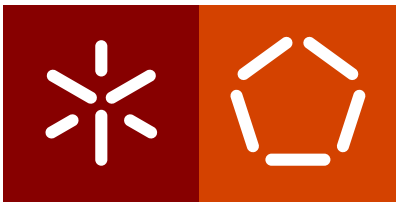

University of Minho
School of Engineering
Electronics Engineering department
Embedded systems

Project: Report

Marketing Digital Outdoor with gesture interaction — Analysis

Group 8

José Pires	A50178
Hugo Freitas	A88258

Supervised by:
Professor Tiago Gomes
Professor Ricardo Roriz
Professor Sérgio Pereira

December 23, 2021

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

Notation	Description	First used on page nr.
API	Application Programming Interface	17
CLI	Command Line Interface	15
COTS	Commercial off-the-shelf	13
CSI	Camera Serial Interface	1
DB	Database	15
DC	Direct Current	2
ER	Entity-Relationship	19
ERD	Entities-Relationships diagram	13, 19
FIFO	First-In, First-Out	28
GIF	Graphics Interchange Format	18
GPIO	General Purpose Input/Output	1
HDMI	High-Definition Multimedia Interface	1
HW	Hardware	i
LCD	Liquid Crystal Display	27
mA	milliampere	3
MDO	Marketing Digital Outdoor	iii
MDO-L	MDO Local System	7
MDO-RC	MDO Remote Client	11
MDO-RS	MDO Remote Server	11
MOSFET	Metal Oxide Semiconductor Field-Effect Transistor	9

Notation	Description	First used on page nr.
PCB	Printed Circuit Board	i
PoE	Power over Ethernet	2
REST	Representation State Transfer	17
SoC	System-on-a-Chip	1
SW	Software	iii
TCP/IP	Transmission Control Protocol/Internet Protocol	14
TTL	Transistor-Transistor Logic	1
UI	User Interface	13
UML	Unified Modeling Language	13
URL	Uniform Resource Locator	17

1. Design

In this section the theoretical foundations are used to design a viable solution, accordingly to the requirements and constraints listed. In the design phase, the product development starts, specifying the system in terms of hardware and software and its associated interfaces, the error handling required, and the design verification.

1.1. Hardware specification

The first step for system design is the HW specification. This can be pictured as a block diagram, this block diagram was already shown and mentioned in section **??** on Fig. **??**.

1.1.1. Architecture

Although that in the analysis phase an overview of the HW architecture was conceptualized, in this section, a more specific HW architecture is illustrated, using a block-diagram.

As it can be seen in Fig. 1.1, the **Raspberry Pi** is the main controller and it is powered by the **Power Supply** through **USB-C**. The **Power Supply** also supplies via **Micro-USB** the **LCD Display** control board and the **Fragrance Diffusion Actuator**. The **LCD Display** board connects to its board through **50-pin Transistor-Transistor Logic (TTL)** and the board connects to the **Raspberry Pi** through **High-Definition Multimedia Interface (HDMI)**. The **Speakers** also connect to the **LCD Display** board through **JST PH2.0**. Beyond being powered up, the **Fragrance Diffusion Actuator** also connects to the **Raspberry Pi** through the **General Purpose Input/Output (GPIO)**. What also connects to the **Raspberry Pi** through **GPIO** are the **Motion Detection** sensors. Lastly, the **Camera** connects through **Camera Serial Interface (CSI)** to the **Raspberry Pi**.

1.1.2. Main Controller

The main controller was also previously mentioned because it makes part of one of the requirements of this project: use the Raspberry Pi 4B (Fig. 1.2). This System-on-a-Chip (SoC) has several of specifications [1]:

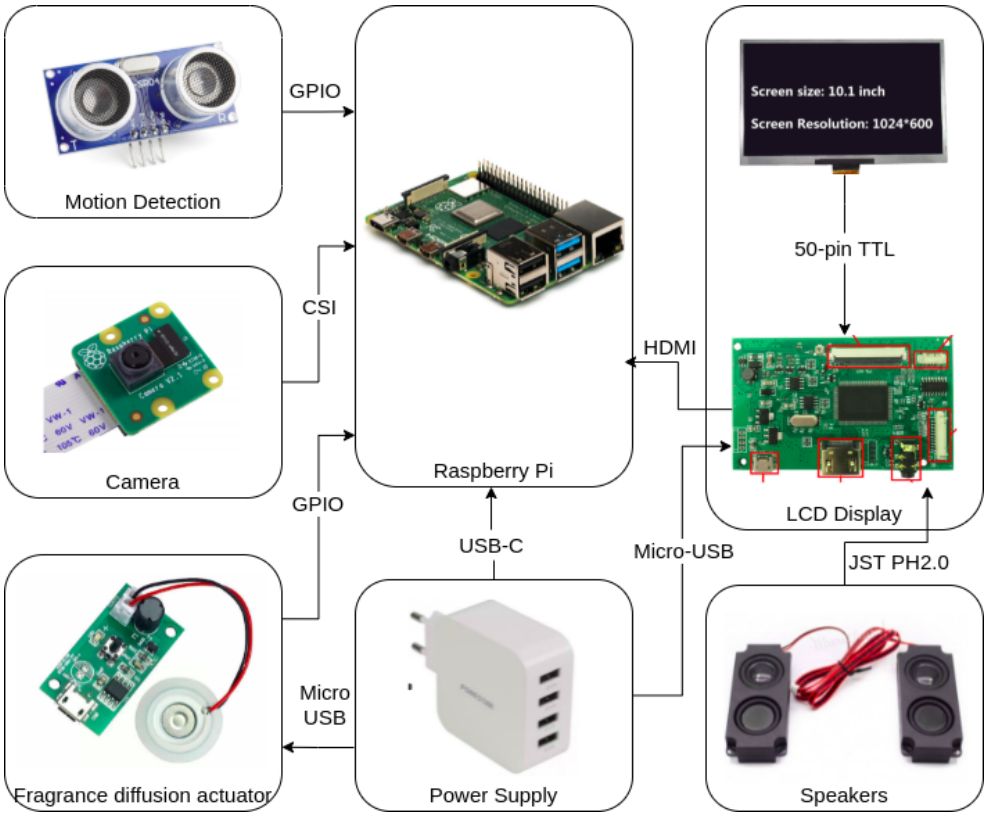

Figure 1.1.: HW architecture Block Diagram

- Processor: it has the Broadcom BCM2711 processor, quad-core Cortex-A72 (ARM v8) 64-bit with 1.5GHz;
- Memory: this model has 4GB LPDDR4 with on-die ECC;
- Connectivity: it has a 2.4 GHz and 5.0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0 with low energy, one Gigabit Ethernet port, four USB ports in which two are 3.0 and another two are 2.0;
- GPIO: it has a standard 40-pin GPIO header that is fully backwards-compatible with previous boards;
- Video and Sound: it has two HDMI ports that support up to 4kp60, a 2-lane MIPI DSI display port, a 2-lane MIPI CSI camera port and a 4-pole stereo audio and composite video port;
- Multimedia: H.265 (4Kp60 decode), H.264(1080p60 decode and 1080p30 encode) and OpenGL ES 3.0 graphics;
- SD card support: Micro SD card slot for loading operating system and data storage;
- Input power: it has 5V Direct Current (DC) via USB-C connector (minimum 3A), a 5V DC via GPIO header (minimum 3A) and Power over Ethernet (PoE) - enabled (requires separate PoE HAT);
- Environment: it has a range of operation between 0°C and 50°C.

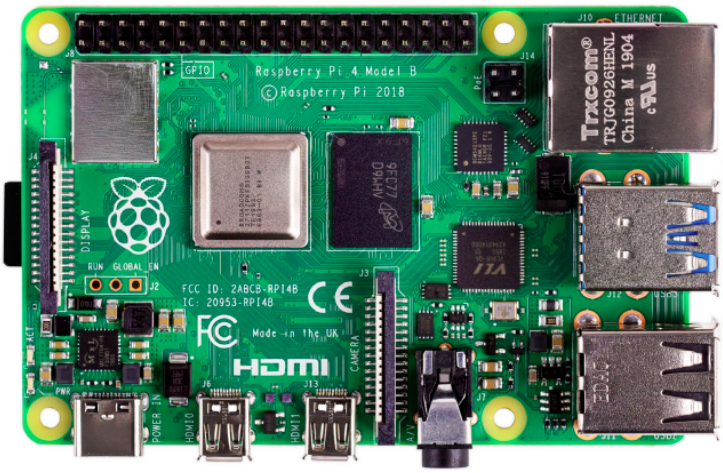

Figure 1.2.: Raspberry Pi model 4B

SD Card

Since this Raspberry supports SD Card, it will be used a micro SD Card with 16 GB that will store everything that is necessary to run the system and handle it.

1.1.3. Motion Detection

For the motion detection, it was already mentioned in section **??** that the best option is to use an ultrasonic sensor. Thus, the sensor that has been chosen is the **HC-SR04 Ultrasonic Sensor**. This sensor has the following specifications [2]:

- Operating Voltage: 5V DC;
- Operating Current: 15 milliampere (mA);
- Operating Frequency: 40 KHz;
- Maximum Range: 4 meters;
- Minimum Range: 2 centimeters;
- Ranging Accuracy: 3 millimeters;
- Measuring Angle: 15 degrees;
- Trigger Input Signal: 10 microseconds TTL pulse;
- Dimension: 45 x 20 x 15 millimeters.

Sensor Pinout

In Fig. 1.3 is described the sensor pinout and each pin works as follows [2]:

1. is the power supply for HC-SR04 Ultrasonic distance sensor which we connect to a 5V supply (for example, 5V pin on Raspberry);
2. pin that is used to trigger the ultrasonic sound pulses;
3. pin that produces a pulse when the reflected signal is received. The length of the pulse is proportional to the time it took for the transmitted signal to be detected.
4. pin that should be connected to the ground (for example, GND pin of the Raspberry).

