, being used to carry out specific tasks. It is a convention (not universally observed) that daemons have names ending with the letter d (example: *httpd*, *sshd*).

Each process belongs to a process group, and the process group can contain one or more processes. There is a process leader in the process group, the Process Identifier (PID) of the leader is the Process Group Identifier (PGID). More than one process group constitutes a "session". The process that establishes the session is the lead process for the session, and the PID of the leader is the Session Identifier (SID) of the session. Each process group in a session is called a "job". The meaning of a session is that multiple jobs can be controlled through one terminal, one foreground operation and the other running in the background. [30]

Becoming a daemon

To become a daemon, a process must perform the following steps:

1. Perform a *fork()*, after which the parent exits and the child continues.
2. The child process calls *setsid()* to start a new session and free itself of any association with a controlling terminal.
3. Clear the process umask to ensure that, when the daemon creates files and directories, they have the requested permissions. (*umask()*)
4. Change the process's current working directory, typically to the root directory (*chdir('/')*). This is necessary because a daemon usually runs until system shutdown, and if the daemon's current working directory is on a file system other than the one containing '/', then that file system can't be unmounted.

5. Close all open file descriptors that the daemon has inherited from its parent. (*close()*) (A daemon may need to keep certain inherited file descriptors open, so this step is optional, or open to variation.)

Since a daemon has no controlling terminal, the *syslog* facility provides a convenient way for daemons (and other applications) to log error and other messages to a central location. These messages are processed by the *syslogd* daemon, which redistributes the messages according to the dictates of the syslogd.conf configuration file.

Where appropriate, daemons should correctly handle the arrival of the SIGTERM and SIGHUP signals. The SIGTERM signal should result in an orderly shutdown of the daemon, while the SIGHUP signal provides a way to trigger the daemon to reinitialize itself by rereading its configuration file and reopening any log files it may be using.

5.3 Signals

A signal is a notification to a process that an event has occurred, being sometimes described as a software interrupt. Signals are analogous to hardware interrupts in that they interrupt the normal flow of execution of a program, and in most cases, it is not possible to predict exactly when a signal will arrive. Signals can be employed as a synchronization technique, or even as a primitive form of interprocess communication (IPC), since one process can send a signal to another process (if it has suitable permissions). [29]

Each signal is defined as a unique integer, starting sequentially from 1. These integers are defined in `<signal.h>` and may vary across implementations. For that, are defined symbolic names in the form *SIGxxxx*. Also, each signal has a default action:

- *term*: terminates the process;
- *core*: process produces a core dump file and terminates;
- *ignore*: ignore the signal;
- *stop*: stops the process;
- *cont*: resumes a stopped process.

A signal is said to be generated by some event. Once generated, a signal is later delivered to a process, which then takes some action in response to the signal. Between the time it is generated and the time it is delivered, a signal is said to be pending.

In the list below are presented some common signals used in Linux.

- `SIGHUP`: hang up detected on controlling terminal or death of controlling process;
- `SIGINT`: issued if the user sends an interrupt signal `ctrl^C`;
- `SIGKILL`: if a process gets this signal it must quit immediately;
- `SIGALRM`: alarm clock signal (used for timers);
- `SIGTERM`: software termination signal.
- `SIGUSR1` and `SIGUSR2`: available for programmer-defined purposes; the kernel never generates these signals for a process.

Kill command

In Linux there is a command, `kill`, used to send a signal to a process, having as default signal the `TERM`.

- `-l`: list signal names;
- `-<signal>`: specify the signal to be sent;
- `<pid> [...]`: send signal to every PID listed.

With this in mind, one can use this command to send, for example, a `SIGTERM` to the process with PID `12` by executing: `kill -SIGTERM 12`.

5.4 Device Drivers

System memory in Linux can be divided into two distinct regions: kernel space and user space. Kernel space is where the kernel (the core of the operating system) executes and provides its services. User space is that set of memory locations in which user processes (everything other than the kernel) run. One of the roles of the kernel is to manage individual user processes within this space and to prevent them from interfering with each other. User processes can access kernel space only through the use of system calls, which are requests in a Unix-like operating system by an active process for a service performed by the kernel, such as input/output or process creation. [31]

A device driver is a set of functions and data, in the kernel, that make the interface with an external I/O device. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface, hiding completely the details of how the device works. Devices are represented by entries in the /dev directory, being considered regular files. Each device has a corresponding device driver, which implements a standard set of operations, including those corresponding to the *open()*, *read()*, *write()*, and *close()* system calls. [32]

User activities are performed by means of a set of standardized calls that are independent of the specific driver. So the role of the device driver is to map those calls to device-specific operations that act on real hardware. Linux kernel offers a set of functions to the user space for the interaction with hardware, as one can see in the table 5.1, in "User Functions". In the kernel space, Linux offers a set of functions that interact directly with the hardware and allows data transfer between kernel space and user space, as shown in "Kernel Functions".

Event	**User Functions**	**Kernel Functions**
Load module	insmod	module_init()
Remove module	rmmod	module_exit()
Open module	fopen()	file_operations: open
Read from device	fread()	file_operations: read
Write to device	fwrite()	file_operations: write
Close module	fclose()	file_operations: close

Table 5.1: Interface between an event, it's user function, and the kernel function called.

When the kernel recognizes that a certain action were requested to a device, it calls an appropriate function from the driver, and transfers the process control from the user to the driver function. After the driver function ends its execution, it gives the control back to the user space process.

To associate the normal files to the kernel mode, the Linux system uses the major number and the minor number. Major number identifies the driver associated to the device and is normally used by Linux system to map I/O requests to driver code. Minor number is dedicated to internal use and it is used by kernel to determine exactly which device was reference. To do the association, it should be created a file or node in /dev directory, (calling *mknod* as root user) that will be used to access the driver.

Linux supports three types of hardware devices: character, block and network device, being the character and block devices the most important. Character devices communicate directly with the user space program, so no buffer is required, for example the system's serial ports. Block devices are accessed by the user space program by a system buffer, and can only be written and read from in multiples of the block size, typically 512 bytes. [29]

5.5 Image Processing

Image processing consists of several techniques and methods used to manipulate images on a computer. The image processing helps improving the stored digital information, automate working with images and optimize image manipulation leading to efficient storage and transmission. Nowadays, image processing has many applications, like filtering images in editing apps and social media, medical technology, computer vision (objects identification, for example), pattern recognition, video processing and more. In this project, the image processing application is the computer vision, to detect available parking spots in the streets, using a camera.

In order to identify parking spots, one needs to find its outlines. The Canny Edge Detection [33] algorithm is a popular edge detection algorithm, and can be used to detect the parking spots outlines, providing an edge image. The canny edge detection is a multi-stage algorithm that goes through the stages: Noise reduction, Sobel Filtering, Non-maximum Suppression and Hysteresis Thresholding.

Noise reduction

The edge detection algorithm is sensible to noise in the image, so the first step is to remove the noise in the image using a 5x5 Gaussian filter. In figure 5.8 one can see the results of the gaussian filter 5x5.

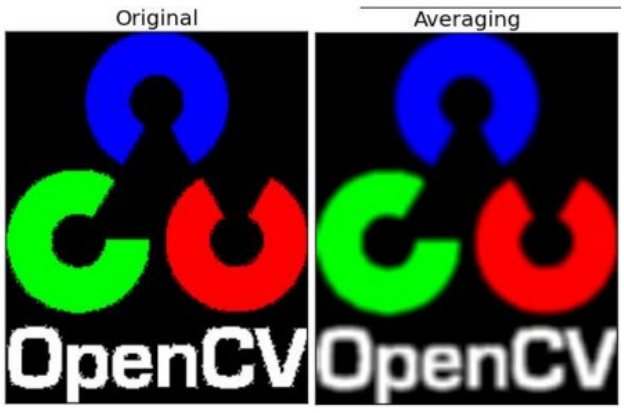


Figure 5.8: Gaussian Filter Results.

Sobel Filtering

Sobel filtering is an algorithm that finds intensity gradients on the image. This filter is applied in both horizontal and vertical direction, allowing to find the edge gradient and direction for each pixel. Figure 5.9, shows the sobel filtering results.

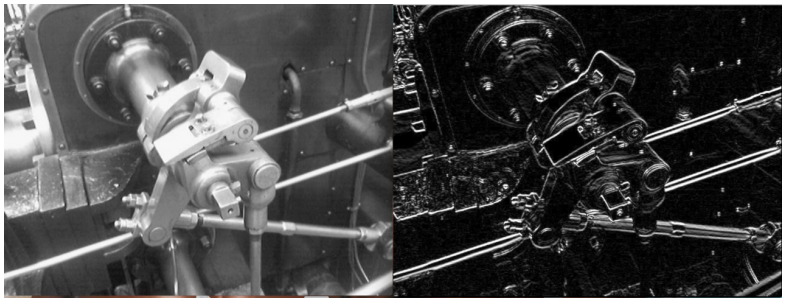


Figure 5.9: Sobel Filter Results.

Non-maximum Suppression

The non-maximum suppression algorithm does a full scan of the image in order to remove any unwanted pixels, that is, the pixels that don't constitute the edges of the image. In figure 5.10, one can see the non-maximum suppression algorithm. The point A is on the edge and the point B and C are in gradient directions. This algorithm suppresses the points B and C (put their pixel to 0) if they aren't a local maximum. If they are a local maximum, they are considered for the next stage as well as the point A.

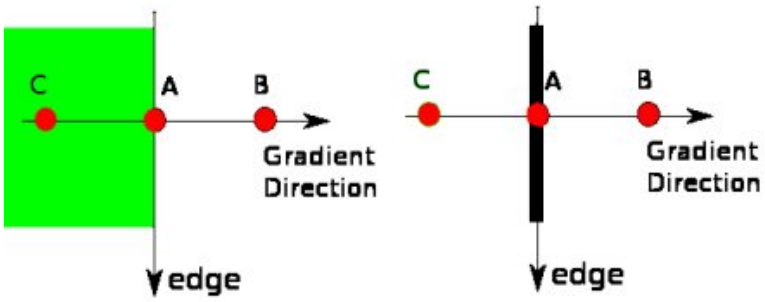


Figure 5.10: Non-maximum Suppression Algorithm.

Hysteresis Thresholding

This stage decides which edges considered until this point are really edges. This algorithm defines two threshold values, *minVal* and *maxVal*, in figure 5.11. The edges that have an intensity gradient higher than *maxVal* are considered edges and the edges that have lower intensity gradient than *minValue*, aren't considered edges. The points with intensity gradient between this values are considered edges when they are connected to other edges with intensity gradient higher than *maxVal* threshold value (the case of point C), and considered non-edges when they are not connected with other points with high intensity gradient value (the case of point B).

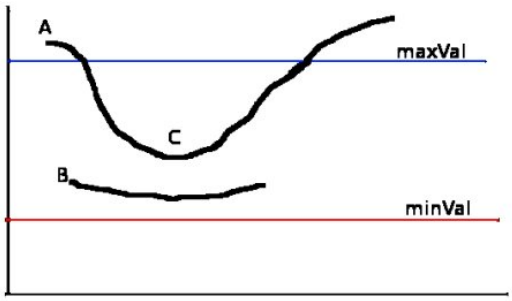


Figure 5.11: Hysteresis Thresholding Algorithm.

This algorithm, the Canny Edge Detection, allows the image processing algorithm to have only the strong edges of an image, being the final result represented in figure 5.12.

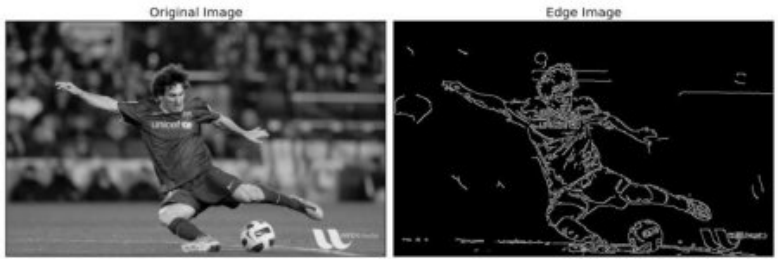


Figure 5.12: Canny Edge Detection Result.

Hough Line Transform

After detecting the edges of the image, one needs to take the processed image and use Hough Line Transform, which is an algorithm used to detect straight lines in an image. [34]

Knowing that a parking spot is delimited by straight line edges, this algorithm provides the extreme coordinates of the parking spot outline. With the parking spots coordinates, one can make a parking map and knows where there are parking spots to after identify if the spot is occupied or not.

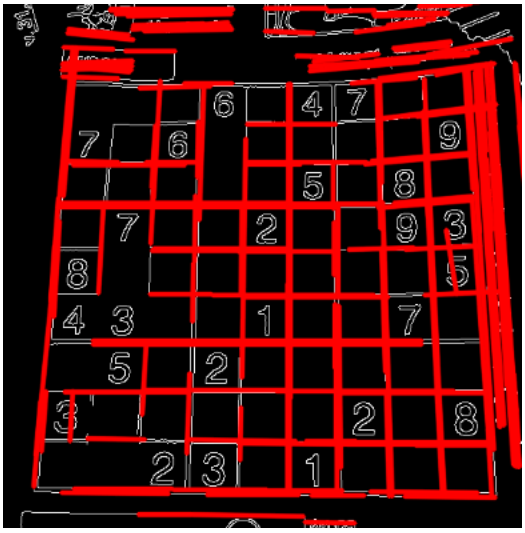


Figure 5.13: Hough Line Transform Result.

Parking Spots Occupancy

With the parking spots mapping, it is possible to identify the parking spots status. One can do that in three different ways:

- Check if the pixel colour of the spot aligns with the colour of an empty parking spot;
- Detect all cars and check if the car's location match with the parking spot coordinates;

- Take two pictures of the parking spots: one available and other occupied. The occupied and empty spots look very different, making it easy to identify the parking spot status.

Pixel Colour Change

To detect if a car is in the parking spot, one can see the change in the pixel colour of the spot. It can be done by calculating the spot pixel colours average and comparing it to the empty spot pixel colour average. If the average calculated is higher or lower than a pre-defined value, then the parking spot is occupied.

Cascade Classifier Training

The second way to identify the parking spot status is using a Cascade Classifier Training [35]. This is divided into two major stages: training and detection. For training it is needed a set of samples, negatives and positives. The negative samples correspond to non-object images and the positive samples correspond to images with detected objects. Then it is required to create a positive vector using an OpenCV utility that gathers all the positives. The last step is to train the cascade, creating a eXtensible Markup Language (XML) file that can be used to detect objects of a frame.

Chapter 6

Hardware Specification

6.1 Development Board

The development board for this project is the Raspberry Pi 4 Model B, shown in figure 6.1, considering it is one of the constraints identified in the analysis phase (3.1). This board includes a 64-bit quad-core ARM processor, the BCM2711, multimedia and connection features, resembling to a computer-like board that serves multiple applications. The following list shows the Raspberry Pi 4 Model B main features:

- 2GB LPDDR4-3200 SDRAM;
- 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE;
- Raspberry Pi standard 40 pin GPIO header;
- 2 USB 3.0 ports and 2 USB 2.0 ports;
- 2 micro-HDMI ports;
- 1 display port (2-lane MIPI DSI);
- 1 camera port (2-lane MIPI CSI);
- 1 jack 3,5 mm port (4-pole stereo audio and composite video port);
- graphic support (OpenGL ES 3.1, Vulkan 1.0);
- Micro-SD card slot.

The Raspberry Pi 4 Model B is used as a development board for this project, as it would not be used in a final application because not all its features are used. So, in a final application, the development board would be chosen, or designed, based on the features needed.

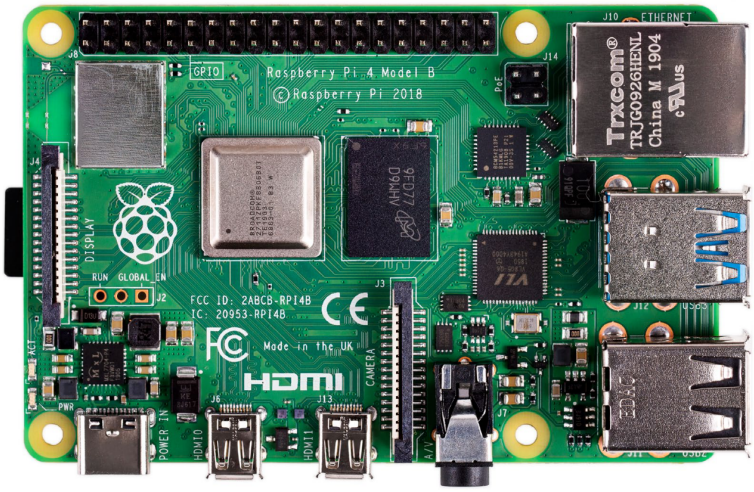


Figure 6.1: Raspberry Pi 4 Model B.

General Purpose Input/ Output (GPIO)

The Raspberry Pi 4 Model B board comes with a standard 40 pin GPIO header, that allows to interface with external peripherals. This GPIO also provides some interface technologies, like UART, I2C or SPI. The GPIO pinout of this board is shown in figure 6.2 [36].

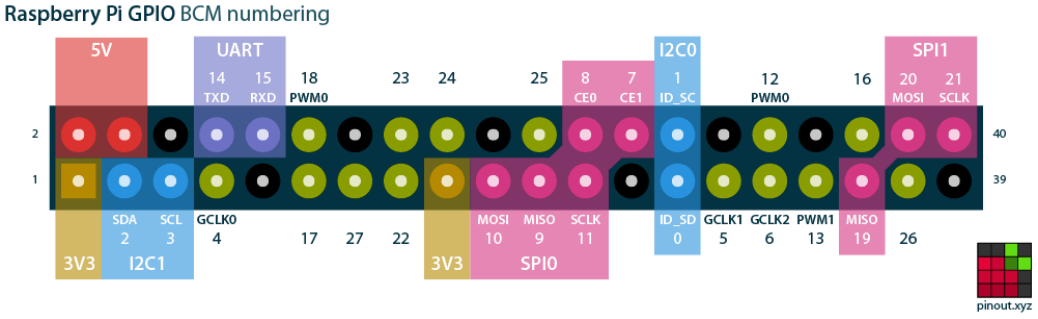


Figure 6.2: Raspberry Pi 4 Model B GPIO Pinout.

6.2 Luminosity Sensor

In order to know when is night time, that is, when the light conditions are low, one needs to determine the ambient light conditions. To do that, it is used a digital ambient light sensor, the TSL2581 [37], represented in 6.3. It was chosen this project because this sensor communicates with the Raspberry Pi using I2C communication protocol, so it gives a digital value of the luminosity, according to the light conditions, so it isn't necessary to calibrate this sensor (with a potenciometer, for example) when installing a new local system.

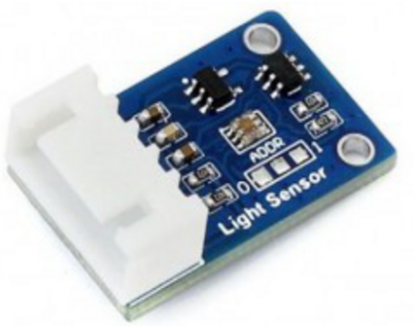


Figure 6.3: TSL2581 Light Sensor.

Characteristics

- I2C interface directly outputs the ambient light intensity value;
- High precision output, with an infrared photodiode;
- Allows connection to 3,3 V or 5,5 V MCU systems;
- 16-bit resolution.

TSL2581 Pinout	Description
VCC	Power in
GND	GND
SDA	I2C data signal
SCL	I2C clock signal
INT	Interrupt output pin

Table 6.1: TSL2581 Light Sensor Pinout Description.

Connection scheme

In table 6.2 it is shown the connection scheme of the Luminosity Sensor to the Raspberry Pi (remember figure 6.2). The figure 6.4 shows the connections layout between the Raspberry Pi and the TSL2581 sensor.

TSL2581 Pin	RPi Pin/ Connector	Notes
VCC	Step-down VOUT+	5 V
GND	Step-down VOUT- & GND	GND
SDA	Pin 2	I2C1-SDA
SCL	Pin 3	I2C1-SCL
INT	-	-

Table 6.2: Connection scheme: Luminosity Sensor.

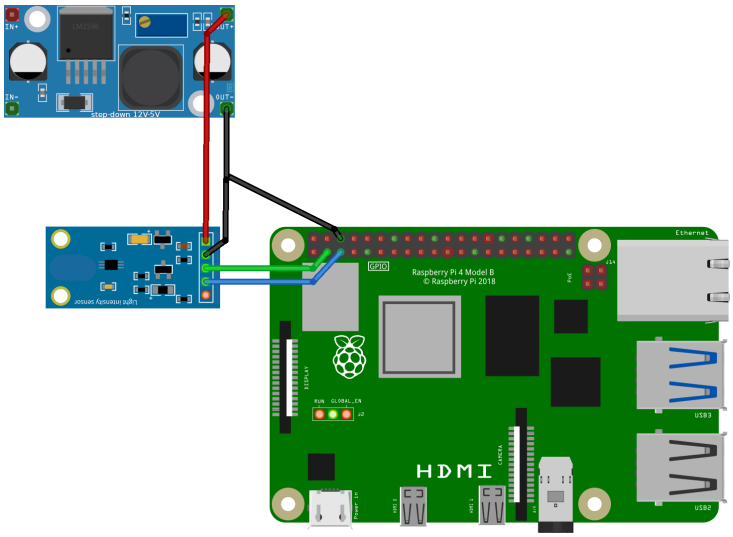


Figure 6.4: Connection Layout: TSL2581.

Test Cases

It is important to know how the light sensor works, and how it will behave upon certain events. In table 6.3 are shown test cases to this module.

Test Case	Expected Output	Real Output
Read ambient luminosity	Return ambient luminosity	-

Table 6.3: Test Cases: Luminosity Sensor.

6.3 Motion Detector

To know when to turn on the lamp, it is necessary to detect movement in the streets. For this project it is used a motion detector, more specifically a Passive Infrared (PIR) sensor. The chosen sensor for this purpose was the PIR HC-SR501, shown in figure 6.5. [38]

Figure 6.5: PIR HC-SR501.

Characteristics

- Supply voltage range of 4,5 - 20 V;
- Static current: 50 uA;
- Detection range of 7 meters;
- Detection angle of 110 degrees;
- Two potenciometers to adjust the trigger sensitivity and the delay of the trigger signal, between 0,3 seconds and 5 minutes;

PIR HC-SR501 Pinout	Description
VCC	Power in
GND	GND
SIGNAL	Output signal

Table 6.4: PIR HC-SR501 Pinout Description.

In the table 6.4 is shown the PIR HC-SR501 interface pins, being the output signal the pin SIGNAL. When no movement is detected, the sensor output is low (0,3 V), and when movement is detected, the sensor SIGNAL is high (3,3 V).

Connection scheme

In table 6.5 it is shown the connection scheme of the PIR sensor to the Raspberry Pi (remember figure 6.2). The figure 6.6 shows the connections layout between the Raspberry Pi and the PIR HC-SR501 sensor.

PIR HC-SR501 Pin	RPi Pin/ Connector	Notes
VCC	Step-down VOUT+	5 V
GND	Step-down VOUT- & GND	GND
SIGNAL	Pin 16	GPIO

Table 6.5: Connection scheme: Motion detector.

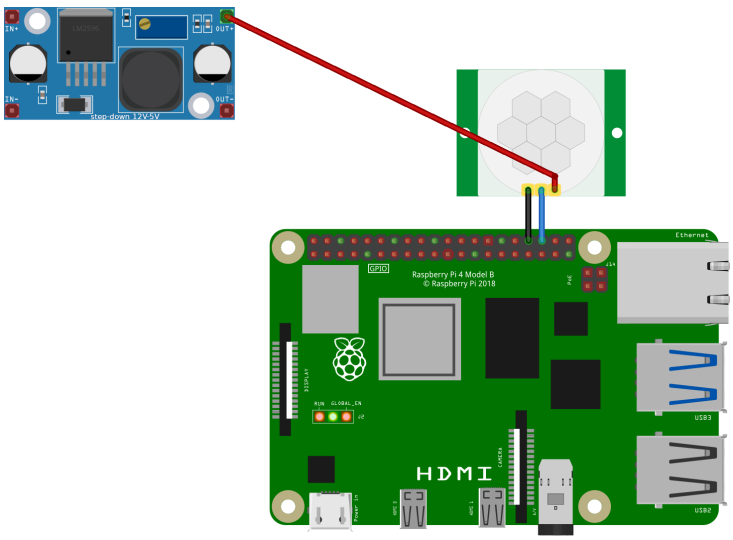


Figure 6.6: Connection Layout: PIR HC-501SR.

Test Cases

It is important to know how the motion detector module works, and how it will behave upon certain events. In table 6.6 are shown test cases to this module.

Test Case	**Expected Output**	**Real Output**
Movement in front of the sensor	Pin SIGNAL high (3,3 V)	-
No movement in front of the sensor	Pin SIGNAL low (0,3 V)	-

Table 6.6: Test Cases: Motion Detector.

6.4 Camera

In order to the detection of available parking spots, it's used a camera, as shown in figure 6.7. This is the Raspberry Pi Camera Module V1, capable of delivering a clear 5 MP resolution image, or 1080p HD video recording at 30fps. [39]

Figure 6.7: Raspberry Pi Camera Module V1.

Characteristics

- 5MP OmniVision 5647 sensor;
- Fixed focus lens onboard;
- Still Picture Resolution: 2592 x 1944;
- 15-pin ribbon cable, to the dedicated 15-pin MIPI CSI;
- Video recording: Supports 1080p @ 30fps, 720p @ 60fps.

Connection scheme

This device plugs directly into the CSI connector on the Raspberry Pi, as shown in figure 6.8, through the use of a 15-pin ribbon cable.

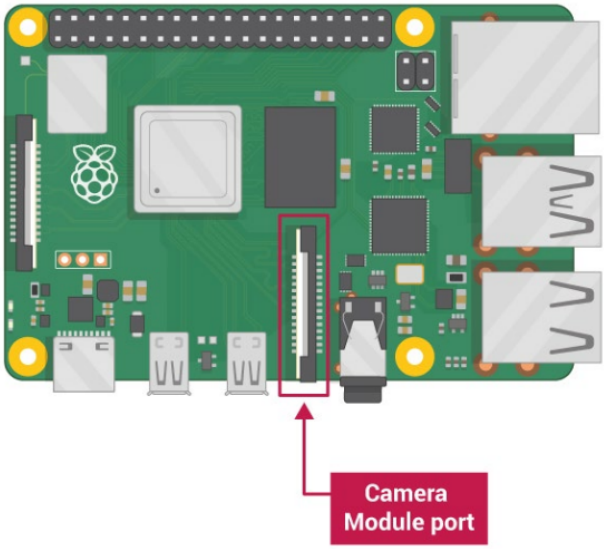


Figure 6.8: Connection scheme: Camera Module.

Test Cases

It is important to know how the camera module will behave upon certain events. In table 6.7 are shown test cases to this module.

Test Case	**Expected Output**	**Real Output**
Take picture	Clear output image	-

Table 6.7: Test Cases: Camera Module.

6.5 LoRa Module

To allow each local system to communicate to the gateway, and vice-versa, it's used LoRa communication technology, requiring for that LoRa Modules, as the one presented in figure 6.9. This module, from Ai-Thinker company, uses SX1278 Integrated Circuit (IC) from SEMTECH, and works on a 433 MHz frequency, with a range up to 10 km in line of sight. [40] [41]

Figure 6.9: LoRa Module SX1278 RA-02 433 MHz.

Characteristics

- Works on 433 MHz;
- Low RX current of 9,9 mA;
- Supply voltage: 1,8 V - 3,7 V;
- LoRa modulation technology (Supports also FSK, GFSK, MSK, GMSK and OOK modulation modes);
- Effective Bitrate : 0,018 - 37,5 kbps;
- Payload length: 64 bytes;
- Half-duplex SPI communication;

- Programmable bit rates up to 300kbps;
- Packet engine with CRC up to 256 bytes;
- Male U.FL connector to support using of external RF antenna - Diameter: 15,5mm.

SX1278 RA-02 Pinout	**Description**
VCC	Power in
GND	Ground
RST	Reset
SCK	SPI clock input
NSS	SPI selected-IN
MISO	SPI data output
MOSI	SPI data input
DIO0	Digital Input/Output Pin 0

Table 6.8: LoRa Module SX1289 RA-02 Pinout Description.

In order to achieve longer range and better signal quality in LoRa communication, an antenna may be used, as the one shown in figure 6.10. RF antennas play a critical role in diverting, directing or concentrating radio wave transmission in a particular direction.

Figure 6.10: RF Antenna 433 MHz.

Characteristics

- Frequency: 433,05 - 434,79 MHz;
- Antenna gain: 2 dBi;
- Connector I-PEX (U.FL) - Diameter: 15,5mm;
- Impedance 50 Ω.

Connection scheme

In table 6.9 it is shown the connection scheme of the LoRa module to the Raspberry Pi (remember figure 6.2). Keep in mind that the RF antenna is directly connected to the LoRa module through an U.FL connector. The figure 6.11 shows the connections layout between the Raspberry Pi and the LoRa SX1278 RA-02 transceiver.

SX1278 RA-02 Pin	**RPi Pin/ Connector**	**Notes**
VCC	Voltage Regulator OUT	3,3 V
GND	Voltage Regulator GND	GND
RST	Pin 4	GPIO
SCK	Pin 11	SPI0 SCLK
NSS	Pin 22	SPI0 CE0
MISO	Pin 9	SPI0 MISO
MOSI	Pin 10	SPI0 MOSI
DIO0	Pin 17	GPIO

Table 6.9: Connection scheme: LoRa Module.

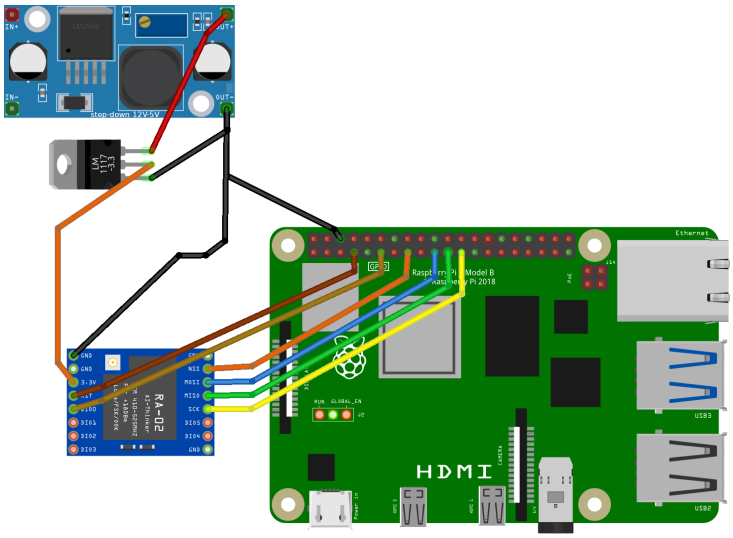


Figure 6.11: Connection Layout: LoRa SX1278 RA-02.

Test Cases

It is important to know how the LoRa module works, and how it will behave upon certain events. In table 6.10 are shown test cases to this module.

Test Case	Expected Output	Real Output
Establish connection	Stable connection established	-
Send data	Data sent correctly	-
Receive data	Data received correctly	-

Table 6.10: Test Cases: LoRa Module.

6.6 Power Module

As specified before, it's needed a power module to power the system, using the power grid. For that, it's necessary the use of an AC/DC power supply, in order to use the power grid electricity, which in Portugal is 230 V AC. The power supply must provide 12 V DC, as it is needed to power the lamp, as previously seen. The power supply used is shown in figure 6.12. [42]

Figure 6.12: ORNO Industrial Power Supply 12 V / 5 A.

Characteristics

- Input voltage: 100-240 V AC, 60 W;
- Output voltage: 12 V DC;
- Output current: 5 A;
- Protected against overload, short circuit, over-voltage and over-heating.

Power Supply Pinout	**Description**
L	Power grid phase 230 V AC
N	Power grid neutral
GND	GND
-V	Output -12 V DC
+V	Output +12 V DC

Table 6.11: Power Supply Pinout Description.

For educational purposes, in this project the power supply will be connected to a power plug, and not directly connected with eletrical grid wires, being for that necessary a plug to bare end wire, as the one shown in figure 6.13.

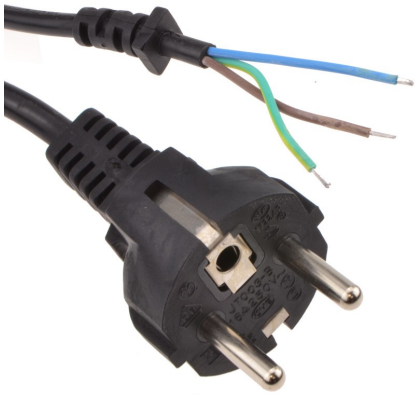


Figure 6.13: Plug to bare end wire 5 A.

The plug separates in three wires, which must be carefully connected to the power supply, as previously seen. These wires have a color pattern, as one shows in table 6.12.

Color pattern	**Description**
Brown wire	Power grid phase 230 V AC
Blue wire	Power grid neutral
Green/Yellow wire	GND

Table 6.12: Color pattern on eletrical wires.

In order to provide a lower voltage to power the Raspberry Pi and sensors, it is necessary to use a step down module, as the one shown in figure 6.14, which can convert 12 V DC to 5 V DC. [43]

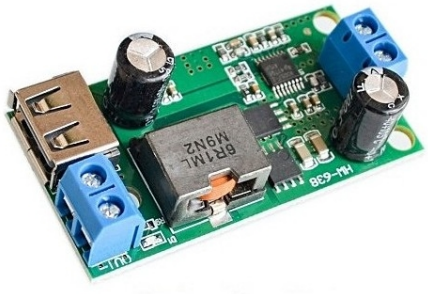


Figure 6.14: Step Down 9-38 V to 5 V / 5 A.

Characteristics

- Input voltage 9-38 V;
- Output voltage 5 V / 5 A;
- Load capacity: 5 A;
- Maximum efficiency of 95 %.

Step Down Pinout	Description
VCC	Input voltage 9-38 V
GND	Input GND
USB	USB output 5 V / 5 A *
VOUT+	Output 5 V / 5 A *
VOUT-	Output GND

* This module provides 5 A for both output pins combined (USB and VOUT+).

Table 6.13: Step-Down Pinout Description.

Connection scheme

In table 6.14 is shown the connection scheme for the step down. Keep in mind that the same power supply output 12 V DC is connected directly to the lamp VCC and to the step down VCC.

Step Down Pin	Connects to
VCC	Power supply Output +12 V DC
GND	Power supply Output GND
USB	Raspberry Pi USB-C
VOUT+	Sensors VCC
VOUT-	Sensors GND

Table 6.14: Connection scheme: Step Down.

In order to power the sensors that need a power supply of 3,3 V, one needs to use a voltage regulator, as the one shown in figure 6.15, that can convert 5 V DC to 3,3 V DC. [44]

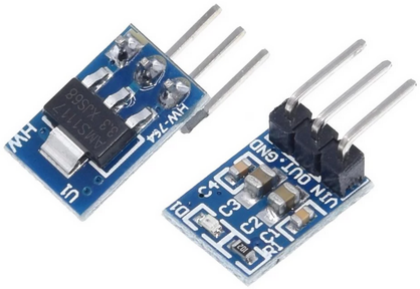


Figure 6.15: Voltage Regulator 5 V - 3,3 V / 800 mA.

Characteristics

- Input voltage 4,5 V - 7 V DC;
- Output voltage 3,3 V / 800 mA;
- Load regulation: 0.2 %.

Voltage Regulator Pinout	**Description**
VIN	Input voltage 4,5-7 V
GND	Input GND
OUT	Output voltage 3,3 V / 800 mA

Table 6.15: Voltage Regulator Pinout Description.

Connection scheme

In table 6.16 is shown the connection scheme for the voltage regulator.

Voltage Regulator Pin	**Connects to**
VIN	Step-Down Output 5 V DC
GND	Step-Down Output GND & Sensor GND Pin
OUT	Sensor Input Pin

Table 6.16: Connection scheme: Voltage Regulator.

Test Cases

It is important to know how the step down works, and how it will behave upon certain events. In table 6.17 are shown test cases to this module.

Test Case	Expected Output	Real Output
Connect power supply to the power grid	Provide 12 V DC	-
Connect step down to the power supply	Provide 5 V DC	-
Connect voltage regulator to the step-down	Provide 3,3 V DC	-

Table 6.17: Test Cases: Power module.

6.7 Driver

In order to control the lamp brightness through a GPIO pin from the Raspberry Pi, it's needed a driver circuit, as shown in figure 6.16. This circuit takes as input a Pulse Width Modulation (PWM) signal, provided by the raspberry Pi, and also the power for the lamp, V_{CL}, which comes directly from the power supply 12 V DC output of the power module. The output of this circuit, the lamp voltage V_L, is directly related to it's brightness, which one wants to control.

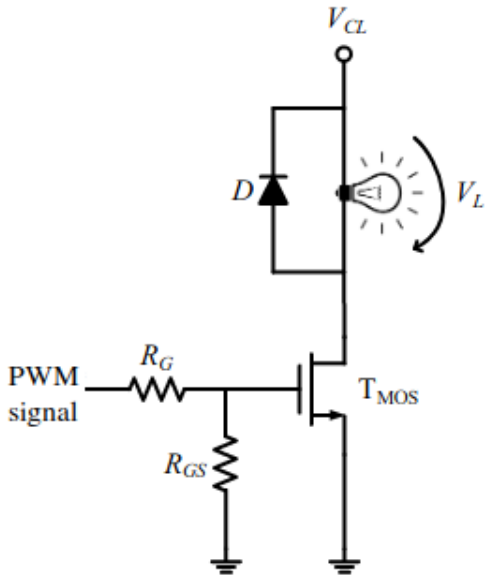


Figure 6.16: Driver circuit to control lamp brightness.

The PWM signal only takes two values, 0 or 3,3 V, but its average value over time is varied, changing at high frequency the time intervals in which the transistor T_{MOS} is conducting (applied voltage 3,3 V) and at cutoff (applied voltage 0 V). So, as the PWM duty cycle increases, the transistor conducts for more time, leading to an increase of V_L, and, therefore, to an increase of the lamp brightness. In any case, the power dissipation in the transistor is very low, because of the rapid change between each transistor state, conducting (at saturation) or at cutoff.

Characteristics

- Transistor MOSFET;
- Resistor R_G limits the current in the gate of the transistor;
- Resistor R_{GS} ensures that the gate is set to 0 potential when the circuit is turned off;
- Diode D, known as free-wheeling, provides a way to discharge the energy stored in the magnetic field of the inductive load, when the transistor turns off. For that reason, this diode must be fast;

In order to specify the components to be used, the following must be taken into account.

The temperature of the transistor (internal at the junction) must not exceed the maximum allowed by the manufacturer (typically 150 °C as in the case of STP60NF06, BUK453, BUZ90). In order for the transistor to be used without a heatsink, the internally dissipated power must not exceed 2 W (taking into account the typical thermal resistance of a TO-220 package, 60 °C/W and the ambient temperature of 30 °C). Considering that 50 % of losses occur in conduction and 50 % in switching (very vague approximation), the transistor can only dissipate 1 W in conduction and 1 W in switching. For an I_{DS} current of 2 A, the internal resistance (R_{DS} of MOSFET to V_{GS} voltage, I_{DS} current and operating temperature) will be a maximum of 250 mΩ (P = $R_{DS}.I_{DS2}$).

Diode D should be fast and the MR852 is recommended. In order to fulfill the above requirements, one can define R_G= 100 Ω, and R_{GS}= 12 kΩ

With this in mind, one can select the following components for the driver, as presented in table 6.18.

Driver Component	**Product Name**
T_{MOS}	BUK453
Diode D	MR852
Resistor R_G	100 Ω; +-5 %
Resistor R_{GS}	12 kΩ; +-5 %

Table 6.18: Driver components.

Connection scheme

In table 6.19 is shown the connection scheme for the driver circuit. (Remember figure 6.2 and table 6.11)

Driver Pin	**Connects to**
PWM Signal	Raspberry Pi Pin 12 - PWM0
V_{CL}	Power supply Output +12 V DC
GND	Power supply Output GND

Table 6.19: Connection scheme: Driver.

Test Cases

It is important to know how the driver works, and how it will behave upon certain events. In table 6.20 are shown test cases to this circuit.

Test Case	Expected Output	Real Output
Apply 3,3 V to the transistor	Lamp at maximum bright	-
Apply 0 V to the transistor	Lamp off	-
Change PWM signal	Variable lamp brightness	-

Table 6.20: Test Cases: Driver.

6.8 Lamp Failure Detector

In order to know when the lamp is broken, one needs to use a lamp failure detector. Since one simply wants to verify if the lamp is on when it should, one can implement the lamp failure through the use of an LDR sensor, as the one shown in figure 6.17. This should be placed pointed to the lamp, in order to detect the lamp brightness. [45]

Figure 6.17: Lamp Failure Detector - LDR.

Characteristics

- Sensitive type photoresistor sensor;

- Has an adjustable potentiometer to adjust the light intensity detected (threshold);
- Working voltage: 3,3 V - 5 V;
- Output format: Digital switching output (0 and 1);
- Output high when the ambient light intensity does not reach the threshold value;
- Output low when the ambient light level exceeds the set threshold;

Lamp Failure Detector Pinout	**Description**
VCC	Power in
GND	Ground
DO TTL	Output signal

Table 6.21: Lamp Failure Detector Pinout Description.

Connection scheme

In table 6.22 is shown the connection scheme for the lamp failure detector. (Remember figure 6.2 and table 6.11) The figure 6.18 shows the connections layout between the Raspberry Pi and the LDR failure detector.

Lamp Failure Detector Pin	**RPi Pin/ Connector**	**Notes**
VCC	Step-down VOUT+	5 V
GND	Step-down VOUT- & GND	GND
DO TTL	Pin 26	GPIO

Table 6.22: Connection scheme: Lamp Failure Detector.

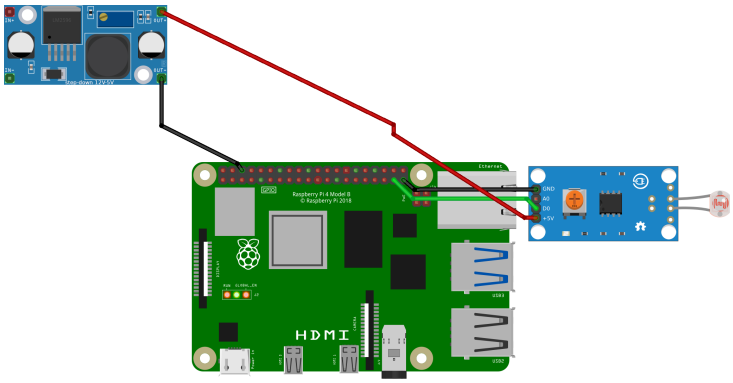


Figure 6.18: Connection Layout: Failure Detector LDR.

Test Cases

It is important to know how the lamp failure detector works, and how it will behave upon certain events. In table 6.23 are shown test cases to the lamp failure detector.

Test Case	Expected Output	Real Output
Place sensor near a turned on lamp	Output Low (0 V)	-
Place sensor near a turned off lamp	Output High (3,3 V)	-

Table 6.23: Test Cases: Lamp Failure Detector.

6.9 Lamp

In order to light the streets efficiently, one will use a LED lamp, that nowadays is the type of lamps with the better energy efficiency. It must be an adjustable lamp to control the lamp brightness, so the lamp selected was a G4 socket 12 V AC/DC LED lamp.

The lamp is controlled using a driver, that is approached in subsection 6.7 that takes a PWM signal (Pin 12) from the Raspberry Pi as input.

Figure 6.19: LED Lamp.

Characteristics

- Power supply: DC 12 V;
- Electric power: 2 W;
- Dimmable;
- Socket: G4;
- Chip controller: SMD3014.

In a real-scale project, it would be used a different lamp, with supply voltage of 230 V AC and other specific characteristics like Ingress Protection (IP) rate, from water and dust (IP65 [46]). It would be also used a LED lamp with higher electric power.

6.10 Connection Layout

In figure 6.20 is shown all the sensors connections with the power modules and the Raspberry Pi.

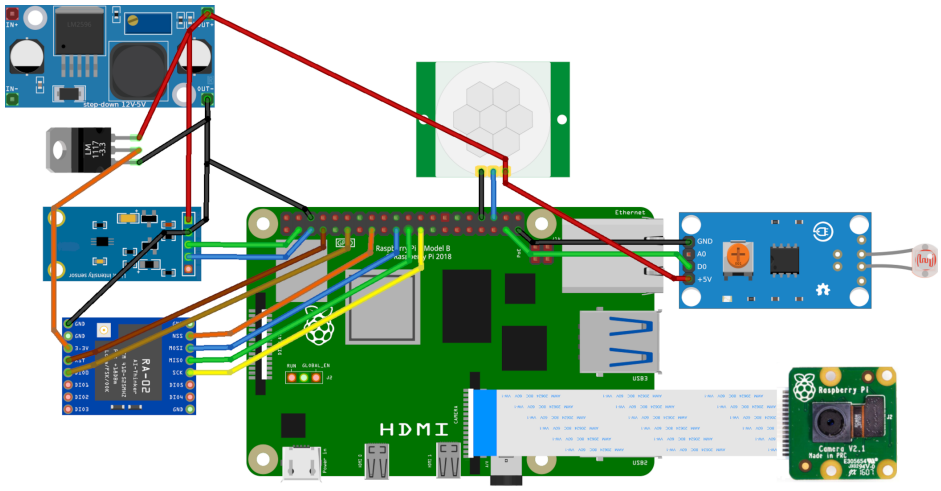


Figure 6.20: Connections Layout.