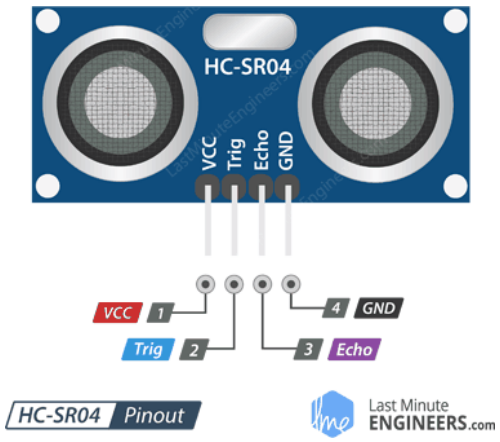

Figure 1.3.: HC-SR04 Pinout (withdrawn from [2])

For this project it will be used three sensors, one placed on the bottom, another on the middle and another one on the top of the machine. With this setup, it can be avoided some perturbations like an animal walking in front of the machine (only the bottom sensor will detect). The disposition of the middle and top sensors can't be too high, because of short people, for example.

1.1.4. Fragrance Diffusion Actuator

The chosen fragrance diffusion actuator is in Fig. 1.4 and has the following specifications [3]:

It has an operating voltage of **5V DC** and an operating current of **300 mA**. The operating power is **2 Watt**. It has a fixed frequency single-chip microcomputer with a frequency of **108 KHz**. The dimensions of the board of the module are **35 * 20 * 17 millimeters**. It has a **strong versatility**, large amount of fog, **stable performance**, the chip has an automatic timing shutdown function (4 hours of continuous work will automatically shut down protection, to turn on again, press the power on again). The 5V USB power supply mode, can be powered by MICRO charging cable. The net diameter of the atomized steel sheet is 16 millimeters, the outer diameter of the silicone ring is 20 millimeters, and the wire length is 8 centimeters.

Figure 1.4.: Fragrance module (withdrawn from [3])

1.1.5. Camera

The camera to use in this project needs to be compatible with the board in use, in this case, the Raspberry Pi. Thus the camera module that is used is the **Raspberry Pi Camera Module V2** (Fig. 1.5).

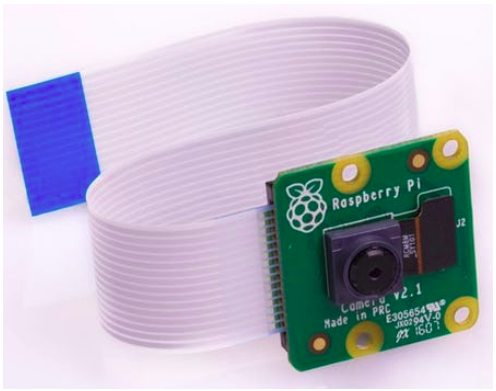

Figure 1.5.: Camera module (withdrawn from [4])

This camera module has a **Sony IMX219 8-megapixel sensor** and can be used to take high-definition video, as well as stills photographs. It supports 1080p30, 720p60 and VGA90 video modes, as well as still capture. It attaches via a 15 centimeters ribbon cable to the CSI port on the Raspberry Pi. The camera works with all models of Raspberry Pi 1, 2, 3 and 4. It can be accessed through the MMAL and V4L APIs, and there are numerous third-party libraries built for it, including the Picamera Python library. [4].

1.1.6. LCD Display

In Fig. 1.6 is the display that is used in this project. One advantage on this display is that it has **audio drivers**, which means that it is only necessary to plug a speaker and the board can handle the rest. It is also important to refer that the display isn't touch because there is no need to it and also this was the chosen one because it was the bigger and best on market considering quality and price.

As it can be seen, the display has **10.1 inches** and it is supplied with **5V DC** and with a current of **2 A** via a micro USB port. Fig. 1.7 show all the interfaces that the board module of the display provides. That was also one more reason for the choice of this display: it has an **HDMI interface** to connect to the Raspberry, the **50Pin TTL Screen Interface** that will connect to the display and two options to plug audio - the **Speaker Interface** and the **3.5mm audio interface**.

Figure 1.6.: Display (withdrawn from [5])

It can also be seen in this figure that the display has also a remote and a board to handle the remote controls, but in this implementation, it will probably not be in use.

1.1.7. Speakers

When playing video ads, it is not only necessary a display, but also a speaker to playback the sound of the ads. As it can be seen in Fig. 1.7, the screen board has two different interfaces of audio: the speaker interface and the audio interface. For this project it will be used speakers that uses the speaker interface, because that type of speakers with that interface are passive speakers and don't need a DC power supply, which is an advantage [6].