Chapter 7

Software Specification

7.1 Local System

One can define and describe briefly how the local system is implemented, making use of threads and processes. As one can see in figure 7.1, this system is composed by two processes: the main process and a daemon, *dSensors*, used to read the sensors *LDR*, *PIR* and *LampFailureDetector*. The communication between the daemon and the main process is done via message queue and through the use of the signal *SIGUSR1*, which the daemon can use to notify the main process of when there is a new message to read in the message queue.

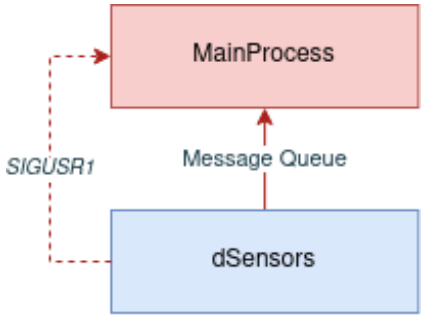


Figure 7.1: Inter-process Communication between Main Process and Daemon.

7.1.1 Class Diagrams

In figure 7.2 is represented the class diagram of the local system's main process. The class *CLocalSystem* is the main class of the main process, which initializes the objects of each class listed below.

- **CLoraComm:** manages the LoRa communications with the gateway, interfacing with the LoRa module;
- **CLamp:** manages the lamp brightness, using a PWM signal;
- **CCamera:** manages the camera device;
- **CParkDetection:** responsible of image processing for detecting parking spots.

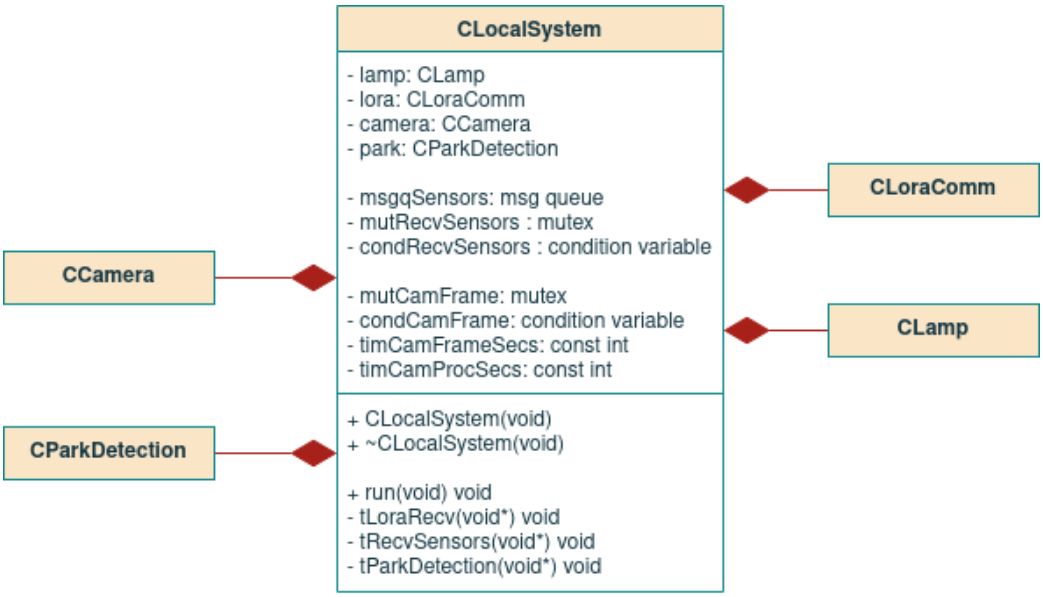


Figure 7.2: Local System Main Process Class Diagram.

In figure 7.3 is represented the class diagram of the local system's daemon. The class *CSensors* is the main class of dSensors, which initializes all objects of each class listed below.

- **CPir:** manages the motion detector, PIR;
- **CLdr:** manages the ambient light sensor, LDR;
- **CFailureDetector:** manages the lamp failure detector;

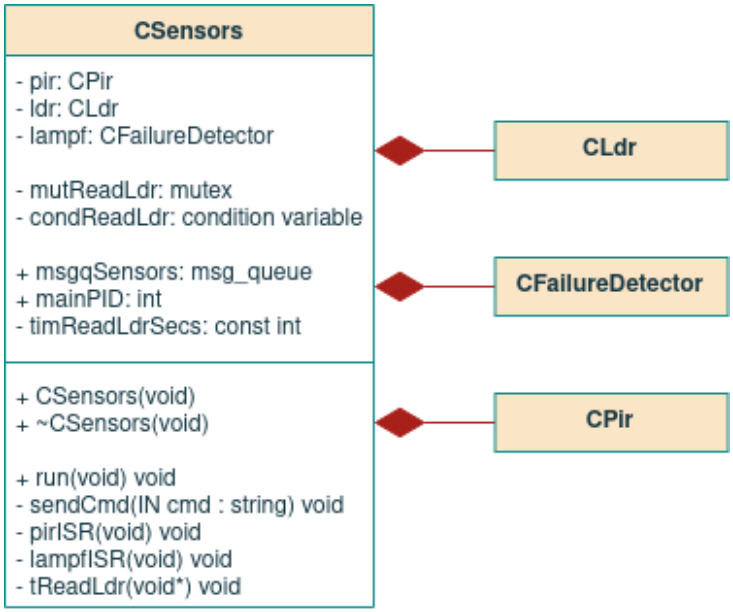


Figure 7.3: Local System dSensors Class Diagram.

Class CLoraComm

In figure 7.4 is shown the *CLoraComm* class diagram. This class defines an object *CLoraComm*, with the address *local_addr*, capable of establishing a LoRa communication at a defined frequency *freqMHz*, with the gateway which has the address *dest_addr*. This class inherits several methods from the class *CCommunication*, represented with a lighter font and identified with '^' at the begin of the method, which will be specified later.

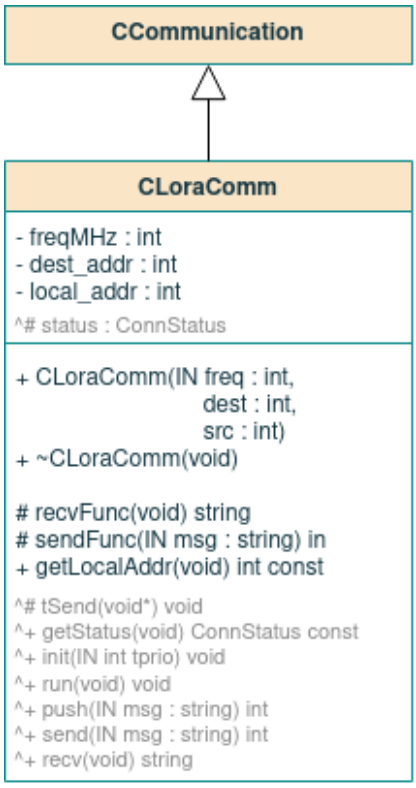


Figure 7.4: Class Diagram: CLoraComm.

Class CCommunication

In figure 7.5 is shown the *CCommunication* class diagram. This class implements the structure for a communication class like *CLoraComm* or *CTCP-client*, which will be shown later. It implements a series of methods to send and receive messages.

This class makes use of a thread, *tSend*, to send messages in non-blocking mode. This thread is created with *init(tprio)* method, which defines the thread priority and creates it. After creation, the thread goes to sleep, and is waken whenever the condition variable *condtSend* is notified, which occurs when *push(msg)* is used. This function adds the given message to a buffer, *TxMsgs*, which is later sent in *tSend*. This way, one has a waiting list of messages to be sent, in order to avoid the loss of a communication. The method *send(msg)* sends a message in blocking mode; *recv()* receives a message in non-blocking mode.

This class has two pure virtual functions: *recvFunc* and *sendFunc*, which must be implemented by the derived classes, since each communication protocol has its own functions to send and receive messages. Each communication has a state, *status*, defined by the enumeration connection status, *ConnStatus*.

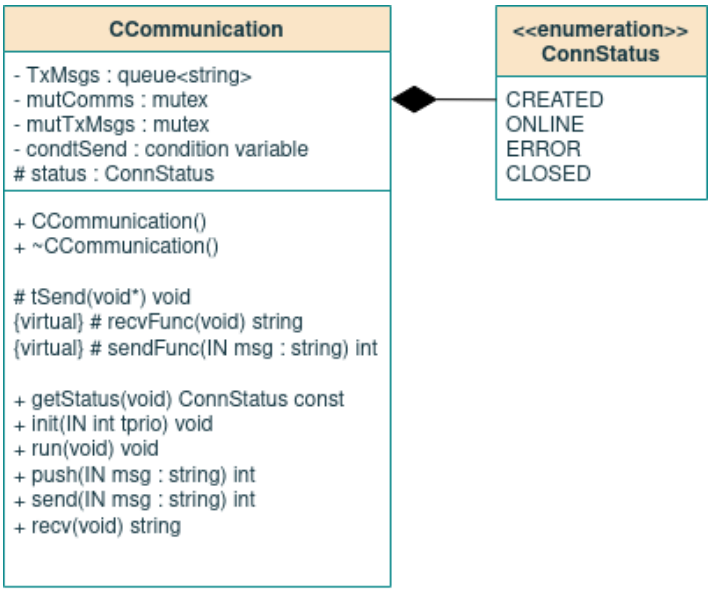


Figure 7.5: Class Diagram: CCommunication.

Class CLamp

In figure 7.6 is shown the *CLamp* class diagram, which defines a *CLamp* object. When creating the object, using the constructor *CLamp()*, one can define how much time the lamp stays at maximum brightness, passing this time to the constructor through parameter, *timeoutSecs*, in seconds, being defined into *LampOnTimeoutSecs*. One can interact with the object *CLamp* by changing the brightness, through the method *setBrightness(lux)*, where *lux* is a value from 0, where the lamp is OFF, to 100, where the lamp is at maximum brightness. Internally, this class uses a mutex, *mutChangePWM* to protect the PWM value when using *setBrightness* method.

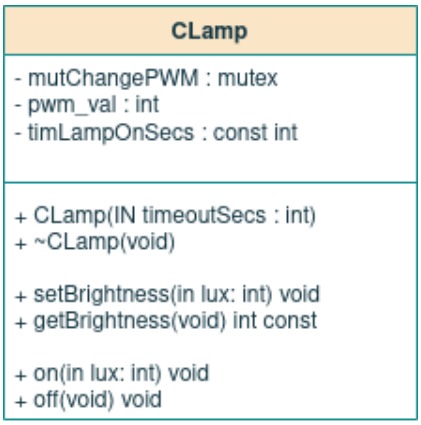


Figure 7.6: Class Diagram: CLamp.

Class CPir

In figure 7.7 is shown the *CPir* class. When movement is detected in the surrounding area of the lamppost, the sensor puts the high digital value in its output, triggering an interrupt service routine. When creating a object *CPir*, one can define the ISR to be executed, passing to the constructor a function pointer, *pirISR*. This class also has methods to enable and disable the respective ISR.

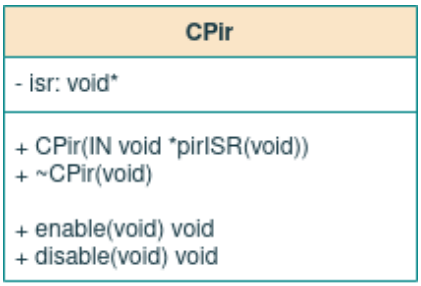


Figure 7.7: Class Diagram: CPir.

Class CCamera

In figure 7.8 is shown the *CCamera* class diagram, that defines a camera object. This class implements the control of a camera device, with filename *camFilename*, through the basic functions *open()*, *close()* and *capture()*, the latter used to get an image frame from the camera.

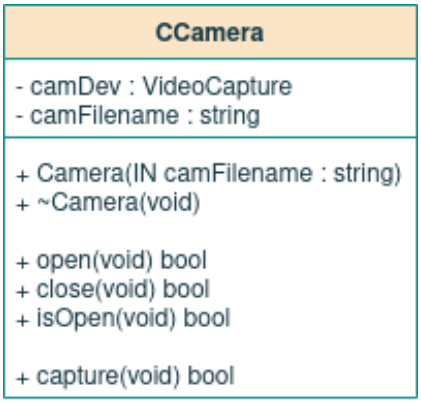


Figure 7.8: Class Diagram: CCamera.

Class CParkDetection

In figure 7.8 is shown the *CParkDetection* class diagram, that implements the tools needed to do process camera frames in order to do parking detection. Before determining if there are parking spots available, one needs to extract the park outline from the camera frame, using *getOutline(frame)*. This method stores in the variable *parkCoords* the coordinates on the image frame relating to the park outline. After this, one can calculate the number of vacants inside that park outline, through *calcVacants(frame)*.

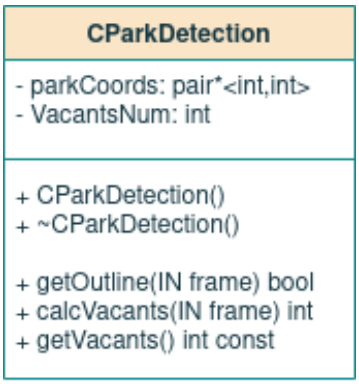


Figure 7.9: Class Diagram: CParkDetection.

Class CLdr

In figure 7.10 is shown the *CLdr* class, that defines the functions to interact with the LDR sensor. This contains an enumeration for the luminosity states, *LuxState*, which can be *DAY*, *NIGHT* or *UNDEF* as undefined, used for initialization. The luminosity state can be obtain by the use of *getLuxState()*.

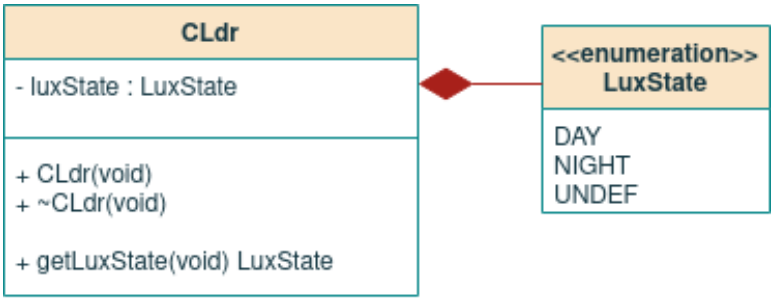


Figure 7.10: Class Diagram: CLdr.

Class CFailureDetector

In figure 7.11 is shown the Failure Detector class, which is similar to the *CPir*. When creating an instance of this class, using the constructor *CFailureDetector*, one can pass to the constructor a pointer to the ISR that will be triggered each time the failure detector senses that the lamp is off. Since one can verify if the lamp is broken only at night, that is when the lamp is turned on at minimum or maximum bright, one should *enable* this detector only when during the night.

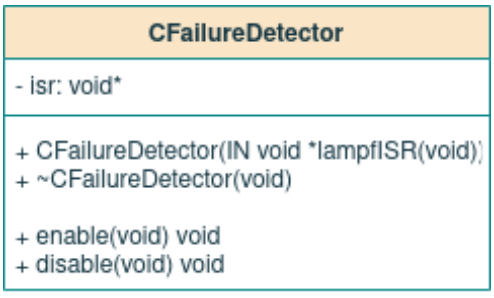


Figure 7.11: Class Diagram: CFailureDetector.

7.1.2 Task Overview

One can list the tasks that compose the main process:

- **CLoraComm::tSend:** sends a message to the gateway, using the LoRa module;
- **CLocalSystem::tParkDetection:** determines parking outline; acquire a camera frame, process it by verifying parking spots availability;
- **CLocalSystem::tLoraRecv:** receives a message from the gateway, using the LoRa module;
- **CLocalSystem::tRecvSensor:** receives messages sent by the daemon, via message queue, regarding sensors information. It's awaken by the use of signal *SIGUSR1*, sent by the daemon when there is a message to read.

Below are listed the tasks that compose the daemon, *dSensors*. As stated above, in *tRecvSensor*, whenever one of the following tasks sends a command to the main process, it is done by message queue, and also, a signal is used, *SIGUSR1*, to alert the main process that a new message was sent.

- **CSensors::tReadLdr:** periodically reads the LDR sensor; if a change in luminosity state is detected (*DAY* to *NIGHT*, or vice-versa) a command is sent to the main process;
- **CSensors::lampfISR:** ISR for the lamp failure detector; this is enabled only during the night, and is executed when the failure detector detects that the lamp is off, sending a command to the main process;
- **CSensors::PirISR:** ISR for the PIR sensor; this is enable only during the night and is executed when motion is detected, sending a command to the main process.

7.1.3 Task Priority

One needs to assign each thread a static priority level to indicate their relative urgency. Moreover, the scheduler will always pick the thread that is ready to execute with the highest priority level. Priorities must be assigned in order to ensure efficiency and execute real-time tasks.

With that in mind, the priority assignment diagram for the local system's main process is represented in figure 7.12, as well as for the daemon, in figure 7.13.

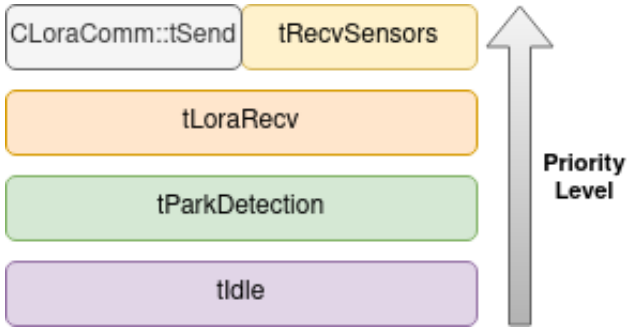


Figure 7.12: Local System Main Process Priority Assignment Schematic.

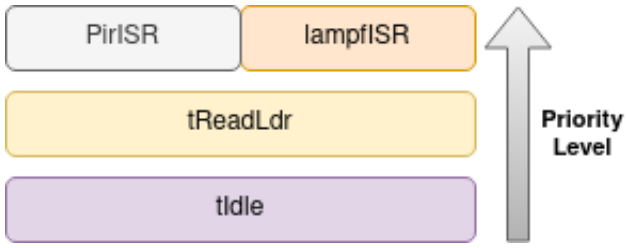


Figure 7.13: Local System dSensors Priority Assignment Schematic.

7.1.4 Task Synchronization

Real-time tasks share resources and services, and as such, should be prepared to await for the availability of these resources and services, like logical resources (buffers and data), physical resources, services like directory services, etc. In order to have coordinate access to shared resources and avoid race conditions, the kernel has resources that provide synchronization tools.

Condition Variables

A condition variable is a task synchronization tool that can be used to block (wait) one or more threads, suspending its execution. The blocked threads are awakened when the condition variable is notified.

The condition variables used in the main process are listed below.

- **CLocalSystem::condRecvSensors:** notifies *tRecvSensors* that a new message is available on the message queue, *msgqSensors*;
- **CLocalSystem::condCamFrame:** notifies *tParkDetection* to acquire a new camera frame and process it;
- **CCommunication::condtSend:** used to notify *CCommunication::tSend* that a new message is ready to be sent.

The condition variables used in the daemon are listed below.

- **CSensors::condReadLdr:** notifies *tReadLdr* to acquire values from LDR sensor and process them;

Mutexes

A mutex is a locking mechanism that provides mutual exclusion, supporting ownership and other protocols. A mutex is initially created in the unlocked state in which it can be acquired by a task. After being acquired, the mutex moves to the locked state. When the task releases the mutex, it returns to the unlocked state.

The mutexes used in the main process are listed below.

- **CLocalSystem::mutRecvSensors:** mutex associated with the condition variable *condRecvSensors* to read a new message from the sensors message queue;
- **CLocalSystem::mutCamFrame:** mutex associated with the condition variable *condCamFrame* to acquire a camera frame;
- **CCommunication::mutComms:** indirectly used by *CLoraComms*, to protect communication usage of send and receive functions;
- **CCommunication::mutTxMsgs:** indirectly used by *CLoraComms*; used to protect the insertion and removal of messages into *TxMsgs*;
- **CLamp::mutChangePWM:** protects the modification of PWM when defining a new PWM value for the lamp.

The mutexes used in the daemon are listed below.

- **CSensors::mutReadLdr:** protects reading of LDR sensor;

Signals

A signal is a notification to a process that an event has occurred. Signals are sometimes described as software interrupts, being analogous to hardware interrupts in that they interrupt the normal flow of execution of a program. One process can send a signal to another process, being in that way, employed as a synchronization technique.

The signals used between the main process and the daemon are listed below.

- **SIGUSR1:** signal sent by *dSensors* to the main process, whenever a new message is inserted in the message queue, *msgqSensors*. This is received in the main process in a signal handler, that will wake up the task *tRecvSensors*, through the condition variable *condRecvSensors*.

7.1.5 Task Communication

Message Queue

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. A message queue, *msgqSensors*, will be used to communicate between the main process and the *dSensors*. In that way, the main process is agnostic to the cyclic reading necessary for the sensor *ldr* and for the interrupts generated by the *pir* and *lampf*, being only informed when necessary, through the message queue.

7.1.6 Commands and Constants

Constants

The constants used in the local system are listed below.

- `LS_ADDR`: defines the local system address, relevant to the communications module, CLoraComm;
- `GATEWAY_ADDR`: defines the gateway address, relevant to CLoraComm;
- `MSGQ_NAME`: defines the name of the message queue, *msgqSensors*.
- `MIN_BRIGHT_PWM`: defines the minimum brightness of the lamp;
- `TIM_LAMP_ON_SECS`: defines the time from which the lamp stays on, at full brightness, in seconds;
- `TIM_READ_LDR_SECS`: defines the sampling period for the LDR sensor, in seconds;
- `TIM_CAM_FRAME_SECS`: defines the sampling period for the Camera frame capture, in seconds;
- `TIM_CAM_PROC_SECS`: defines the maximum time expected for the duration of image processing, in seconds;

Commands

The commands used for the communication between the daemon *dSensors* and the main process are listed below.

- `ON`: Set the lamp brightness to its maximum (`PWM=100`);
- `MIN`: Set the lamp brightness to its minimum (`PWM=MIN_BRIGHT_PWM`);
- `OFF`: Turn off the lamp (`PWM=0`);
- `FAIL`: Turn off the lamp (`PWM=0`);

The commands used for the communication between the local system and the remote system are listed below.

- `LAMP ON`: Lamp is at full brightness;
- `LAMP MIN`: Lamp is at minimum brightness;
- `LAMP OFF`: Lamp is off;
- `LAMP FAIL`: Lamp is broken (failure detected);
- `PARK <num>`: Parking has now `<num>` available parking spots.
- `ID <local_addr>`: Identify local system to the rest of the network, presenting its local address;
- `CRQ <local_addr>`: Connection request to the gateway;

7.1.7 Start-Up Process

Main Process

The start-up for the main process is shown in figure 7.14. Firstly, a *CLocalSystem* object is created, initializing all objects, that will be detailed later. After that, *run()* method from *CLocalSystem* is used, which will wait for the termination of all tasks.

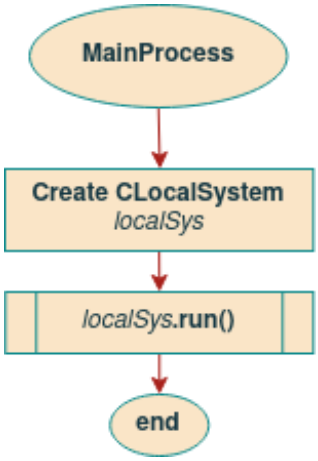


Figure 7.14: Start-Up Process: Main Process.

dSensors

The start-up for the daemon is shown in figure 7.15. Firstly, the process becomes a daemon, being that represented by *daemonize()*. After that, a message queue is opened in order to communicate with the main process. Then, it will wait until it receives a message in the message queue, that will be the main process PID. After this setups, a *CSensors* object is created, and the *run()* method is used, working similarly to the one presented in the main process start-up process.

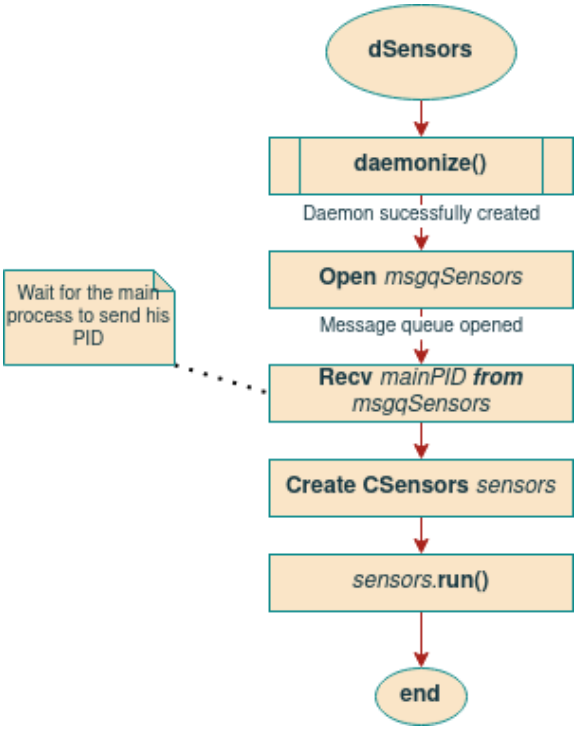


Figure 7.15: Start-Up Process: dSensors.

7.1.8 Flowcharts

CLocalSystem Methods

The class constructor is shown in figure 7.16. This is responsible for initializing all synchronization tools and private variables used, as well as creating the tasks *tLoraRecv*, *tRecvSensors* and *tParkDetection*.

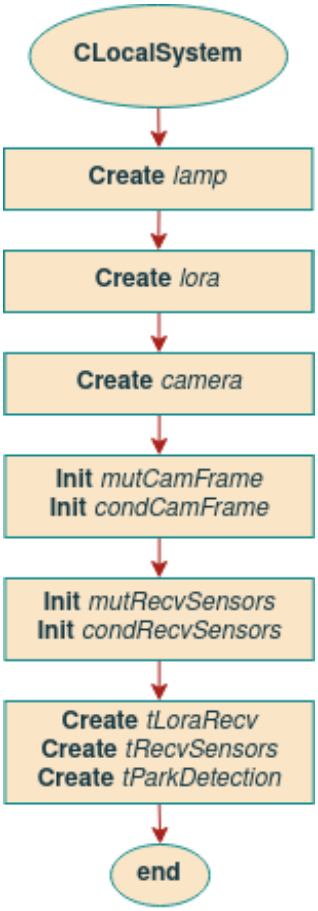


Figure 7.16: Flowchart: CLocalSystem constructor.

The method *run()*, presented in figure 7.17, it's implemented similarly in various classes. This is responsible for starting timers that trigger the execution of tasks, and wait (*join*) for the termination of the tasks.

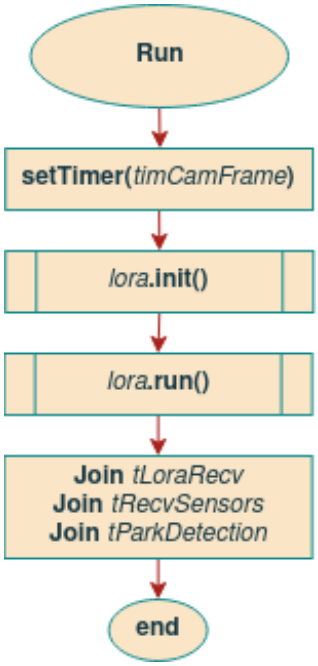


Figure 7.17: Flowchart: CLocalSystem Run method.

In figure 7.18 is presented the task responsible for receiving a message from the gateway, via LoRa communication, parse it and execute the respective command.

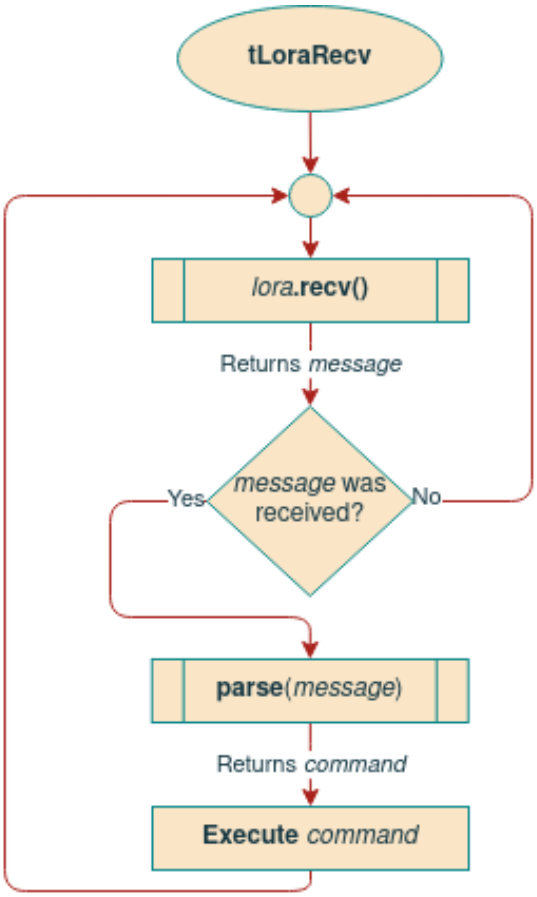


Figure 7.18: Flowchart: CLocalSystem tLoraRecv method.

In figure 7.19 is presented the task responsible for receiving messages from the sensors daemon, via message queue. When there are no messages to read from the message queue, the task goes to sleep and is awaken when *condRecvSensors* is notified. This happens in the *sigHandler*, presented in figure 7.20, which is the signal handler for the main process, being this in charge of signaling the condition variable *condRecvSensors* when a *SIGUSR1* signal is catched.

After the condition variable is notified, the message is read from the message queue and a command is sent do the remote system, to inform of the occurrence. This command begins with *"LAMP"*, indicating that the command is relative to the lamp state, and has an argument that is relative to the event that occurred in the sensors (*ON*, *MIN*, *OFF*, *FAIL*). After communicating to the remote system, one needs to set the appropriate PWM value, therefore, the lamp brightness, according to the received command from the sensors. For example, if the received command from the sensors was *"OFF"*, then the PWM must be 0.

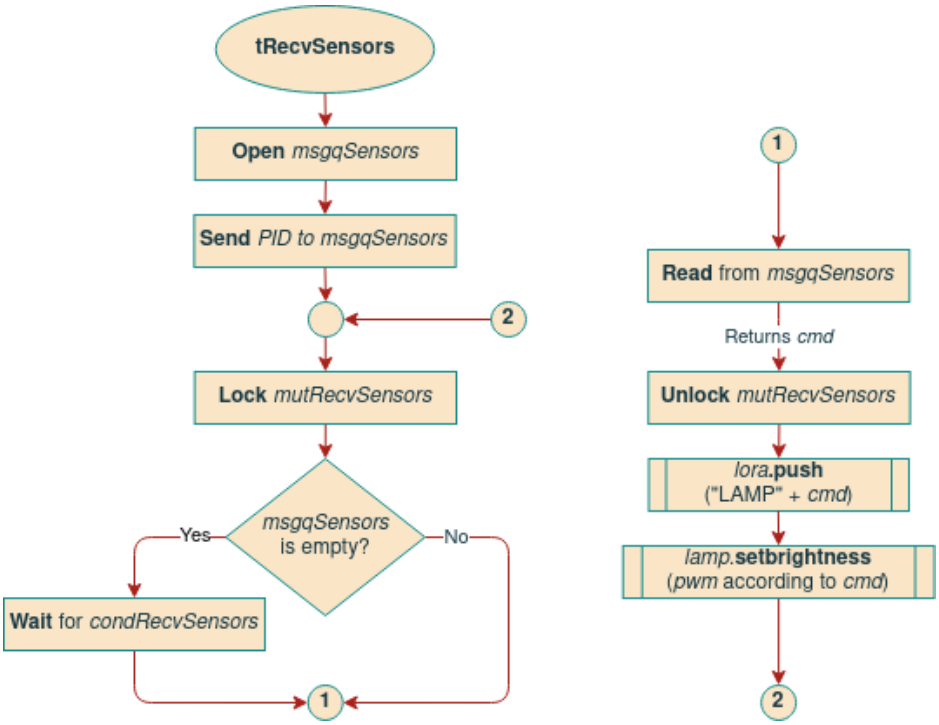


Figure 7.19: Flowchart: CLocalSystem tRecvSensors method.

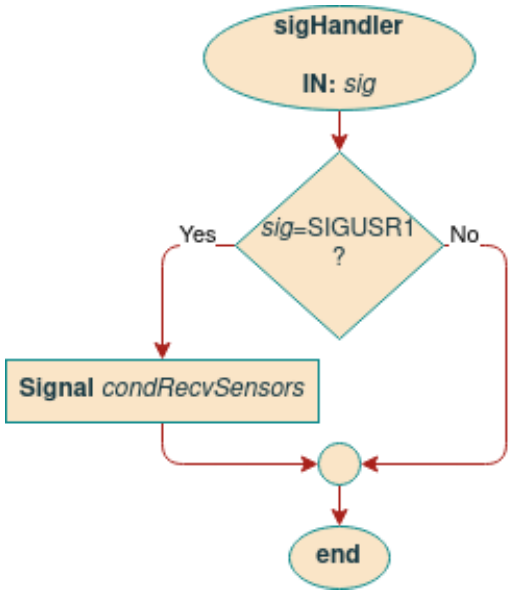


Figure 7.20: Flowchart: CLocalSystem sigHandler method.

This task, figure 7.21, is responsible for using the *camera* object, from *CCamera*, and the *park* object, from *CParkDetection*, in order to capture image frames and process them, calculating the number of available spaces in the parking detected. This task is periodically awaken by the use of a timer, signaling the condition variable *condCamFrame*. Besides that, a timer can be used, *timCamProc*, to detect if the image processing does not exceed a maximum amount of time.

When a different number of vacants is determined, that is communicated to the remote system using the command *"PARK"*, following by the number of vacants number.

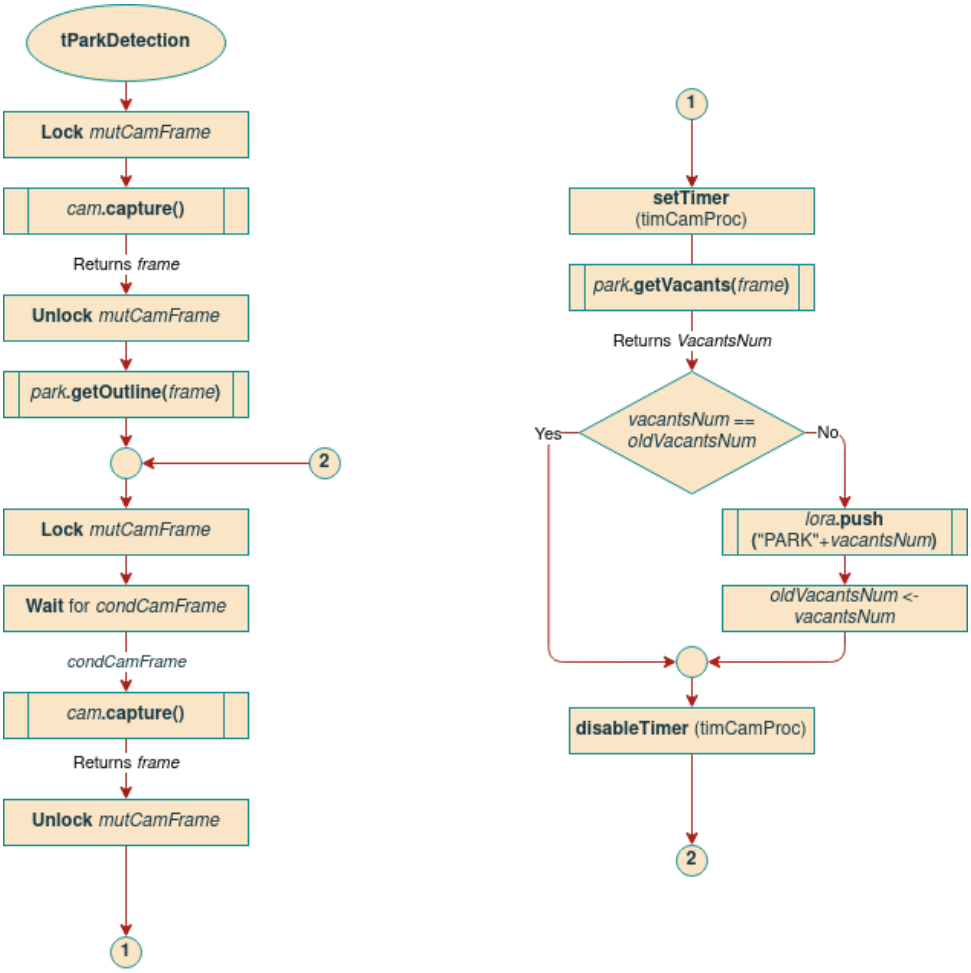


Figure 7.21: Flowchart: CLocalSystem tParkDetection method.

CSensors Methods

As in the previous class, the class constructor and the *run()* method serves similar purposes, therefore, one has no need to specify them.

In figure 7.22 is shown the task responsible for reading luminosity values using the LDR sensor. If there is a change in the luminosity state, to the state NIGHT, this task enables the lamp failure detector, *lampf*, and the motion detector, *pir*, and then, a command is sent to the main process, using *sendCmd(cmd)* method, indicating that the lamp should be turned on at minimum bright. If the change in the luminosity state is to the state DAY, the opposite occurs, disabling the detectors, and sending a command to the main process, indicating that the lamp should be off.

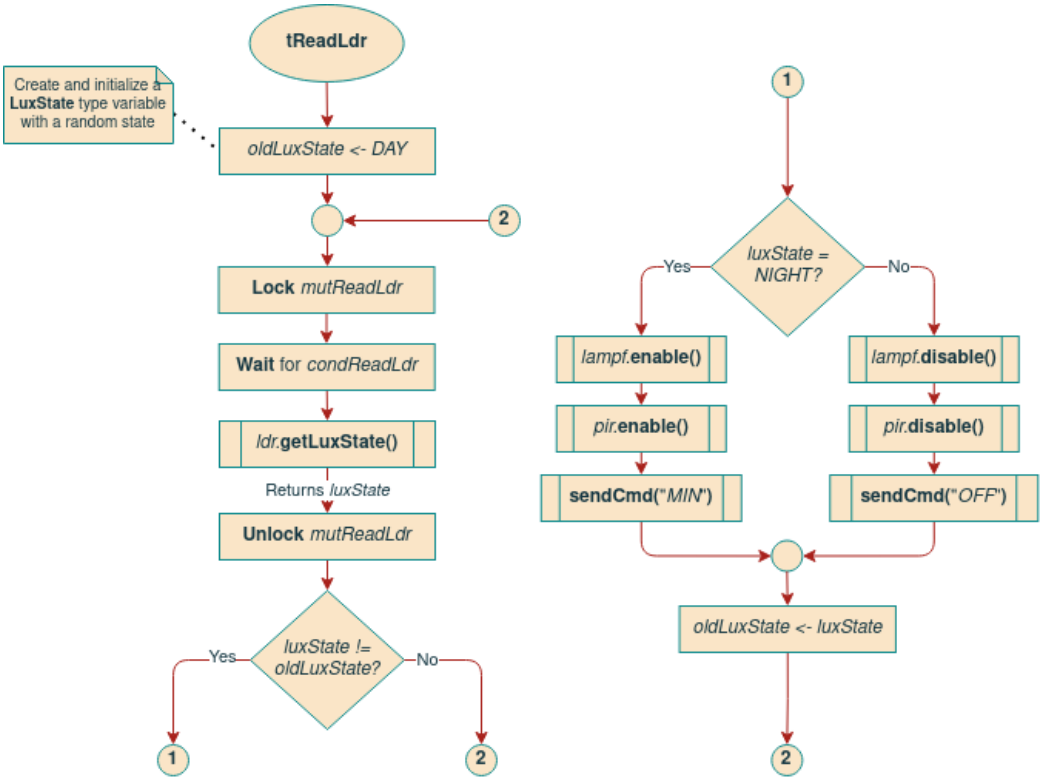


Figure 7.22: Flowchart: CSensors tReadLdr method.

The method *PirISR*, is executed when there is motion detected, sending the *"ON"* command to the main process, indicating that the lamp should be at maximum bright.

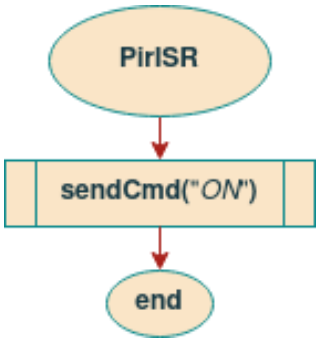


Figure 7.23: Flowchart: CSensors PirISR method.

The method *lampfISR* is very similar to the *PirISR*, since it is executed when the lamp failure detector detects that the lamp is off, when it shouldn't. This sends a command to the main process, *"FAIL"*, indicating that there was detected a failure in the lamp and that it should be turned off.

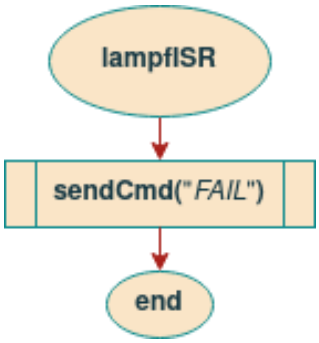


Figure 7.24: Flowchart: CSensors lampfISR method.

The method *sendCmd*, shown in figure 7.25, is responsible for sending a string, *cmd*, to the message queue, *msgqSensors*. After that, it sends a signal, *SIGUSR1*, to the process *mainPID*, which is the main process PID.

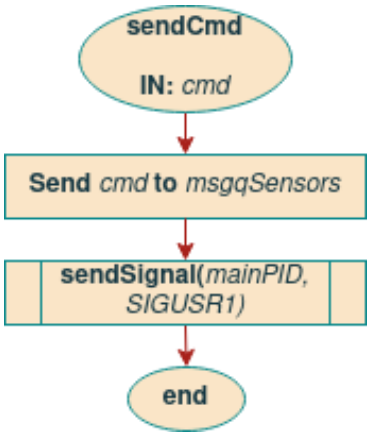


Figure 7.25: Flowchart: CSensors sendCmd method.

CCommunication Methods

The class constructor is shown in figure 7.26. This is responsible for initializing all synchronization tools and private variables used.

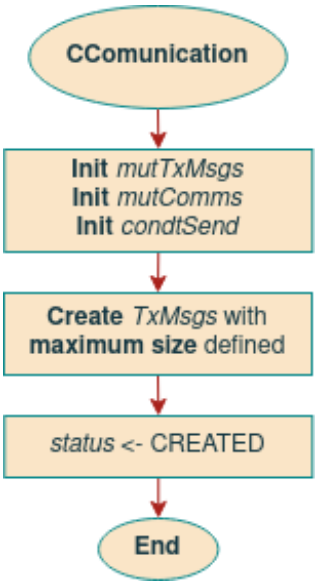


Figure 7.26: Flowchart: CCommunication constructor.

To send a message, this class has a built-in thread, *tSend* that can be created using *init(tprio)*, which expects a priority for the thread, *tprio*, as shown in figure 7.27.

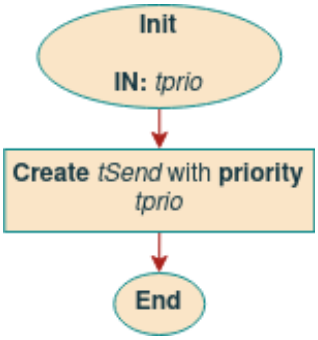


Figure 7.27: Flowchart: CCommunication Init method.

A message can be put in the waiting list to be sent through the use of *push(msg)*, as shown in figure 7.28. This function is responsible for adding a new message, *msg*, to the *TxMsgs* vector, and signal the condition variable *condSend*, for the thread *tSend* to send the message.

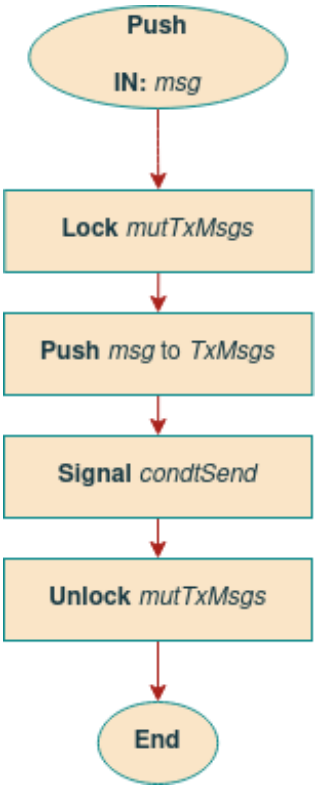


Figure 7.28: Flowchart: CCommunication Push method.