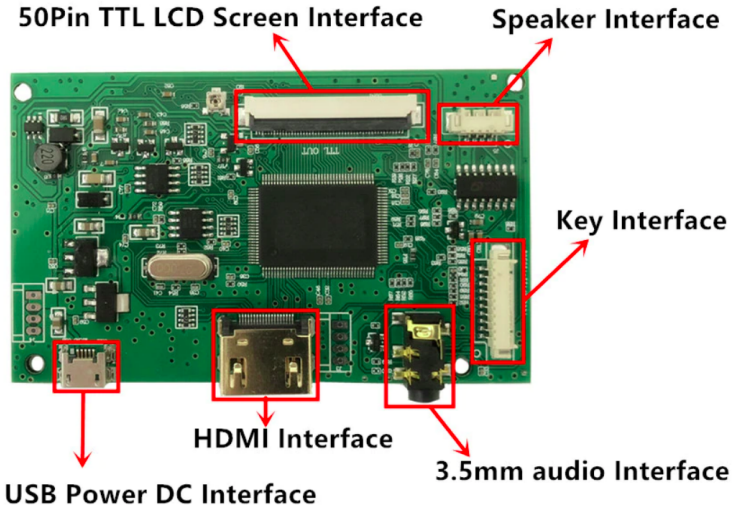

Figure 1.7.: Display Interfaces (withdrawn from [5])

Thus, the speakers that will be used have an impedance of 8 Ohm and a power of 5 Watts and are displayed on Fig 1.8.

Figure 1.8.: Speakers (withdrawn from [7])

1.1.8. Power Supply

The MDO Local System (MDO-L) will be a plugged in system, so it will be needed a plugged in power supply to supply all the components of the system. In total, the power consumption will not overtake 20 Watts and all the supplies necessary are plugged by USB (Raspberry Pi, Fragrance Diffusion Actuator, and screen). So, it will be used the power supply in Fig. 1.9, that has 4 outputs of 5 V DC and 2.4 A DC each.

Figure 1.9.: Power Supply (withdrawn from [8])

1.1.9. On/Off button

In this context, an On/Off button can be useful to power On/Off the MDO-L. However, this feature is not that necessary, so, this prototype will be only powered through the power supply previously talked in section 1.1.8 and will be always on until the power supply be unplugged, powering off the machine.

1.1.10. Total HW cost

The total HW cost can now be precisely calculated, once that all the hardware is now specified on table 1.1, yielding about 175 EUR.

Table 1.1.: Total spending on Hardware

Item	Quantity	Price (€)
Raspberry Pi 4B	1	70.00
Ultrasonic Sensor HC-SR04	3	11.70
Fragrance Diffusion Actuator	1	3.23
Raspberry Pi Camera V2	1	20.00
LCD Display	1	51.95
Speakers	1	3.00
Power Supply	1	13.50
	Total	**173.38**

1.2. Hardware interfaces definition

After specifying the HW, it is important to define its interfaces. Firstly, it is necessary to have basic know-how of the main board's pinout. In Fig. 1.10 is represented the pinout of the Raspberry Pi.

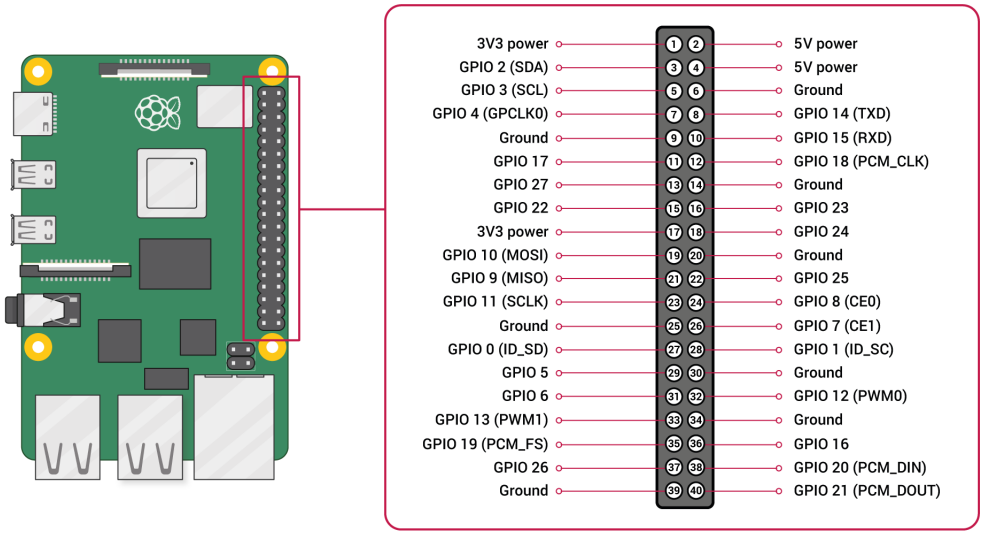

Figure 1.10.: Power Supply (withdrawn from [9])

1.2.1. Peripherals Mapping

After have a basic know-how of the Raspberry's pinout, it is now possible to map all the HW components. As it can be seen in Fig. 1.11, only two types of peripherals need to be mapped in the Raspberry: the **ultrasonic sensor** and the **fragrance diffusion actuator**.

Firstly, the **ultrasonic sensors** can have three of their four pins in common: the **Vcc**, the **GND** and the **Trig**. The first two are for obvious reasons: they can be powered for the same source, so they are connected to the **pin 2 (5V power)** and **pin 9 (Ground)**, respectively. The last one is because the trigger only triggers the sensors to start the acquisition, so, they can all start at the same time and have the same trigger source, so, they are all connected to the **pin 11 (GPIO 17)**. Then, each one of them need a specific pin to connect to its **Echo** in order to make the distance read, so, the pins that are chosen are **pin 15 (GPIO 22)**, **pin 16 (GPIO 23)** and **pin 18 (GPIO 24)**.