This thread, *tSend* presented in figure 7.29, is responsible for sending queued messages to the gateway, using LoRa communication. A conditional variable is used to wake this thread when there is a new message available to be sent. When the queue *TxMsgs* is empty, the task goes to sleep, waiting for the condition variable *condSend* to notify this task. After this, the mutex *mutComms* is used to protect the communication. Then, a message is popped from the messages queue, and sent to the gateway using the method *Send*, shown in figure 7.30. This continues to happen until the *TxMsgs* queue gets empty.

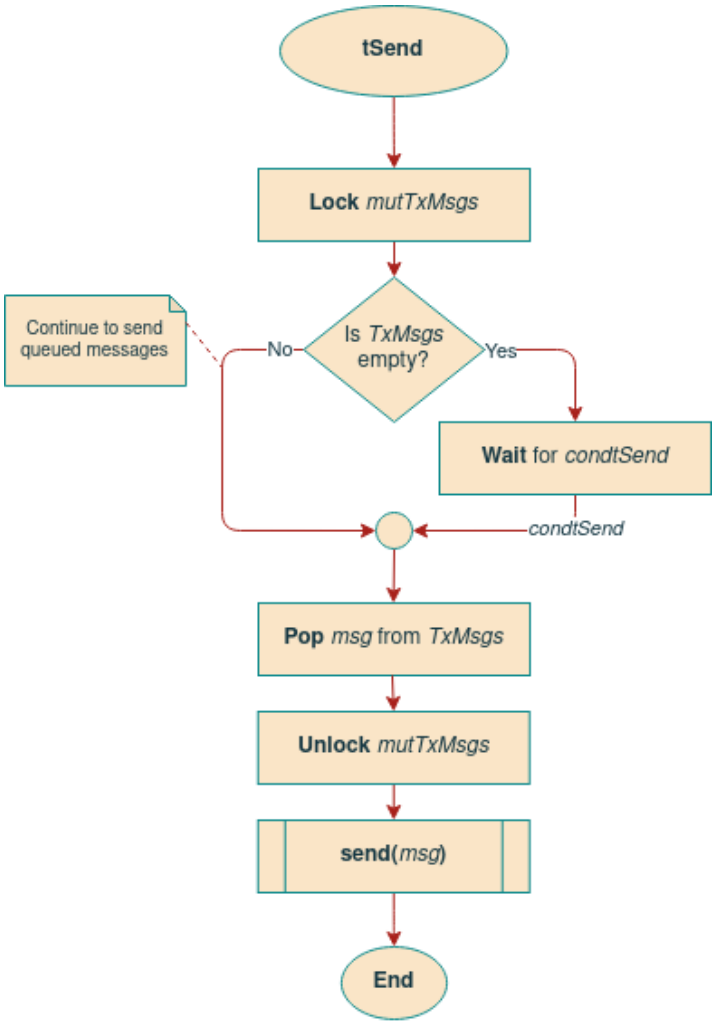


Figure 7.29: Flowchart: CCommunication tSend thread.

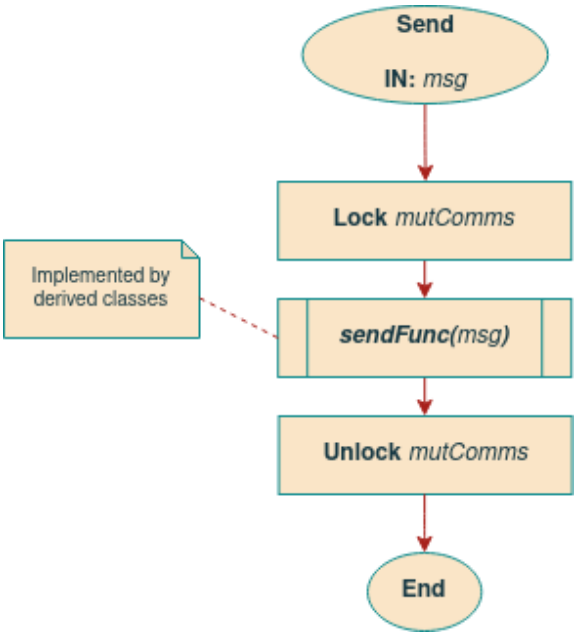


Figure 7.30: Flowchart: CCommunication Send method.

There is also another method, *recv()*, that receives a message from the gateway, in a non blocking mode, making use of task synchronization tools to ensure that sending and receiving don't occur at the same time. This function should be used continuously if one doesn't want to miss any communication. This method is presented in figure 7.31.

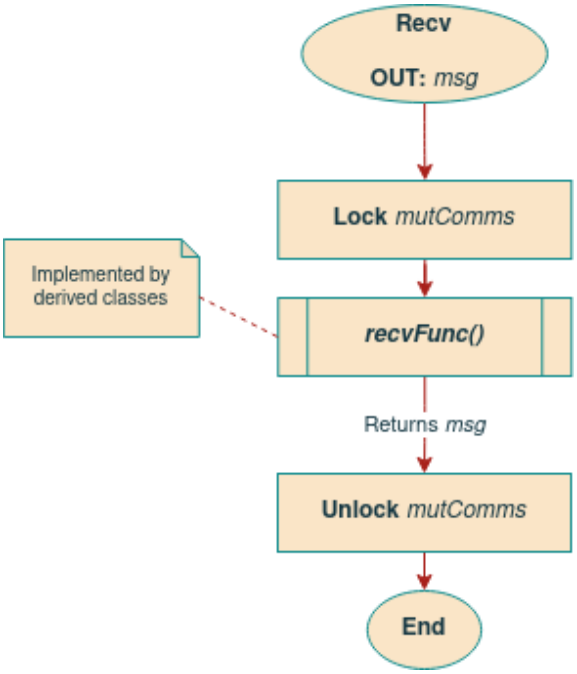


Figure 7.31: Flowchart: CCommunication Recv method.

CLoraComm Methods

A CLoraComm object can be created through the use of the constructor, as shown in figure 7.32. This starts by initializing the LoRa communication, using the given frequency, *freq*, which will be 433 MHz, and defining the pins connected to the module. At the end, all private members are also initialized.

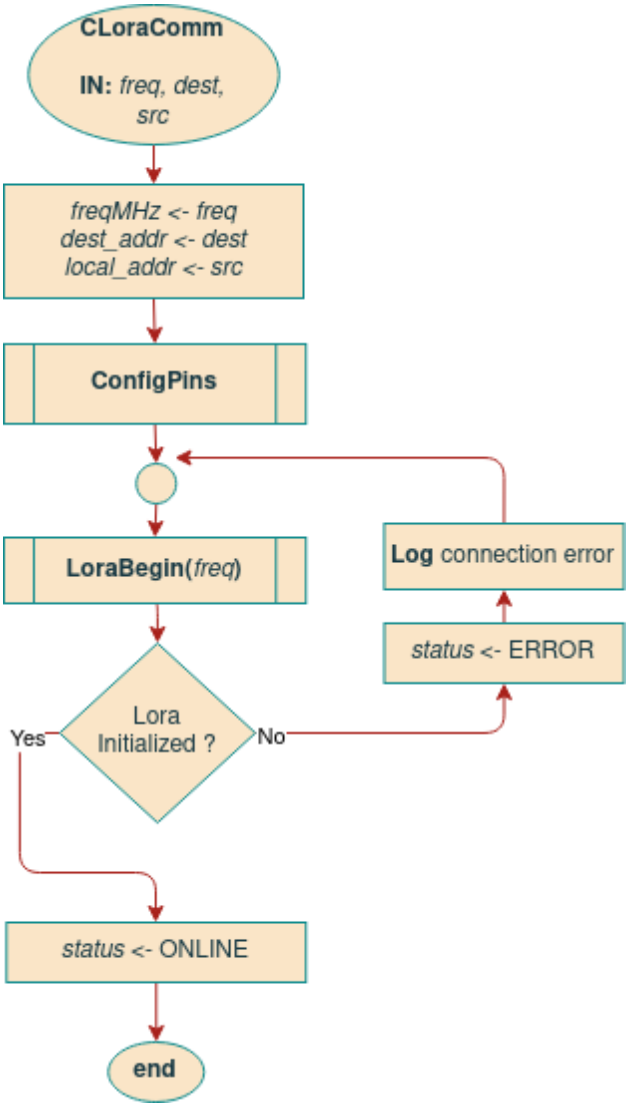


Figure 7.32: Flowchart: CLoraComm constructor.

This class inherits from *CCommunication* class, so this must implement *recvFunc* and *sendFunc*, as they are pure virtual methods. In this class, one will use existent functions *LoraReceive* and *LoraSend* to implement Lora communication, as shown in figures 7.33 and 7.34. This functions make use of the address of the local system, *local_addr*, defining the source ID, and the address of the gateway, *dest_addr*, defining the destination ID.

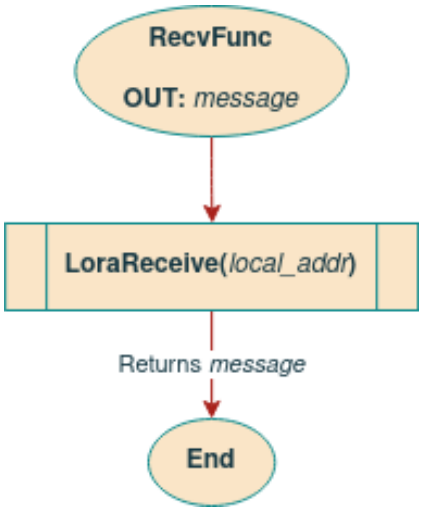


Figure 7.33: Flowchart: CLoraComm recvFunc method.

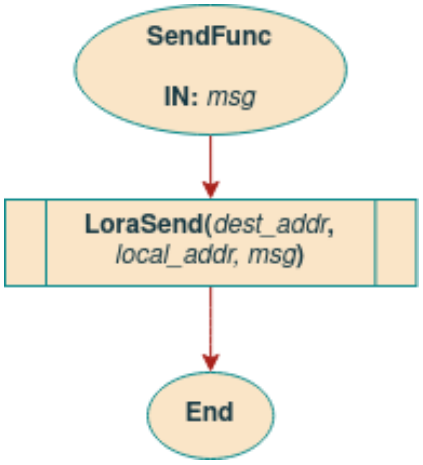


Figure 7.34: Flowchart: CLoraComm sendFunc method.

CLamp Methods

A CLamp object can be created through the class constructor, presented in figure 7.35. When creating an object, one must pass to the constructor the time from which the lamp will stay on after one sets the brightness to the maximum. This value must be in seconds and is stored in the variable *timLampOnSecs*.

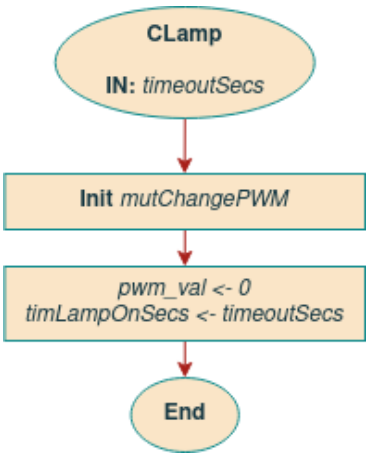


Figure 7.35: Flowchart: CLamp constructor.

This functions, presented in figure 7.36, are responsible for enabling and disabling the PWM for the lamp, which is responsible for controlling the lamp brightness. When turning on the lamp PWM, one can define the lamp brightness level, through *lux* argument.

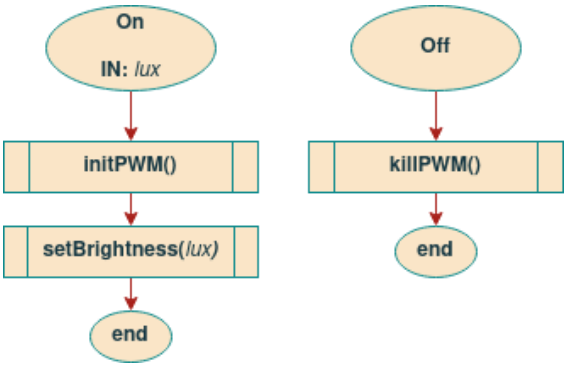


Figure 7.36: Flowchart: CLamp On and Off methods.

This function, presented in figure 7.37, is responsible for changing the PWM associated with the lamp, which is directly related to its brightness. Through *setPWM(lux)* one can change the applied lamp PWM to *lux* value, being this an integer between 0 to 100. When the PWM is maximum, a timer is started that defines how much time the lamp is ON, *timLampOnSecs* in seconds.

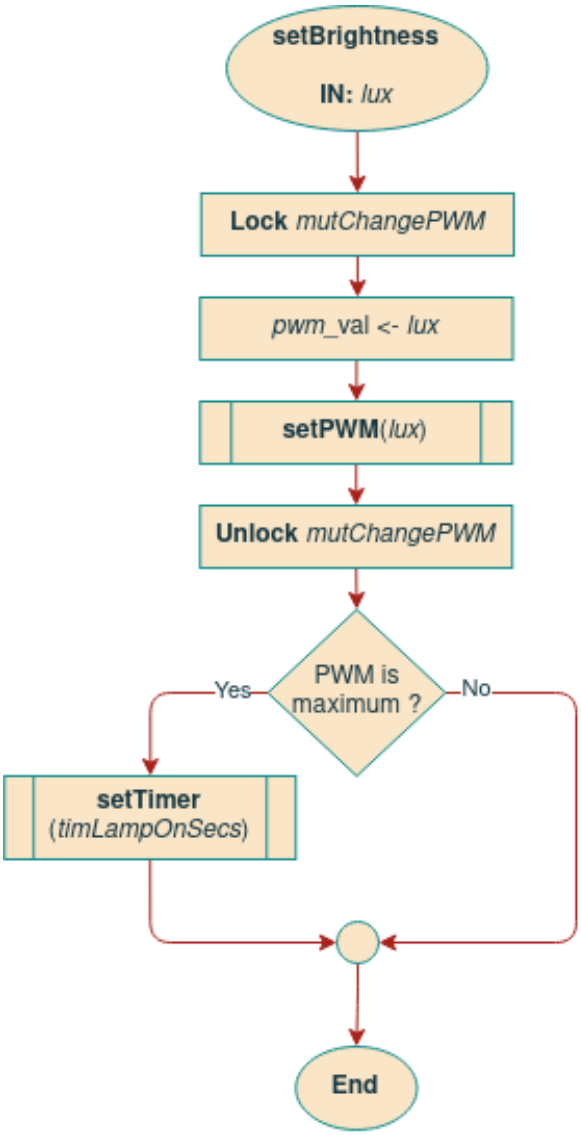


Figure 7.37: Flowchart: CLamp setBrightness method.

CParkDetection Methods

In the method *calcVacantsNum()*, one receives an image frame and processes it in order to determine the number of available parking spots. This does the detection of cars using a pre-trained model. When analyzing the image, if the coordinates of a detected car matches the coordinates of the parking spot (i.e the parking outline, obtained by *getOutline()*) then one can assume that the parking spot is occupied.

This method, represented in figure 7.38, is used to process the image captured by the camera in order to detect the parking spots outline. This is done using the algorithm defined previously in the section 5.5, and starts by converting the frame to a grey scale image, apply the canny edge filter to highlight the edges of the image captured. After having the edges, one can select only the vertical and horizontal straight lines and intersect them, storing the intersection points, that are the parking spot coordinates, into the private member *parkCoords*.

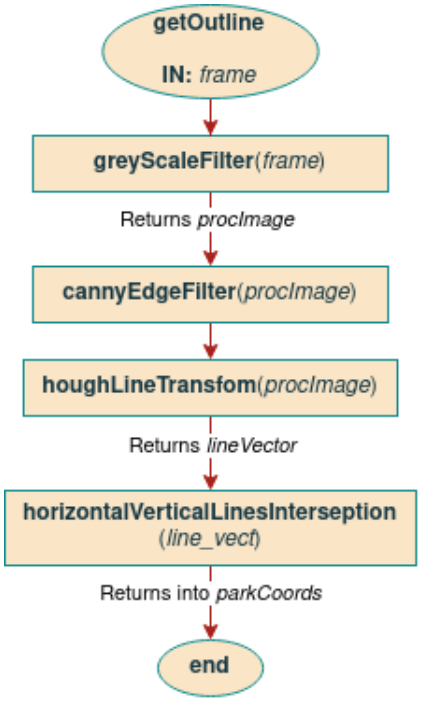


Figure 7.38: Flowchart: CParkDetection getOutline method.

CLdr Methods

The constructor for the class *CLdr* is presented in 7.39. This creates an *CLdr* object, initializing the LDR sensor, defining the initial luminosity state, *luxState*.

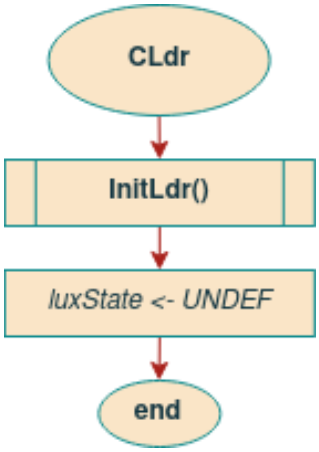


Figure 7.39: Flowchart: CLdr constructor.

In order to obtain the current luminosity state, *DAY* or *NIGHT*, there is a method called *getLuxState*, presented in figure 7.40. This method, gets the sensed luminosity level from the LDR sensor, and determines the current luminosity state.

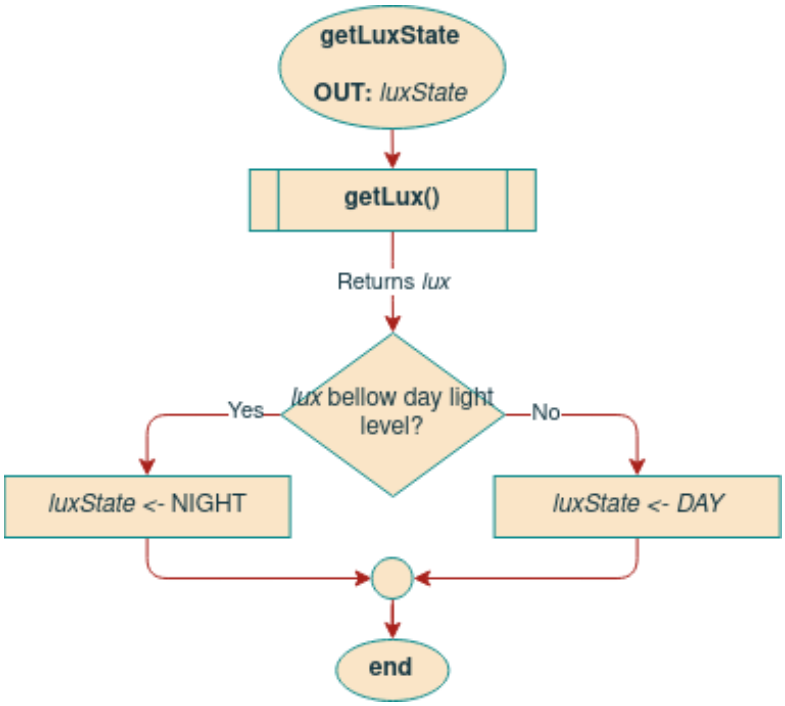


Figure 7.40: Flowchart: CLdr getLuxState method.

CPir Methods

In figure 7.41 is shown the CPir class constructor. This creates a CPir object, by firstly inserting the respective device driver, which will be developed, as a constraint of this project. Then an ISR is assigned to the PIR sensor, being that defined by a function pointer passed by argument into the constructor. Whenever the PIR detects movement in the surroundings, the output pin will change its value, triggering an interrupt, handled by the defined ISR.

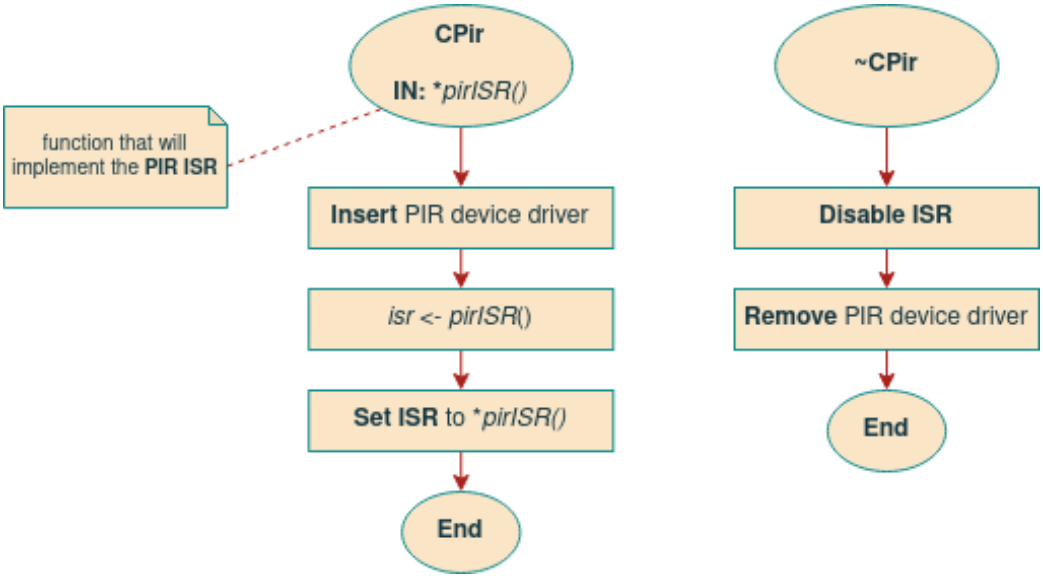


Figure 7.41: Flowchart: CPir constructor.

This class also implements two methods, *enable()* and *disable()*, which are responsible for enabling and disabling the CPir ISR, *isr*.

CFailureDetector Methods

In figure 7.42 is shown the CFailureDetector class constructor, responsible of creating a CFailureDetector object, by assigning an ISR to this detector, which is passed by argument into the constructor.