Lastly, the **fragrance diffusion actuator**. Although this one is powered up by Micro-USB, it is necessary to activate or deactivate the diffusion and this is only possible with a **Metal Oxide Semiconductor Field-Effect**

Transistor (MOSFET), activating its gate. The gate is connected to the **pin 36 (GPIO 16)** and this is the pin responsible to activate or deactivate the fragrance diffusion.

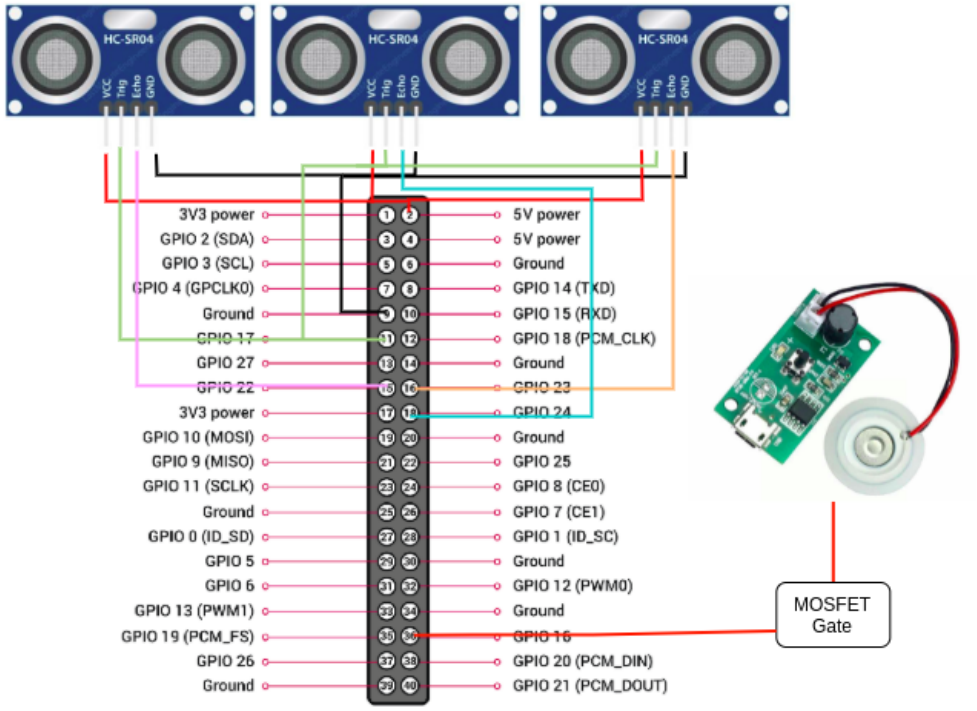

Figure 1.11.: Peripheral Mapping

To turn things more clear, Table 1.2 shows all the pin mapping in a more clear way.

1.2.2. Test Cases

In order to verify if all the hardware is in good conditions, are made some test cases to each component. In Table 1.3 are displayed all the test cases to be done in all pieces of HW. Basically, all tests are functional tests and need to be done to know if some component may be damaged or also could not be in the error range provided by the manufacturer.

1.2.3. PCB Design

As it can be seen in Fig: 1.11, it is a little bit complex and messy to look and conceive a circuit with all those connections, and since many connections are in common, it is an advantage to design a PCB to make things more simpler.

Table 1.2.: Pin Mapping

Pin Mapping		
Controller Pin	**Interface Device Pin**	**Function**
Pin 2 (5V)	Ultrasonic Sensors (Vcc)	Supply 5V to the sensors
Pin 9 (Ground)	Ultrasonic Sensors (GND)	Close the supply connection
Pin 11 (GPIO 17)	Ultrasonic Sensors (Trig)	Make pulses to the Trigger's sensors in order to start the distance acquisition
Pin 15 (GPIO 22)	Ultrasonic Sensor 1 (Echo)	Handle the Echo Pin of the first sensor in order to measure the distance
Pin 16 (GPIO 23)	Ultrasonic Sensor 2 (Echo)	Handle the Echo Pin of the second sensor in order to measure the distance
Pin 18 (GPIO 24)	Ultrasonic Sensor 3 (Echo)	Handle the Echo Pin of the third sensor in order to measure the distance
Pin 36 (GPIO 16)	MOSFET's Gate	Toggle the gate of the MOSFET in order to turn On/Off the diffusion actuator

For the design of this PCB it was used the software **PADS**, developed by Mentor Graphics. In Fig. 1.12 is the schematic of the connections to the PCB, this part was made on **PADS Logic** and basically is joining everything that is common to the sensors to then connect to the raspberry Pi, then there's a pin for every Trigger of every sensor and finally a pin for the gate of the MOSFET and two pins for its supply.