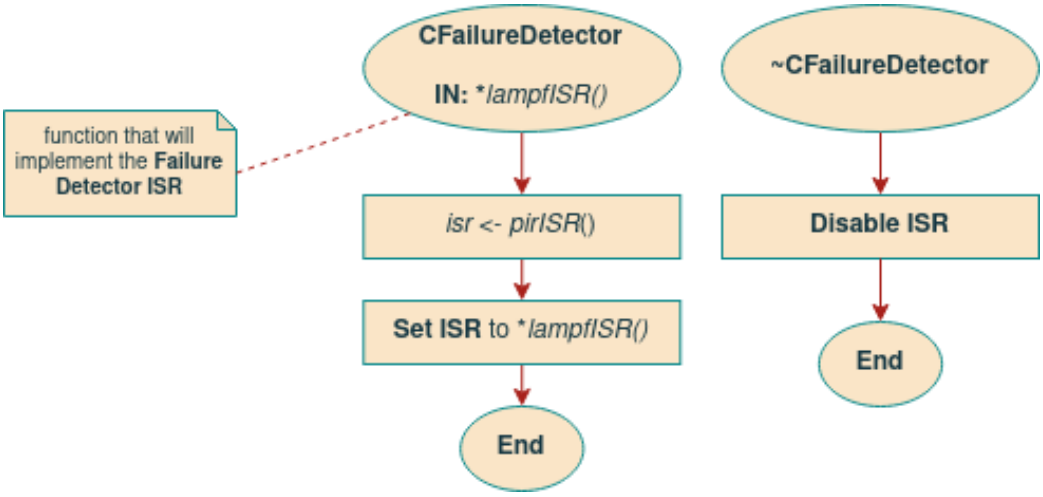


Figure 7.42: Flowchart: CFailureDetector constructor.

This class also implements two methods, *enable()* and *disable()*, very much like CPir methods, which are responsible for enabling and disabling the CFailureDetector ISR, *isr*.

7.1.9 Device Driver

In this project it will be implemented a device driver for a GPIO pin, used to read from the sensors PIR and LampFailureDetector.

First, one has to see what are registers that the board uses, relating to GPIO pins. In the Raspberry Pi 4B documentation, [47], one finds the registers easily for the GPIO pins that needs to be used for implementing the interface. The BCM2711 chip has 58 GPIO lines, split into three banks. Bank 0 is the one mapped on the header pins, from GPIO 0 to GPIO 27. Also, the GPIO registers base address is the 0x7E200000 and the address used is 32-bits.

To configure a pin as output or input, or to set or reset its value, the registers presented in table 7.1 are used.

Address	**Field Name**	**Description**	**R/W**
0x7E200000	GPFSEL0	GPIO Function Select 0 (GPIO 0-9)	R/W
0x7E200004	GPFSEL1	GPIO Function Select 1 (GPIO 10-19)	R/W
0x7E200008	GPFSEL2	GPIO Function Select 2 (GPIO 20-29)	R/W
0x7E20000C	GPFSEL3	GPIO Function Select 3 (GPIO 30-39)	R/W
0x7E200010	GPFSEL4	GPIO Function Select 4 (GPIO 40-49)	R/W
0x7E200014	GPFSEL5	GPIO Function Select 5 (GPIO 50-53)	R/W
0x7E200018	-	Reserved -	-
0x7E20001C	GPSET0	GPIO Pin Output Set 0 (0-31)	W
0x7E200020	GPSET1	GPIO Pin Output Set 1 (32-53)	W
0x7E200024	-	Reserved -	-
0x7E200028	GPCLR0	GPIO Pin Output Clear 0 (0-31)	W
0x7E20002C	GPCLR1	GPIO Pin Output Clear 1 (32-53)	W

Table 7.1: Raspberry Pi Registers used for GPIO configuration.

The *GPFSELx* register is used to define the operation of the GPIO pins, input or output. The *GPFSETx / GPFCLRx* are registers used to set or clear pins, respectively. Each register *GPFSELx* contains 10 GPIOs.

As defined in the Hardware Specification section, PIR sensor will have its output connected to the pin 27 of the Raspberry Pi, as the Lamp Failure Detector will have its output connected to the pin 22. Therefore, the register to define both pins as input is *GPFSEL2*.

7.1.10 Test Cases

It is important to know how the local system works, and how it will behave upon certain events. In table 7.2 are shown test cases to this system.

Test Case	Expected Output	Real Output
LoRa Communications		
Establish connection with the gateway	Connection established	-
Send a command to the gateway	Command sent	-
Receive commands	Command received	-
Process commands	Execute command if valid	-
Lamp PWM Control		
Set PWM=100	Lamp at maximum bright	-
Set PWM=0	Lamp off	-
Change PWM signal	Variable lamp brightness	-
Sensors		
Motion detected	Lamp at maximum bright	-
Lamp at maximum bright	Lamp ON for a defined period of time	-
LDR detect nighttime	Lamp at minimum bright	-
LDR detect daytime	Lamp OFF	-
Motion/LDR detection	Send correct commands to remote system	-
Lamp Failure Detected	Sent correct command to remote system	-
Camera		
Open camera device	Device opened correctly	-
Capture camera image	Variable containing image	-
Get parking outline	Obtain parking coordinates from the image	-
Car detection	Car detected inside parking outline	-
Parking detection	Get number of available parks	-

Table 7.2: Test Cases: Local System.

7.2 Gateway

One can define the relationship between the tasks as follows, in figure 7.43. The rule of the gateway is to receive LoRa messages, from the Local Systems (LS), and send the content in TCP-IP messages, to the remote server (RS), or vice versa.

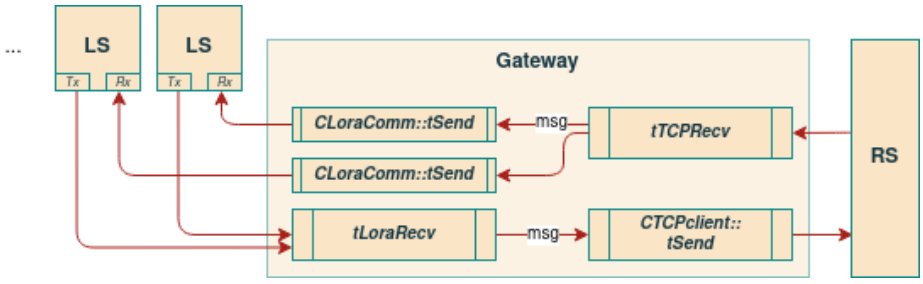


Figure 7.43: Gateway Overview.

7.2.1 Class Diagrams

In figure 7.44 is represented the class *CGateway*, which is the main class of the system, responsible for initializing the objects of each class listed below.

- **CLoraComm:** manages the LoRa communications with the local systems, interfacing with the LoRa module; this class was already detailed previously;
- **CTCPclient:** manages the TCP-IP communication with the remote server;

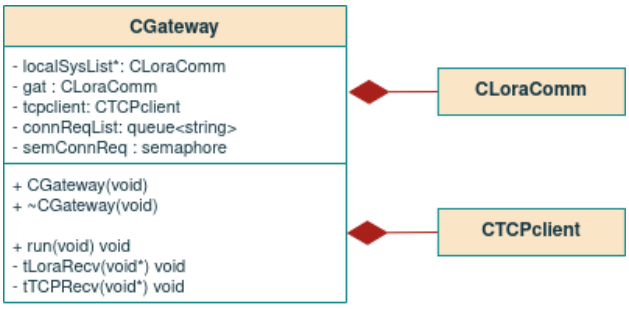


Figure 7.44: CGateway Class Diagram.

Class CTCPclient

In figure 7.45 is shown the CTCPclient class. This class defines a TCP-client capable of establishing a connection to a given remote server and to exchange messages to it, via TCP-IP. Like in the CLoraComm class, this class inherits from the class CCommunication.

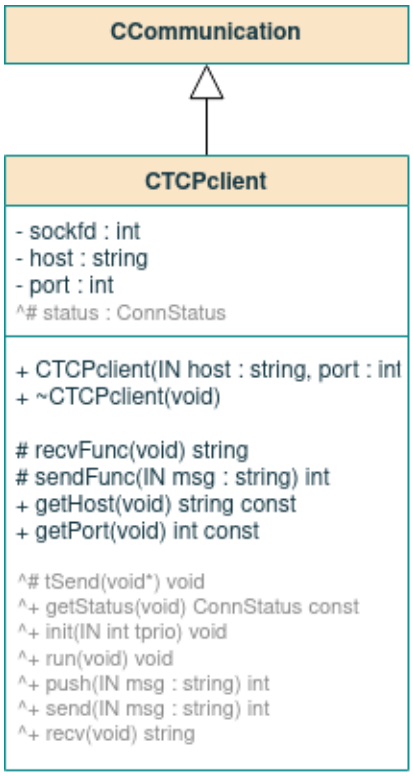


Figure 7.45: Class Diagram: TCPclient.

7.2.2 Task Overview

One can define and describe briefly how the gateway is implemented, making use of threads and processes.

- **CGateway::tLoraRecv:** receives all messages from local systems, using LoRa communication;
- **CGateway::tTCPRecv:** receives all messages from the remote server, using TCP-IP communication;
- **CLoraComm::tSend:** sends all received messages from the remote server, in *tTCPRecv*, to the local systems, using LoRa communication;
- **CTCPclient::tSend:** sends all received messages from local systems, in *tLoraRecv*, to the remote server, using TCP-IP communication;

7.2.3 Task Priority

The priority assignment diagram for the gateway is represented in figure 7.46.

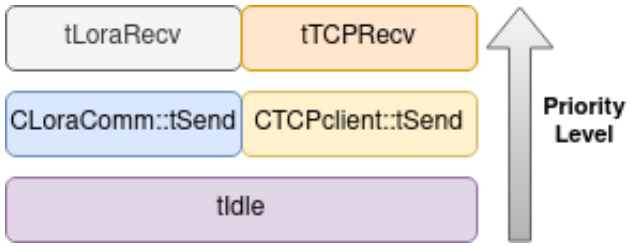


Figure 7.46: Gateway Priority Assignment Schematic.

7.2.4 Task Synchronization

Condition Variables

The condition variables used in this system are listed below.

- **CCommunication::condSend:** already detailed in the local system section; this condition variable is used indirectly by *CTCPclient* and *CLoraComm*.

Mutexes

The mutexes used in this system are listed below.

- **CCommunication::mutComms** and **CCommunication::mutTxMsgs:** already detailed in the local system section; this mutexes are used indirectly by *CTCPclient* and *CLoraComm*.

Semaphores

A semaphore is another tool of task synchronization, implemented as a variable or abstract data type. A semaphore can be seen as a record of how many units of a particular resource are available. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked) are called binary semaphores.

The semaphores used in this system are listed below.

- **semConnReq**: used to signalize the *run* method to create a new CLoraComm object, providing a new LoRa connection with a local system.

7.2.5 Start-up Process

In figure 7.47 is shown the start-up process for the gateway. When booting up, the gateway creates a CGateway object, initializing all needed members, and after that, runs until all the threads termination.

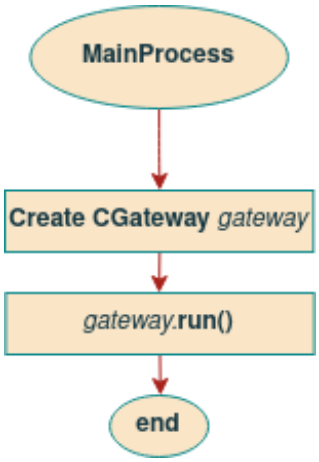


Figure 7.47: Start-Up Process: Gateway.

7.2.6 Flowcharts

CGateway Methods

In figure 7.48 is shown the *CGateway* class constructor. With this, one can create a CTCPclient and a CLoraComm objects, named *gat*, which will be the one responsible for receiving all the packets from the local systems. After this the threads are created and both objects begin running their threads for receiving and sending messages.

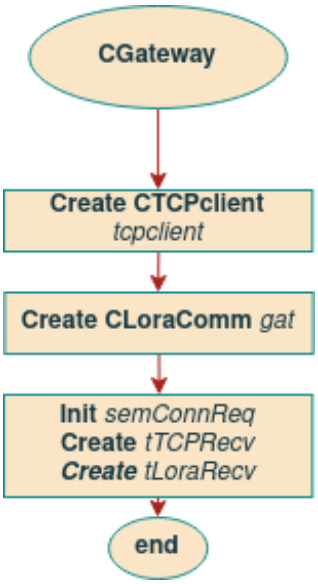


Figure 7.48: Flowchart: CGateway constructor.

In figure 7.49 is presented the *run* method from *CGateway* class. This is responsible for creating a new CLoraComm object for each new LoRa connection with a local system, being that controlled by the semaphore *semConnReq*. After that, the newly created object is added to the list of local systems already connected, *localSysList*.

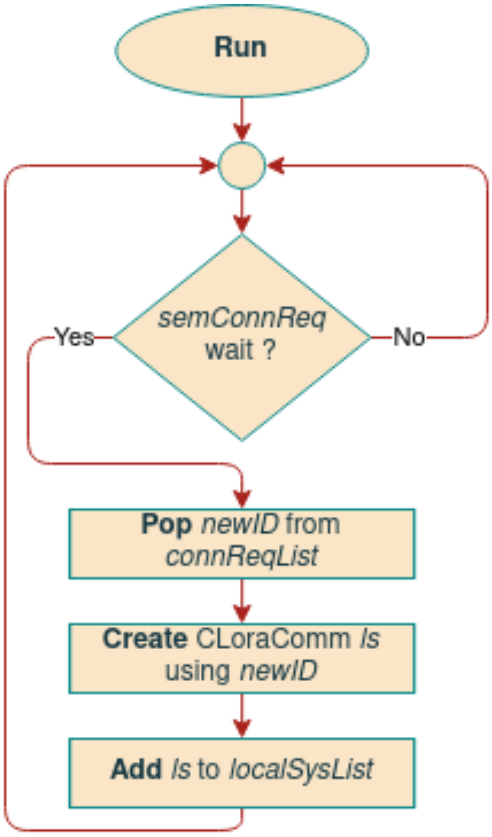


Figure 7.49: Flowchart: CGateway Run method.

This task, presented in figure 7.50, is responsible for receiving all the packets sent by the local systems, through LoRa communication. When a message is received, the gateway determines if it is a new connection request from a local system. If so, the message is added to the *connReqList* and the semaphore *semConnReq* is signalized in order for the *run* method to create a new object CLoraComm regarding the new connected local system. After this, the message is pushed into the TCP client list of queued messages, waiting to be sent to the remote server.

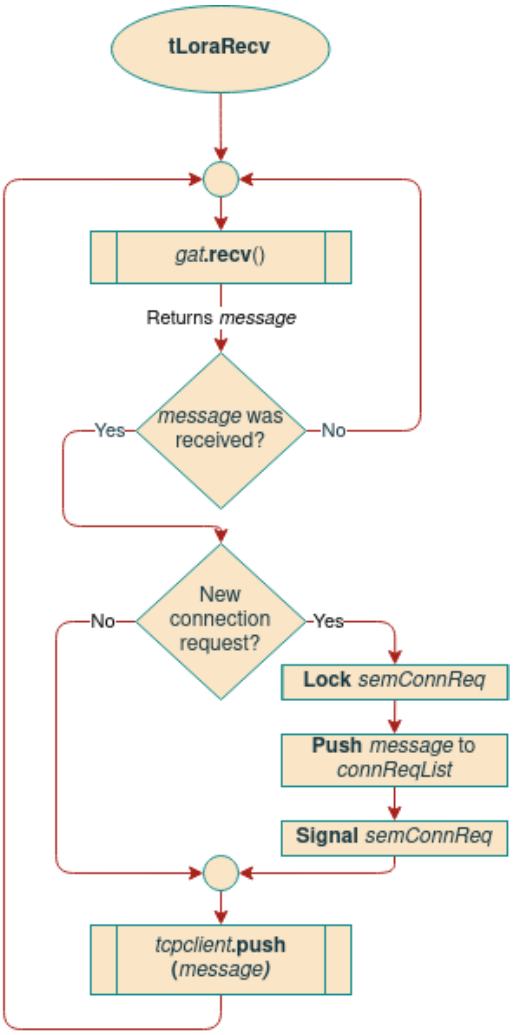


Figure 7.50: Flowchart: CGateway tLoraRecv method.

This task, presented in figure 7.51, is responsible for receiving the messages sent by the remote system to the local systems. Unlike the *tsend* tasks, this thread never enters the sleep state because one can receive a message at any moment. If it is received a message, then one can parse it, obtaining the correct destination local system object and the command to send, pushing it to the messages queue in CLoraComm, signaling its thread to send the message, via LoRa communication.

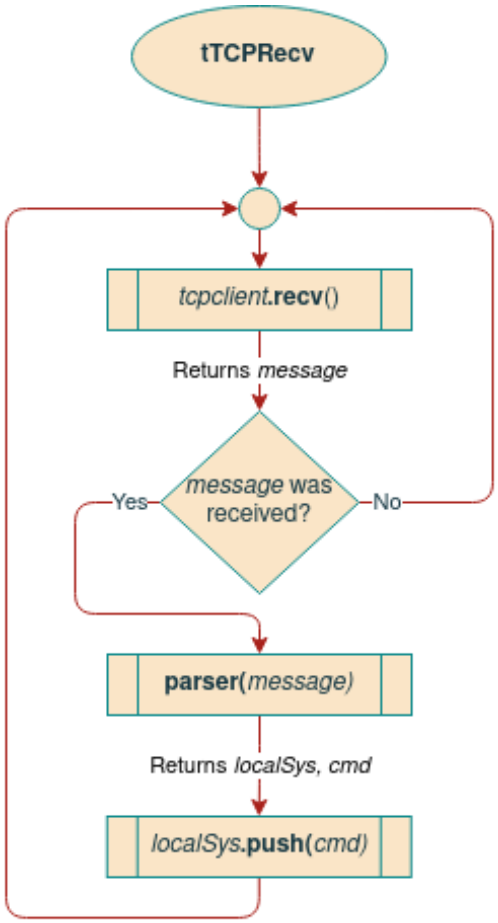


Figure 7.51: Flowchart: CGateway tTCPRecv method.

CTCPclient Methods

A TCPclient can be created through the use of the constructor, shown in figure 7.52. This connects to a given server, defined by the string *host*, and to the port *port*, via TCP-IP and initializes all the private members.

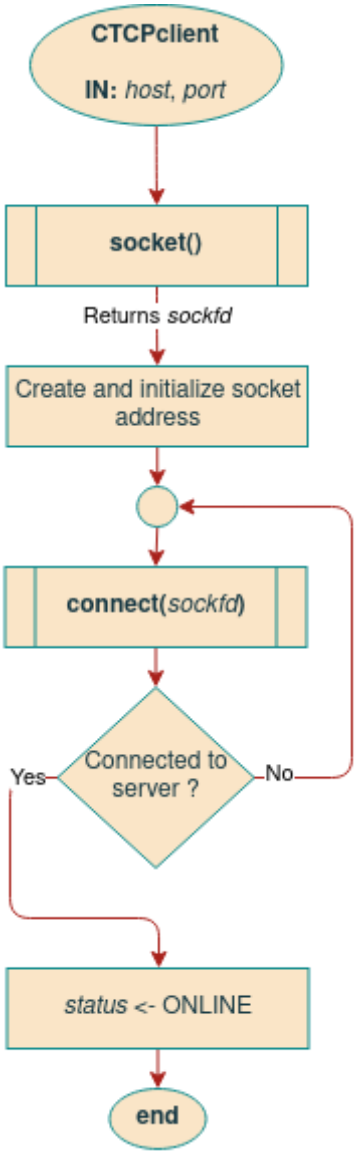


Figure 7.52: Class Diagram: CTCPclient constructor.

As in CLoraComm, this class must implement *recvFunc* and *sendFunc*, as they are pure virtual methods from CCommunication.

In this class, one will use existent functions *TCPReceive* and *TCPSend* to implement TCP-IP communication, as shown in figures 7.53 and 7.54. This functions make use of the socket created when the TCP-IP connection was established.

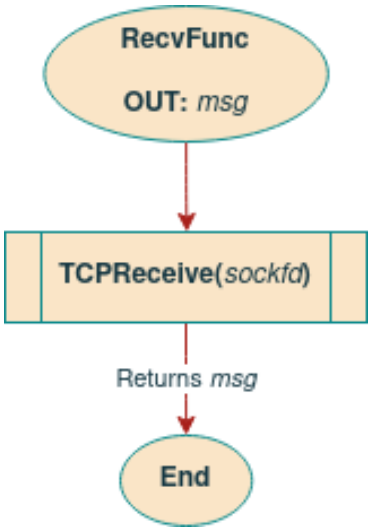


Figure 7.53: Flowchart: CTCPclient recvFunc method.

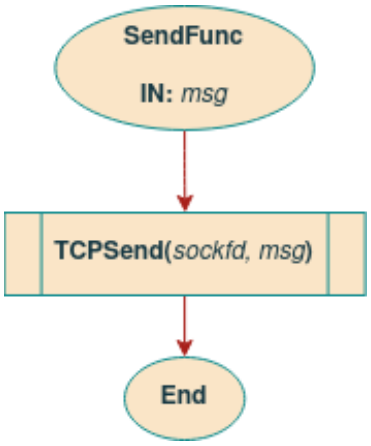


Figure 7.54: Flowchart: CTCPclient sendFunc method.

7.2.7 Test Cases

It is important to know how the gateway works, and how it will behave upon certain events. In table 7.3 are shown test cases to this system.

Test Case	Expected Output	Real Output
LoRa Communications		
Establish connection with LS	Connection established	-
Receive message from LS	Identify the referent LS	-
Send message to a specific LS	Only the defined LS receives the message	-
Send message to a group of LS	All in the group receive the message	-
TCP-IP Communications		
Establish connection with RS	Connection established	-
Receive message from RS	Identify destination ID	-
Send message to the RS	Message sent	-

[LS] Local System
[RS] Remote Server

Table 7.3: Test Cases: Gateway.

7.3 Remote System

The remote system is responsible for the communication between the local systems and the internet. In this project, the internet is used to maintain a cloud database, so the remote system does the data bridge between the network of street lampposts and a cloud.

7.3.1 Class Diagram

In figure 7.55 is represented the class diagram of the remote system. The class *CRemoteSystem* is the main class of this system, that initializes the objects of each class, listed below. As members, this class has a server, *CTCPServer*, a list of connected clients, *CClients*, and a database, *CDataBase*.

- **CTCPServer:** responsible for creating the server and accepting the clients connections;
- **CClient:** client class, stores the clients information, receives their messages, executing the respective command and sends its output to the client;
- **CDataBase:** provides an interface between the server and the database.

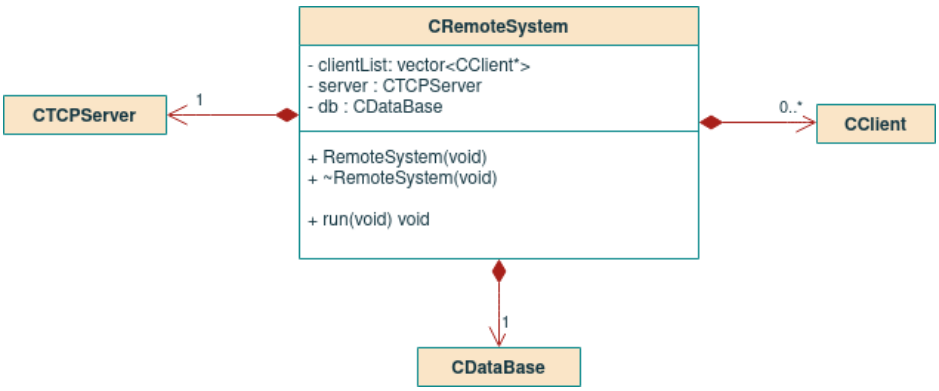


Figure 7.55: Remote System Class Diagram.

Class CTCPServer

This class is responsible for creating the server, using *createServer* method. Furthermore, using the function *acceptConnection*, this class accepts the clients connections returning the socket file descriptor. This class contains the information about the server address, *addr*, the listen socket descriptor, *listenSd* and the maximum number of connected clients, *maxClients*. In figure 7.56, one can see the class *CTCPServer* diagram.

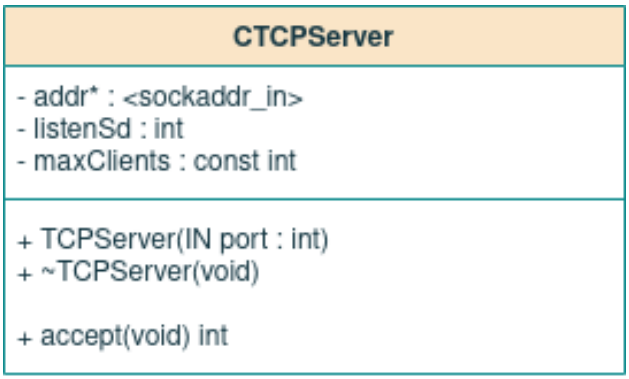


Figure 7.56: Class Diagram: CTCPServer.

Class CClient

The class *CClient*, represented in figure 7.57, is responsible for storing information about a client connected to the remote system, like the *cmdList*, a vector that stores the commands list that the client can execute and *clientSock*. The *clientSock* has information about the client like its *state*, *name*, identification number, *index*, sock file descriptor, *sockFd* and *type*, that can be *APPLICATION*, for a mobile application client, *WEBSITE*, for a web site client or *GATEWAY*, for a gateway device client. Furthermore, this class inherits from the class *CCommunication*, that implements the mechanism of sending messages to the client, and implements the thread to receive and parse messages from the client, *tRecv*.

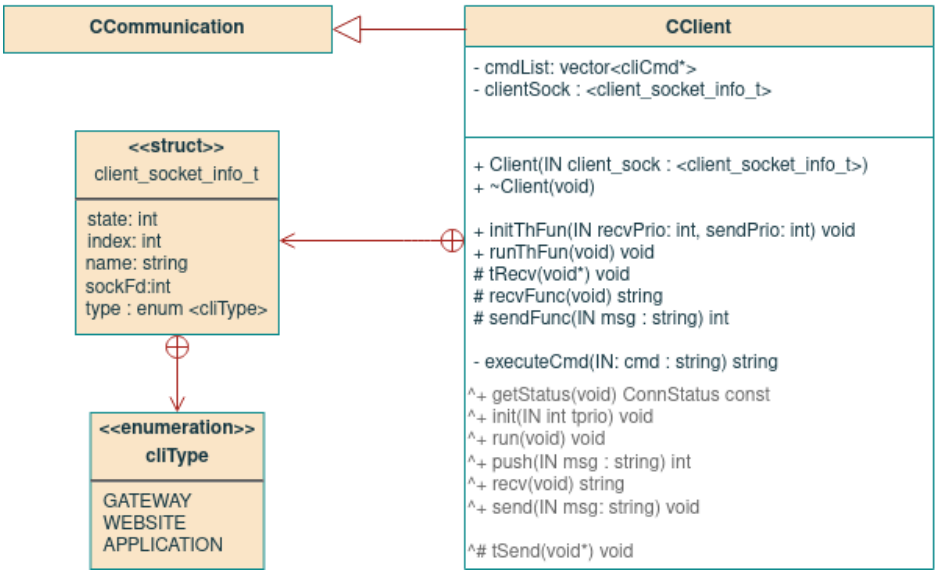


Figure 7.57: Class Diagram: CClient.

Class CDataBase

This class diagram is shown in figure 7.58, and is responsible for implement the interface between the server and the database, having for that a pointer to a database of type *MYSQL*. To interact with the database, the server must first prepare a query, using the function *prepareQuery*, and can execute the prepared query in the database using the method *execute*, that returns an error and assigns the command output to the string *outMessage*.

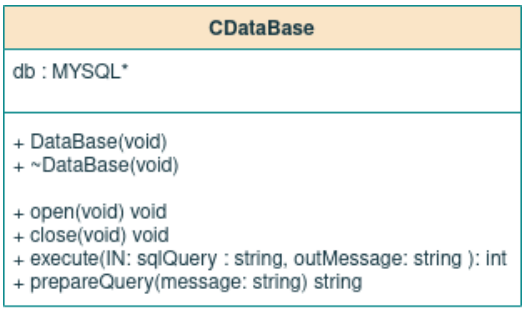


Figure 7.58: Class Diagram: CDataBase.

7.3.2 Task Overview

One can define and describe briefly how the remote system is implemented, making use of threads and processes.

- **CClient::tSend:** sends a message to the client;
- **CClient::tRecv:** receives a message from the client.

7.3.3 Task Priority

The priority assignment for the remote system is shown in figure 7.59.

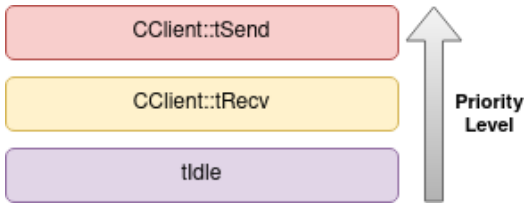


Figure 7.59: Remote System Main Process Priority Assignment.

7.3.4 Task Synchronization

As seen before, to have coordinate access to shared ressouces and services and avoid race conditions, the kernel has resources that provide synchronization between the tasks.

Mutexes

The mutexes used for the remote system are listed below.

- **CCommunication::mutComms:** already detailed in the local system section; used in-directly by *CClient*; used to protect the communications;
- **CCommunication::mutTxMsgs:** already detailed in the local system section; used in-directly by *CClient*; used to protect the insertion and removal of messages into *TxMsgs*;

Condition Variables

The condition variables used for the remote system are listed below.

- **CCommunication::condSend:** already detailed in the local system section; this condition variable is used indirectly by *CClient*; used to notify CCommunication::tSend that a new message is ready to be sent.

7.3.5 Commands and Constants

Constants

The constants used in the remote system are listed below.

- `MAX_CLIENT_NUM`: defines the maximum number of clients supported by the TCP server;
- `SERVER_PORT`: defines the port in which the server will listen for client connections;

Remote Server Commands

The commands used for the communication between the remote server and the local systems, through the gateway, are listed below.

- `IDRQ`: ID request; Used to obtain a list of local systems connected to each gateway;
- `<lsID> ON`: Turn on the lamp at full brightness of the local system with ID `<lsID>`;

The commands used for the communication between the remote server and the remote client WebSite, are listed below.

- `PARK <num> <GPScoord>`: Send `<num>` of available parking spots in the GPS coordinates, `<GPScoord>`.

The commands used for the communication between the remote server and the remote client Mobile Application, are listed below.

- `OP <op_result>`: Presents to the mobile application the last operation result, `<op_result>`.
- `LS <lsID> <status> <GPScoord>`: Sends to the mobile application the information of a single local system, regarding the local system ID, current status and its GPS coordinates.

Remote Client Commands

The commands used for the communication between the remote client Web-Site and the remote server, are listed below.

- `GET <streetName>`: request to obtain parking spots information in a street, `<streetName>`.

The commands used for the communication between the remote server and the remote client Mobile Application, are listed below.

- `ADD <street> <postCode> <parish> <county> <district> <GPScoord>`: add a new lamppost to the network, giving its street name, post code, parish, county, district and GPS coordinates;
- `REP <lsID> <street> <postCode>`: repair of a lamppost, with ID `<lsID>`, at the a given street and post code. When receiving this command, the remote server changes the status of the lamppost to OFF.

7.3.6 Start-up Process

The start-up process is shown in the figure 7.60. When the program initiates, the start-up process instantiate a object of the class *CRemoteSystem*. After that, it uses the method *run* that will be seen in the next subsection.

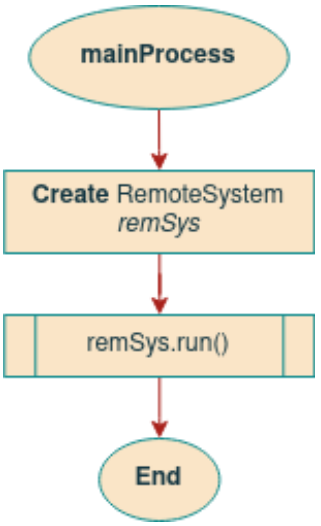


Figure 7.60: Start-Up Process.

7.3.7 Flowcharts

CRemoteSystem

The figure 7.61 represents the *CRemoteSystem* class's constructor that is responsible for the creation of the *clientList* vector, the database object, *db* and the server *server* in the port passed by argument.

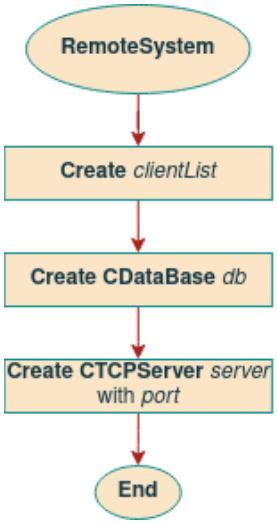


Figure 7.61: Flowchart: CRemoteSystem Constructor.

In figure 7.62 is shown the *CRemoteSystem* run method. Is is used to, after the creation of the server socket, accept the connections coming from the clients, using the *CTCPServer* method to accept connections, *acceptConnection*, that returns the socket descriptor of the new connection. If the socket descriptor *sd* is valid, then it is created a client *CClient* with this variable and the client is inserted in the client vector, *clientList*.

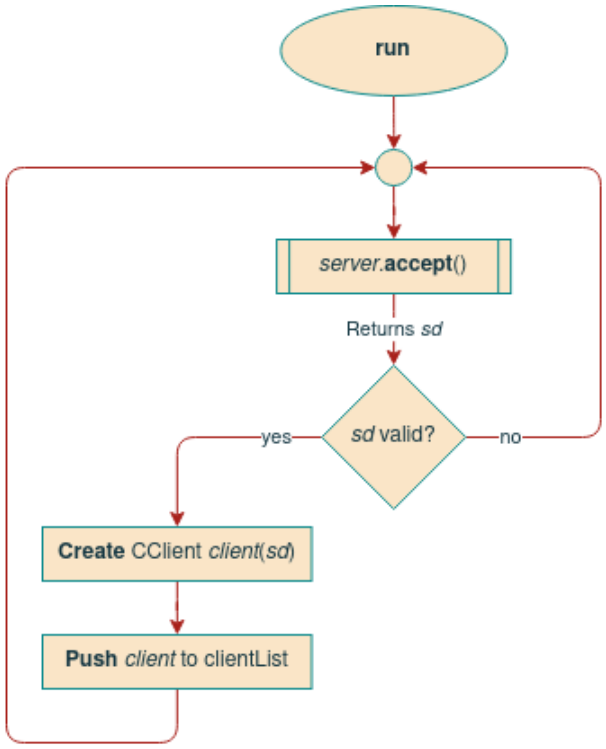


Figure 7.62: Flowchart: CRemoteSystem run.

CTCPServer

In figure 7.63 is represented the constructor of the *CTPCServer* class, that is responsible for creating the server socket in the port passed by argument for a maximum number of clients, *maxClient*. This constructor creates a server socket with the IPv4 protocol connection, initialize the *addr* with the port number passed by argument, binds the socket and listen to a maximum number of clients connections specified by *maxClient* passed by argument.

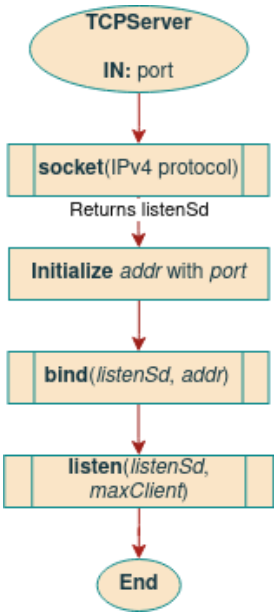


Figure 7.63: Flowchart: CTCPServer Constructor.

In figure 7.64 is represented the *acceptConnection* method, that accepts a connection from a client in the socket referenced by the socket file descriptor *listenSd*, returning the file descriptor of the new connection, *sd*.

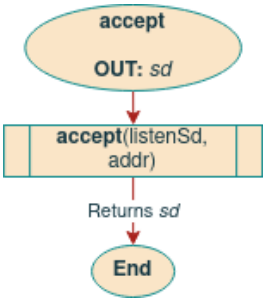


Figure 7.64: Flowchart: CTCPServer accept.

CClient

In figure 7.65 is represented the constructor of the *CClient* class. Every time a client connects to the remote system, it creates an instance of this class, passing by argument, the information about the client to initialize the

clientSock variable. The constructor also creates the command list that the type of client can execute, *cmdList*, and the threads *tRecv* and *tSend*, using the method *initThFun*.

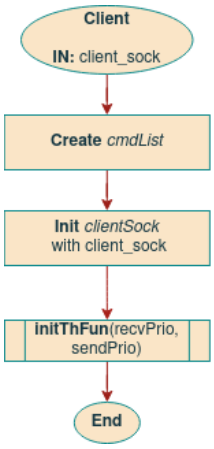


Figure 7.65: Flowchart: CClient Constructor.

In the method *initThFun*, shown in figure 7.66, it is created the thread *tRecv* with priority *recvPrio*, that receives messages from the clients, and the thread *tSend* with priority *sendPrio*, that is implemented in the class *CCommunication* and sends messages to the clients.

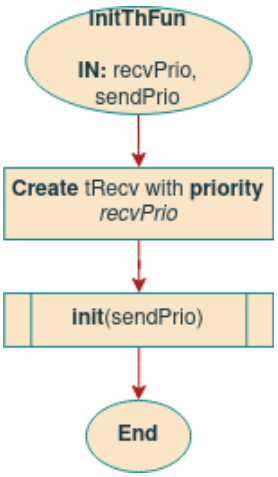


Figure 7.66: Flowchart: CClient initThFun.

The method *runThFun*, represented in figure 7.67, is responsible for waiting for the termination of the threads *tRecv* and *tSend*.

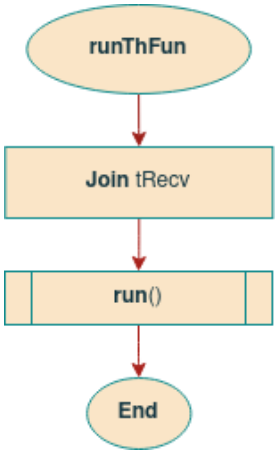


Figure 7.67: Flowchart: CClient runThFun.

The figure 7.68 shows the thread that receive messages from the clients, *tRecv*. When a message is received, the function *recv* returns the received message. If it is not empty, then one can parse and execute the command extracted from the received message, using the *parseExecute* function. When the command is valid, this function returns a message to be sent back to the client. In order to send this message to the client, one needs to use the *CCommunication* method to push this message to the sending buffer.

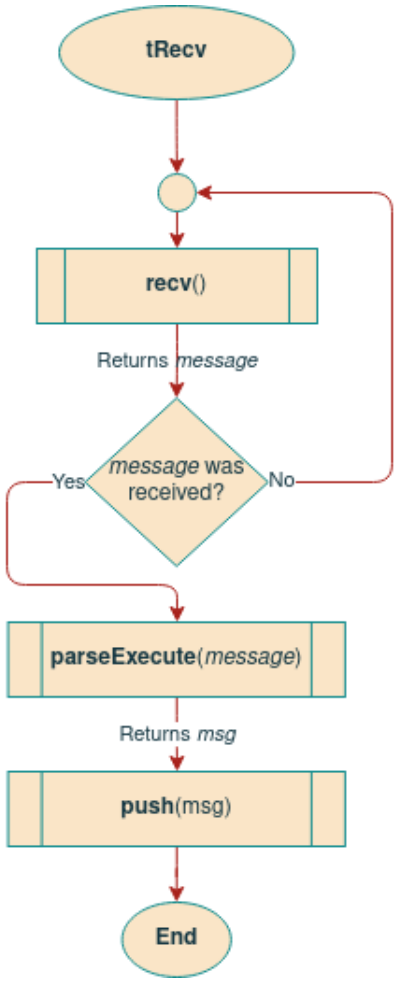


Figure 7.68: Flowchart: CClient tRecv.

In figure 7.69 is represented the flowchart of the *parseExecute* function, that is responsible for parsing and execute the message sent by the client. It is needed to verify if the client that sent the command has rights to execute that command, so the command sent is searched in the *cmdList* assigned to the respective client. If the message corresponds to a valid command for that client, then one can execute the command. If the command doesn't exist in the command list of that client, then the *msg* is assigned with an error message. The output of this function is the string message to be sent back to the client, *msg*.

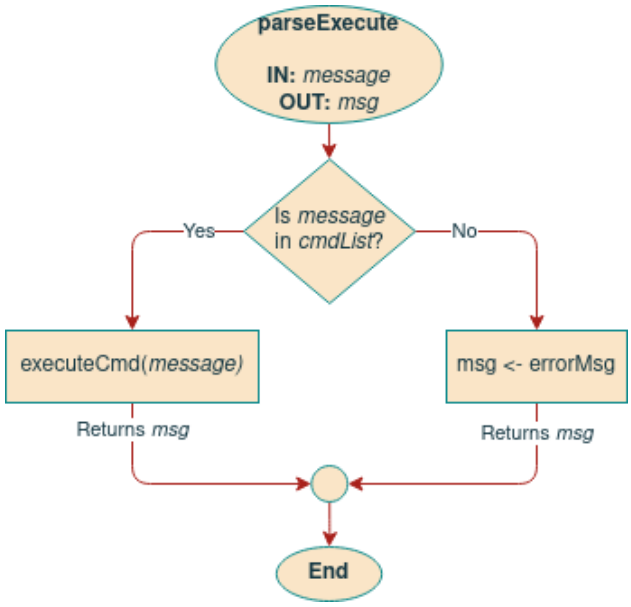


Figure 7.69: Flowchart: CClient parseExecute.

As for the previous subsystems, this class must implement the *recvFunc* and *sendFunc* methods, as they are pure virtual methods from CCommunication.

In this class, one will use existent functions *TCPReceive* and *TCPSend* to implement TCP-IP communication, as shown in figures 7.70 and 7.71. This functions make use of the socket created when the TCP-IP connection was established.

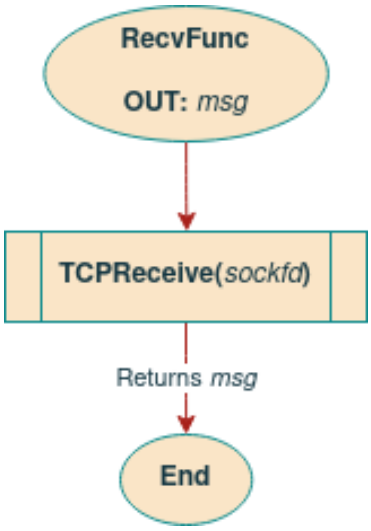


Figure 7.70: Flowchart: CClient recvFunc method.

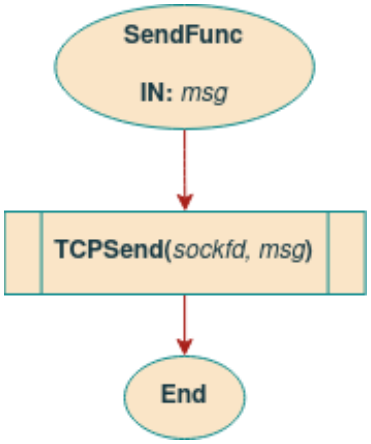


Figure 7.71: Flowchart: CClient sendFunc method.

7.3.8 Test Cases

It is important to know how the remote server works, and how it will behave upon certain events. In table 7.4 are shown test cases to this system.

Test Case	Expected Output	Real Output
Database Management		
Open DB	DB opened and ready to use	-
Get data from DB	Receive requested data	-
Insert/Update/Remove data into DB	Insert/Update/Remove successful	-
TCP-IP Communications		
Establish connection with RC	Connection established	-
Identify the RC	RC type and ID obtained	-
Process commands	Execute command if valid	-
Gateway Communications		
Receive message from a gateway	Identify the referent LS	-
Send message to a specific gateway	Only the defined gateway receives the message	-
Send message to a group of gateways	All in the group receive the message	-
Send message to the gateway connected to a specific LS	Only the specified LS receives the message	-
Receive lamp ON command from a LS	Send same command to the neighbors LS	-
WebSite Communications		
Get number of parking spots in a given location	DB returns number of parking spots in that location, if existent in the DB	-
Mobile App. Communications		
Operator Login	Verify credentials	-
Add/remove/update lamppost info	Verify new info and insert into DB if valid	-

[DB] Database
[LS] Local System
[RS] Remote Server
[RC] Remote Client

Table 7.4: Test Cases: Remote server.

7.4 Database

Relational Model

The database's relational model is presented below.

lamppost (pole_id, gps, pole_status);
location (gps, post_code, street_name);
region (post_code, operator_id, parish, county, district);
operator (operator_id, operator_name);
parking_space (park_id, gps, park_type, park_status).

Logical Data Model

The database's logical data model is presented in figure 7.72.