When the logic is done, it is moved to the layout on **PADS Layout** where all the components are placed and connected through tracks. In Fig 1.13 is the final layout of the PCB. Note that there are two tracks that are more wide, this is because they are the tracks that supply the MOSFET and this one need more current, which means wider tracks.

Connectors J1, J2 and J3 are for the sensors, connector J4 is for the GPIO of the Raspberry Pi, connector J5 supplies the sensors and connector J6 supplies the MOSFET Q1.

It is to note that the final size of the PCB is **2400 * 2200 mils**. Converting this to centimeters gives a dimension of approximately **6.1 * 5.6 centimeters**.

1.3. Software specification

Next, the SW responsible for system operation is specified for all subsystems — **MDO Remote Client (MDO-RC)**, **MDO Remote Server (MDO-RS)**, and **MDO Local System (MDO-L)**. All these subsystems are event-driven (asynchronous), and they can be more easily specified using state-machine diagrams, previously illustrated in the analysis phase (Section **??**). Also in the analysis phase, the use case diagrams helped to identify the main features required for the system and the respective sequence diagrams helped

Table 1.3.: Hardware Test Cases

HW Component	Type of test	Description	Expected Result
Ultrasonic Sensor	Functional	One will connect the ultrasonic sensor to the Raspberry Pi. Then, an object will be approximated to the sensor from several distances with the corresponding distance being measured with a measuring tape.	If the distance measured by the sensor and the measuring device are within the error margin provided by the manufacturer, the device is compliant.
Camera	Functional	One will connect the camera to the Raspberry Pi and then will try to make image acquisition in real time, take pictures and also try to take some gifs and videos.	If the quality of the image and the image acquisition are in good quality according to the manufacturer's information, then the device is in good conditions of use.
Fragrance Diffusion Actuator	Functional	One will connect a MOSFET to the fragrance diffusion and connect its gate to the Raspberry Pi. Then, one will trigger the MOSFET's gate and watch the actuator's behavior.	If the diffusion actuator diffuses in good state the fragrance and also if the MOSFET doesn't overheat, then the set of HW is in good conditions of use.
Power Supply	Functional	One will connect the three components (LCD, Fragrance Diffusion Actuator and Raspberry Pi) to the power supply and see their behavior.	If all components work properly, then this piece of HW is in good conditions of use.
Speakers	Functional	One will connect the Speakers to the LCD board and try to run a video with sound in order to verify the sound provided by the speakers.	If the sound output matches the manufacturer's characteristics of the product, then this set of HW is compliant.
LCD Display	Functional	One will connect the LCD board through HDMI to the Raspberry Pi, the LCD Display to its board and supply the board and then try to send some images and videos to the LCD Display.	If the images and videos are displayed in good quality and are in match with the manufacturer's information, then this set of HW is in good conditions of use.

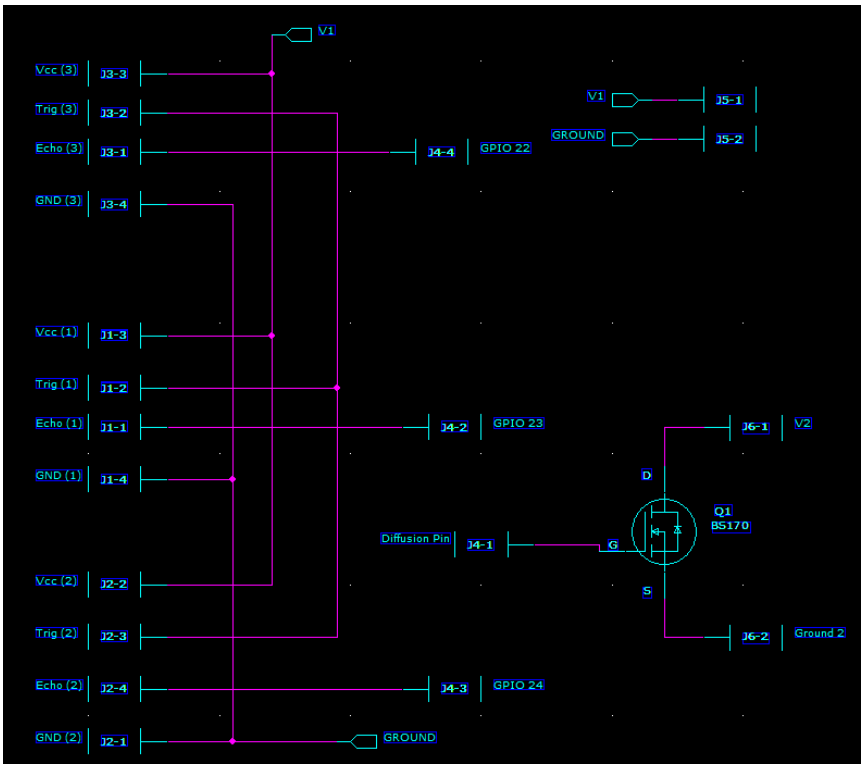

Figure 1.12.: PCB schematic

to clarify the intervening objects and the interaction among them.