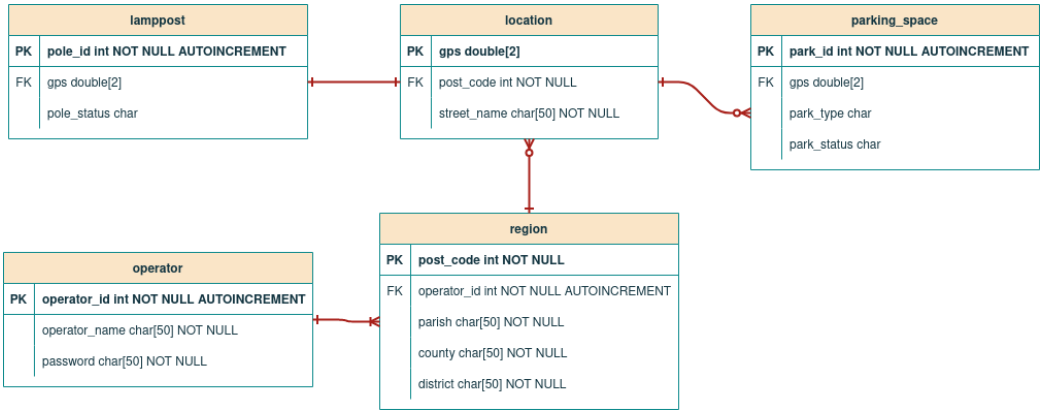


Figure 7.72: Database Logical Data Model.

Methods used

Through the remote server, each remote client can have different permissions of access to the database, using the methods shown below.

- **Mobile Application:** Add, Get, Update, Delete;
- **Web Site:** Get;
- **Gateway:** Update.

7.5 GUI Layouts

In this section it is presented the GUI layouts, where the users can interact with the system.

Mobile Application

The mobile application is used by the operator, in order to manage the network of lampposts that is assigned to him.

The first step is to login the system, so he has to insert is credentials *ID Operator* and *Password*, provided by the company. The login layout is shown in figure 7.73a. If the login was successful, then the operator can select one of this operations: *Add New Lamppost*, *Repair Lamppost* or *Consult Lamppost Network*, as shown in figure 7.73b. The operator can also logout of his account.

(a) Login. (b) Select Operation.

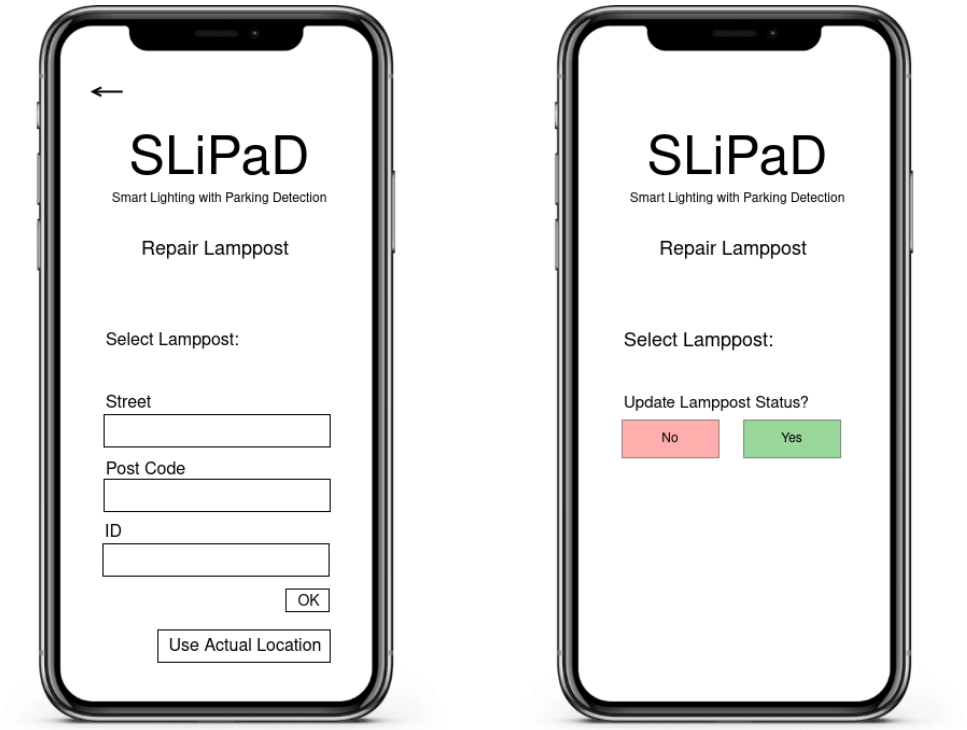
Figure 7.73: Mobile Application Layout.

If the operator clicks on the *Add New Lamppost* option, then he is redirected to the layout represented in figure 7.74, where he can add a new lamppost to the network, inserting all the information about the location of the post or use his mobile device actual location.

Figure 7.74: Mobile Application Layout: Add New Lamppost.

If the option chosen was the *Repair Lamppost*, in order to change the lamppost status, he can insert the location of the post or use his mobile device actual location, as seen in figure 7.75a.

If the lamppost selection was valid, then the layout represented in figure 7.75b appears, to the operator confirm that he wants to update the lamppost status.



(a) Repair Lamppost. (b) Repair Lamppost Confirmation.

Figure 7.75: Mobile Application Layout.

On the other hand, if the operator selects the option *Consult Lamppost Network*, the network information will appear like shown in figure 7.76, where the red point are lampposts with a broken lamp and the green points are the lampposts with no lamp problems.

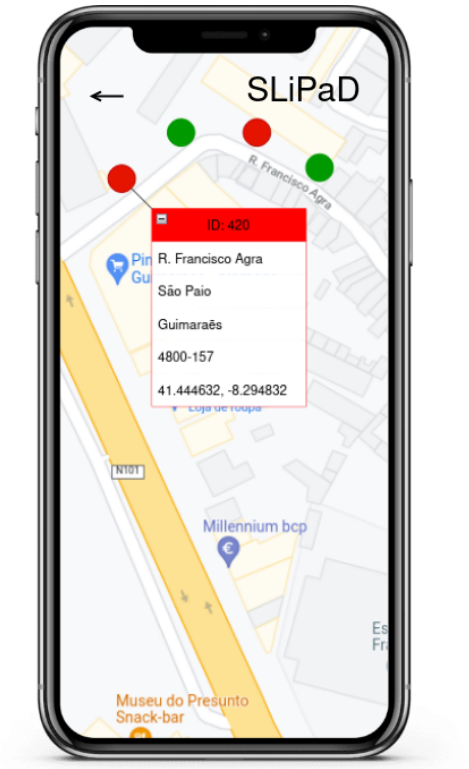


Figure 7.76: Mobile Application Layout: Consult Lamppost Network.

Web Site

The Web Site can be used by a user that wants to know where there are parking spots available near a location.

When the user enters the site, it is presented a menu where he can insert the location where he wants to know the parking spots availability or use his actual location, as seen in figure 7.77a.

After inserting the location, it will be shown the parking spots available near the location inserted, as seen in figure 7.77b, where the red points represent the location of an empty parking spot.

(a) Enter Location. (b) Empty Parking Spots Location.

Figure 7.77: Web Site Layout.

7.6 Tools

- **Git:** free and open source distributed version control system;
- **GitHub:** provider of Internet hosting for software development and version control using Git;
- **Buildroot:** Tool to configure and generate the Raspberry Pi Kernel image;
- **C/C++:** Programming language used to develop local system and remote system core;
- **Qt Creator:** Cross-platform IDE used for the Mobile Application development;
- **HTML:** Programming language chosen for the WebSite development;
- **Python:** Programming language chosen for the Haar cascade training stage;
- **MySQL:** Relational database management system used for the remote server database;

7.7 COTS

- **POSIX Threads API:** Used for thread creation and management;
- **OpenCV API:** Used for image capture and processing;
- **Qt API:** Used for the GUI;
- **RaspiCam API:** C++ API for using Raspberry camera with/without OpenCv;
- **Google Cloud SQL:** Google Cloud service that allows for immutable data storage and retrieval;

7.8 Third-Party Libraries

- **Light Sensor (TSL258x) Device Driver:** Open-source device driver used for interfacing with the luminosity sensor [48];
- **LoRa SX1278 Library:** An Arduino open-source library for sending and receiving data using LoRa radios [49]. This is implemented to the Arduino board, but it will be adapted to the Raspberry Pi 4 Model B.

Chapter 8

Implementation

8.1 Tools Setup

Before doing the system configuration it is necessary to first setup all of the used tools, as will be next presented.

8.1.1 Git

In order to make collaboration easier, allowing change by multiple people to all be merged into one source, Git will be used. Git is the most commonly used version control system. Before using it, it is necessary to do the correct setup as shown bellow.

```
1 $ sudo apt install git
2 $ git config --global user.name "John Doe"
3 $ git config --global user.email johndoe@example.com
4 $ git config --global core.editor subl
5 $ cd ~
6 $ git clone git@github.com:ESRGgroup9/slipad.git
```

With this steps, Git is installed in a local machine, username and user email is defined alongside with the default core editor. After this, one can clone the repository for this project, created in GitHub.

8.1.2 Buildroot

Buildroot is a simple, efficient and easy-to-use tool used to generate this project's embedded Linux system, through cross-compilation. The steps in order to install Buildroot in a local machine is shown bellow.

```
1 $ cd ~
2 $ mkdir buildroot
3 $ cd buildroot
4 $ wget https://buildroot.org/downloads/buildroot-2021.02.5.tar.gz
5 $ tar xzf buildroot-2021.02.5.tar.gz
6 $ cd buildroot-2021.02.5
```

After the installation is done, one can do the base configurations, essential to the support the rest of the configurations.

```
1 $ make raspberrypi4_defconfig
2 $ make menuconfig
3 $ make xconfig
4 $ make
5 $ make clean
```

The first command is used to configure a kernel image for the Raspberry Pi 4, as it does the necessary configurations regarding hardware handling along with fetching some board specific packages. Then with the second and third commands, one can generate the graphic interface seen in figure 8.1, presenting several sub-menus.

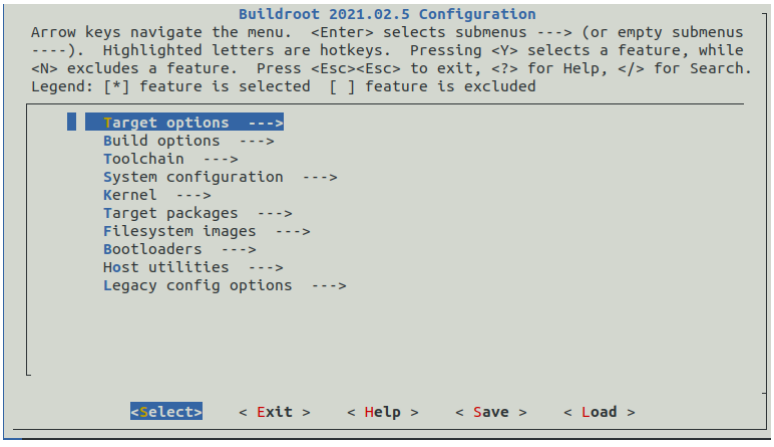


Figure 8.1: Buildroot menuconfig.

8.1.3 Qt

8.1.4 MySQL

8.2 System Configuration

In the Buildroot, using `make menuconfig` one can use a graphic interface (as previously shown in figure 8.1) to select packages and change configurations in order to create a custom linux image.

Under `Target-Packages` select:

- `Development tools`, select:
  - `git`
- `Networking applications`, select:
  - `arp-scan`
  - `arptables-legacy`
  - `dhcpcd`
  - `dropbear`
- `Hardware Handling`
  - `spi-tools`: LoRa module communicates with the Raspberry Pi through SPI, therefore it is necessary to enable the protocol and the tools required to use it.
- `Libraries -> Hardware Handling`, select:
  - `bcm2835`

8.3 Device Drivers

8.3.1 tsl2581

In order to insert the `tsl2581` device driver, available on the Linux kernel ([48]) one needs to execute a series of steps.

Firstly, one needs to add a new entry on the Raspberry Pi device tree, regarding the `tsl2581`. For that, one needs to change the Device tree Source file (.dts) for the Raspberry Pi, named `bcm2711-rpi-4-b.dts`. This will be later compiled into a device tree binary (.dtb), when creating an image using Buildroot, which will be later passed by the boot loader to the operating system kernel. [50]

```
$ cd ~/buildroot/buildroot-2021.02.5/output/build/linux-custom/arch/arm/boot/dts
$ nano bcm2711-rpi-4-b.dts
```

At the end of the file, before `__overrrides__`, the following code must be added, where `reg` is the I2C address of the device. [51]

```
&i2c1 {
  status = "okay";

  tsl2581@29 {
    compatible = "amstaos,tsl2581";
    reg = <0x29>;
  };
};
```

For the Raspberry Pi, in the `/boot/config.txt` file, one must enable `i2c1` with a `dtoverlay`, by adding the line of code shown next.

```
dtoverlay=i2c1
```

When booting the Raspberry Pi, one must insert the `tsl2581` device driver, but before that, some modules must be added using `modprobe`.

This is an I2C based sensor, which uses the Industrial Input/Output (IIO) interface. In order to provide the IIO API, needed in the device driver to be used, it is necessary to enable the IIO kernel subsystem, `industrialio`. To enable the I2C bus, one must load the `i2c-bcm2835` module. [52]

After that, one can insert the `tsl2581` device driver, `ldr.ko`.

```
1 $ modprobe industrialio
2 $ modprobe i2c-bcm2835
3 $ insmod ldr.ko
```

After the device driver insertion, one can use it to communicate with the sensor in order to acquire the ambient luminosity. For that, a "single" on-demand read can be issued by user-space directly by reading `/sys/bus/devices/iio:device/in_<type><index>_raw`. In this case, the `read_raw()` callback should handle basically all the steps necessary to get the required measurement. [53]

8.3.2 PIR

8.4 Image Generation

After all the necessary tools, packages and configurations are done, one needs to finally create the Linux custom image, that will run on the Raspberry Pi.

```
1 $ cd ~/buildroot/buildroot-2021.02.5/
2 $ make
```

After that one can copy the image into the SD card, that will go into the Raspberry Pi. For that one can use the linux `dd` command. [54]

```
1 $ sudo dd if=./output/images/sdcard.img of=/dev/sdb
```

With this command, one must define the input file, `if`, which is the linux image, and the output file, `of`, which is the SD card, being in this example named `sdb`.

8.5 System Initialization

In the `/boot/config.txt` file are some configuration parameters that are read when the system boots from the microSD card.

blablabla

Bibliography

[1] libelium, "Iot solutions smart cities," https://www.libelium.com/iot-solutions/smart-cities/ [Accessed on 17/11/2021].

[2] C. Helman, "Energy crisis 2021: How bad is it, and how long will it last?" Forbes, 2021, https://www.forbes.com/sites/christopherhelman/2021/10/19/energy-crisis-2021-how-bad-is-it-and-how-long-will-it-last/?sh=6a5feff14c63 [Accessed on 28/10/2021].

[3] H. Ritchie and M. Roser, "Energy," Our World in Data, 2020, https://ourworldindata.org/energy [Accessed on 28/10/2021].

[4] "Cars parked 23 hours a day," https://www.racfoundation.org/media-centre/cars-parked-23-hours-a-day [Accessed on 8/11/2021].

[5] R. Pi, "Raspberry pi 4," https://www.raspberrypi.com/products/raspberry-pi-4-model-b/ [Accessed on 22/11/2021].

[6] L. Alliance, "What is lorawan specification," https://lora-alliance.org/about-lorawan/ [Accessed on 3/12/2021].

[7] W.-S. Alliance, "Iot networks," https://wi-sun.org/iot-networks/ [Accessed on 3/12/2021].

[8] Z. Alliance, "Zigbee," https://zigbeealliance.org/solution/zigbee/ [Accessed on 3/12/2021].

[9] Sigfox, "Sigfox technology," https://www.sigfox.com/en/what-sigfox/technology [Accessed on 3/12/2021].

[10] GSMA, "Narrowband – internet of things," https://www.gsma.com/iot/narrow-band-internet-of-things-nb-iot/ [Accessed on 3/12/2021].

[11] ——, "Long term evolution for machines," https://www.gsma.com/iot/long-term-evolution-machine-type-communication-lte-mtc-cat-m1/ [Accessed on 3/12/2021].

[12] M. Intelligence, "Global connected street lighting market - growth, trends, covid-19 impact, and forecasts (2021 - 2026)," https://www.mordorintelligence.com/industry-reports/connected-street-lights-market [Accessed on 8/11/2021].

[13] FLASHNET, "intelilight smart street lighting control," https://www.flashnet.ro/project/intelilight/ [Accessed on 1/11/2021].

[14] Telensa, "Smart city solutions," https://www.telensa.com/solutions/ [Accessed on 1/11/2021].

[15] D. B. M. Research, "Global smart parking market," https://www.databridgemarketresearch.com/reports/global-smart-parking-market [Accessed on 22/11/2021].

[16] "intuvision parking," https://www.intuvisiontech.com/intuvisionVA_solutions/intuvisionVA_parking [Accessed on 8/11/2021].

[17] T. T. Network, "What are lora and lorawan," https://www.thethingsnetwork.org/docs/lorawan/what-is-lorawan/ [Accessed on 2/12/2021].

[18] L. Alliance, "What is lorawan," https://lora-alliance.org/wp-content/uploads/2020/11/what-is-lorawan.pdf [Accessed on 3/12/2021].

[19] ——, "Lora alliance," https://lora-alliance.org/ [Accessed on 9/12/2021].

[20] e. a. Yu Jiang, "Physical layer identification of lora devices," https://jwcn-eurasipjournals.springeropen.com/track/pdf/10.1186/s13638-019-1542-x.pdf [Accessed on 9/12/2021].

[21] S. Afzal, "I2c primer: What is i2c?" https://www.analog.com/en/technical-articles/i2c-primer-what-is-i2c-part-1.html [Accessed on 9/12/2021].

[22] S. Campbell, "Basics of the i2c communication protocol," https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/ [Accessed on 9/12/2021].

[23] P. Dhaker, "Introduction to spi interface," https://www.analog.com/en/analog-dialogue/articles/introduction-to-spi-interface.html [Accessed on 9/12/2021].

[24] S. Campbell, "Basics of the spi communication protocol," https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/ [Accessed on 9/12/2021].

[25] M. Alliance, "Mipi camera serial interface 2," https://www.mipi.org/specifications/csi-2 [Accessed on 9/12/2021].

[26] E. Products, "Csi-2 sensors in embedded designs," https://www.electronicproducts.com/camera-serial-interface-csi-2-sensors-in-embedded-designs/# [Accessed on 9/12/2021].

[27] M. W. Docs, "An overview of http," https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview [Accessed on 10/12/2021].

[28] ——, "Http request methods," https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods [Accessed on 10/12/2021].

[29] M. KerrisK, "The linux programming interface," chapter 14, 37 [Accessed on 8/12/2021].

[30] A. Cloud, "Deep understanding of daemon processes," https://topic.alibabacloud.com/a/deep-understanding-of-daemon-processes-from-the-concept-of-process-groups-font-colorredsessionsfont-and-endpoints_8_8_30138096.html [Accessed on 8/12/2021].

[31] T. L. I. Project, "Kernel space definition," http://www.linfo.org/kernel_space.html [Accessed on 8/12/2021].

[32] J. C. A. Rubini and G. Kroah-Hartman, "Linux device drivers," chapter 1 [Accessed on 8/12/2021].

[33] OpenCV, "Canny edge detection," https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html [Accessed on 9/12/2021].

[34] ——, "Hough line transform," https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html [Accessed on 9/12/2021].

[35] ——, "Cascade classifier training," https://docs.opencv.org/4.x/dc/d88/tutorial_traincascade.html [Accessed on 10/12/2021].

[36] R. P. Pinout, "The raspberry pi gpio pinout guide," https://pinout.xyz/# [Accessed on 1/12/2021].

[37] BotnRoll, "Tsl2581," https://www.botnroll.com/pt/luz-imagem/3516-sensor-de-luz-ambiente-tsl2581-i2c-ws.html [Accessed on 8/12/2021].

[38] ——, "Pir hc-sr501," https://www.botnroll.com/pt/outros/744-detector-de-movimento-pir.html [Accessed on 8/12/2021].

[39] P. Supply, "Raspberry pi camera board v1.3," https://uk.pi-supply.com/products/raspberry-pi-camera-board-v1-3-5mp-1080p [Accessed on 8/12/2021].

[40] SEMTECH, "Sx1276/77/78/79 transceiver," https://www.makerfabs.com/desfile/files/sx1276_77_78_79.pdf [Accessed on 8/12/2021].

[41] M. Robotics, "Sx1278 lora module 433m 10km ra-02," https://www.robotics.org.za/MCR127802 [Accessed on 8/12/2021].

[42] Electrofun, "Fonte de alimentação industrial 12v 60w 5a orno," https://www.electrofun.pt/inicio/fonte-de-alimentacao-industrial-12v-60w-5a-orno [Accessed on 8/12/2021].

[43] BotnRoll, "Step-down 9 38v para usb 5v 5a," https://www.botnroll.com/pt/conversores-dcdc/2986-step-down-9-38v-para-usb-5v-5a.html [Accessed on 8/12/2021].

[44] ——, "Regulador de tensão 5vdc para 3.3vdc," https://www.botnroll.com/pt/conversores-dcdc/3861-regulador-de-tens-o-5vdc-para-3-3vdc-800ma-step-down-ams1117-3-3.html [Accessed on 8/12/2021].

[45] ——, "Módulo sensor luz ldr," https://www.botnroll.com/pt/luz-imagem/3742-m-dulo-sensor-luz-ldr-c-potenci-metro.html [Accessed on 8/12/2021].

[46] R. Solutions, "Ip ratings and standards explained," https://rainfordsolutions.com/products/ingress-protection-ip-rated-enclosures/ip-enclosure-ratings-standards-explained/ [Accessed on 8/12/2021].

[47] R. P. T. Ltd, "Bcm2711 arm peripherals," https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf [Accessed on 8/12/2021].

[48] Linux, "Device driver for monitoring ambient light intensity." https://github.com/torvalds/linux/blob/master/drivers/iio/light/tsl2583.c [Accessed on 8/12/2021].

[49] ——, "An arduino library for sending and receiving data using lora radios." https://github.com/sandeepmistry/arduino-LoRa [Accessed on 9/12/2021].

[50] FileInfo.com, ".dtb file extension," https://fileinfo.com/extension/dtb [Accessed on 1/1/2022].

[51] L. Kernel, "Documentation-devicetree-bindings-iio-light-tsl2583.txt," https://mjmwired.net/kernel/Documentation/devicetree/bindings/iio/light/tsl2583.txt [Accessed on 1/1/2022].

[52] L. K. D. Database, "Config-i2c-bcm2835: Broadcom bcm2835 i2c controller," https://cateee.net/lkddb/web-lkddb/I2C_BCM2835.html [Accessed on 1/1/2022].

[53] bootlin, "The backbone of a linux industrial i/o driver," https://bootlin.com/blog/the-backbone-of-a-linux-industrial-i-o-driver/ [Accessed on 1/1/2022].

[54] L. manual page, "dd," https://man7.org/linux/man-pages/man1/dd.1.html [Accessed on 1/1/2022].