In this section, the analysis phase information is used to derive the static architecture of the system — classes diagram — and to specify algorithms for its implementation through flowcharts, keeping in mind that the several subsystems operate multiple tasks concurrently, thus requiring the tasks' specification and its priorities. The data frame formats are specified for communication between the different modules. The Entities-Relationships diagram (ERD) are depicted to design the required databases and the User Interface (UI) mock-ups are recalled. The test cases for each subsystem are listed, defining its operation and the expected result. The Commercial off-the-shelf (COTS) SW and the third-party libraries are identified and a mapping between class topics and the foreseeable implementation is presented for clarification. Finally, the SW tools are listed.

1.3.1. Software architecture

The system's SW architecture was devised using Unified Modeling Language (UML) component diagrams for **Remote Client** (Fig. 1.14), **Remote Server** (Fig. 1.15), and **Local System** (Fig. 1.16). Each component diagram illustrates all SW components for the system in analysis and the interaction between them, and its

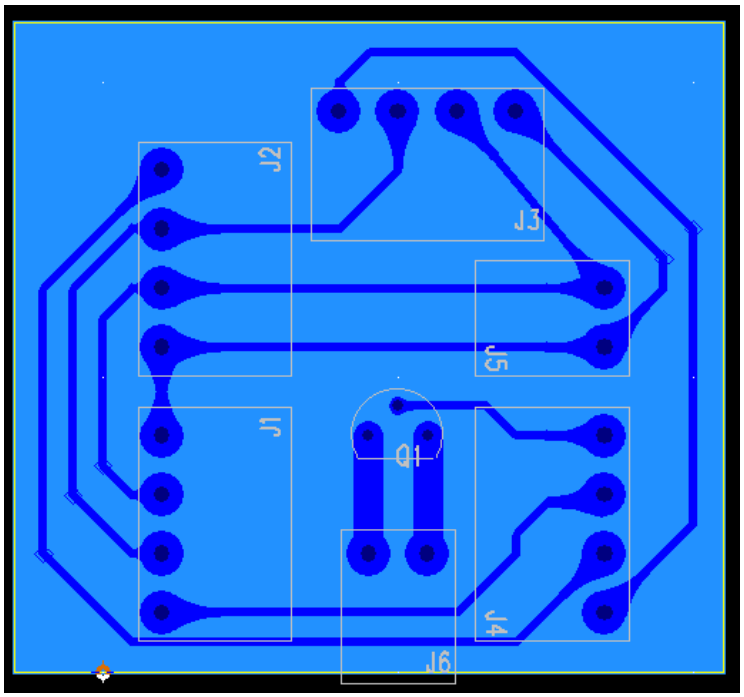

Figure 1.13.: PCB layout

interfaces with external subsystems.

Remote Client

Fig. 1.14 depicts the **Remote Client** SW architecture, encapsulated in the package **MDO-RC: AppManager**, and is comprised of the following artifacts:

- User Interface package: contains the **UI** and **UI Engine**. It is responsible for providing user feedback and capturing UI events which drive the **Remote Client**'s logic.
- Comm Manager package: manages incoming and outgoing connections to the **Remote Server** (package **MDO-RS: App Manager**), periodically checking the connection status by pinging the **Remote Server**. All connections consist of Transmission Control Protocol/Internet Protocol (TCP/IP) sockets.
- DB Manager package: manages the queries and the associated responses by building or parsing them, respectively.
- Remote Controller package: contains the **Cmd Parser**, which parses the command responses received from the **Remote Server** when the **Admin** is performing remote control of the **Local System**.
- RC Rx Parser component: high-level parser which filters between **db responses** and **cmd responses** for appropriate dispatching.

- TCP/IP Tx socket: outgoing connection node, through which **tx frames** are sent to the **Remote Server**.
- TCP/IP Rx socket: incoming connection node, through which **rx frames** are received from the **Remote Server**.

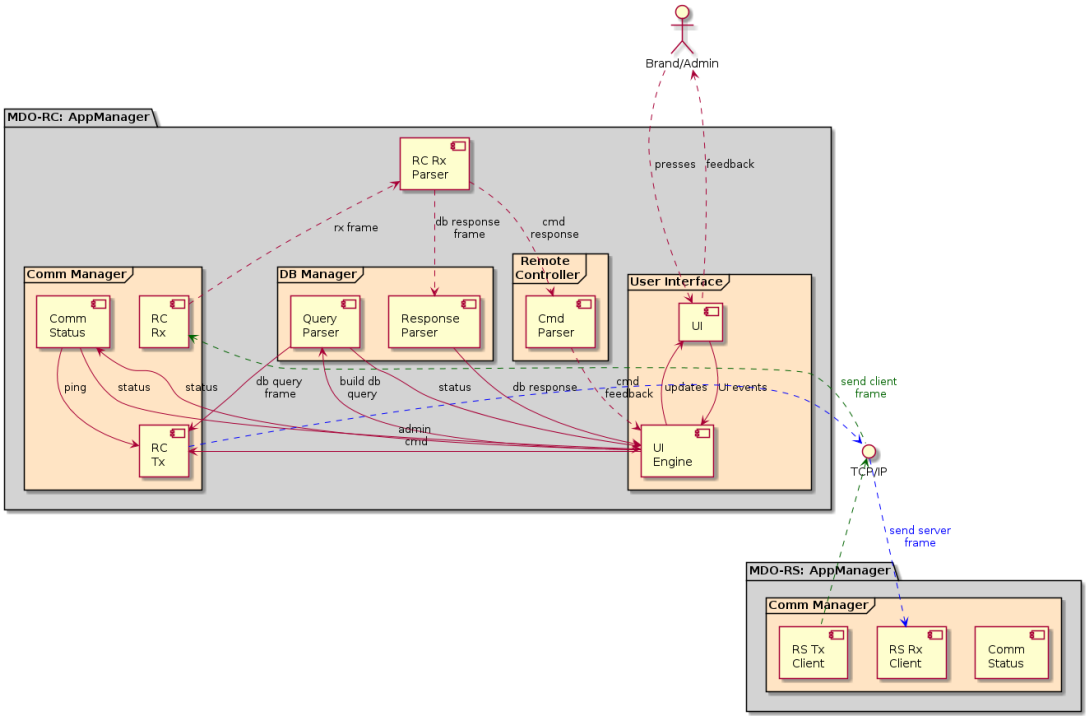

Figure 1.14.: SW architecture: component diagram — **Remote Client**

Remote Server

Fig. 1.15 depicts the **Remote Server** SW architecture, encapsulated in the package MDO-RS: AppManager. It interacts with the **Remote Client** (package **MDO-RC: AppManager**), with the **Local System** (package **MDO-L: AppManager**), and with the **Database (DB) server** (**MDO-RS: DB Server**). The database management is done using client-server architecture, with **MDO-RS: AppManager** containing the DB client and **MDO-RS: DB Server** the server.

The MDO-RS: AppManager package is comprised of the following artifacts:

- Command–Line Interface package: contains the **Command Line Interface (CLI)**, the **CLI Engine**, and the **CLI Parser**. It provides the server external interface for clients to perform requests, capturing the events which drive the **Remote Server**'s logic.

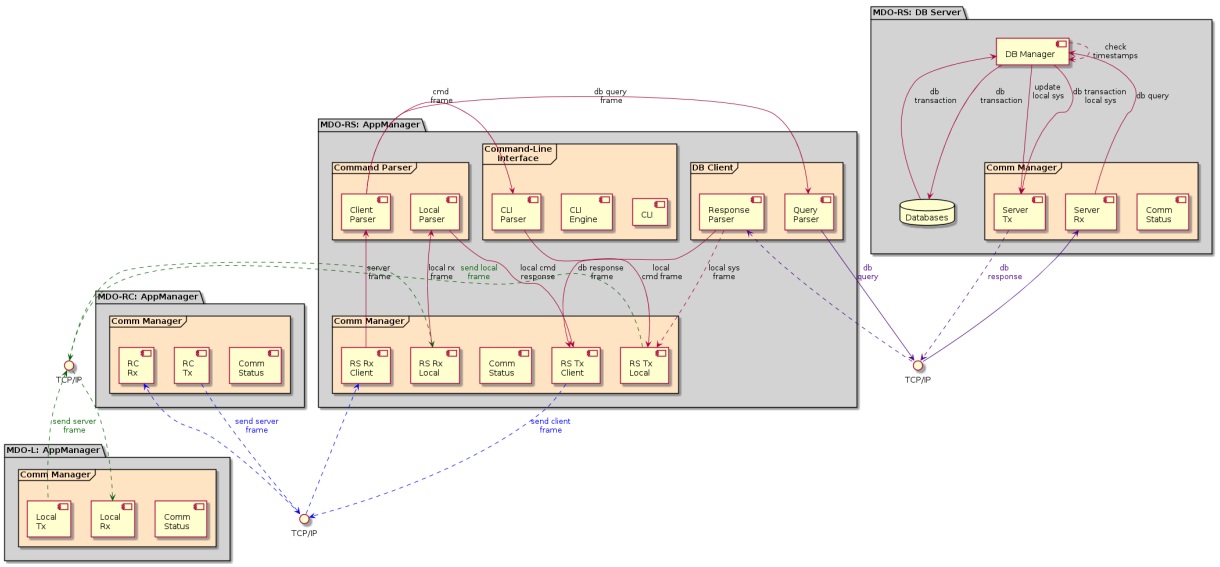

Figure 1.15.: SW architecture: component diagram — **Remote Server**

- Comm Manager package: manages incoming and outgoing connections to the **Remote Client** (package **MDO-RC: App Manager**), and each **Local System** (package **MDO-L: AppManager**) it needs to interact. It periodically checks all connections statuses. All connections consist of TCP/IP sockets.
- DB Client package: manages the queries and the associated responses by building or parsing them, respectively. It performs the requests for **DB server** with the solicited queries.
- Command Parser package: contains the **Client Parser**, and the **Local Parser** to parse and handle frames received from the **Remote Client** or **Local System**, respectively, forwarding it for appropriate dispatching.
- TCP/IP sockets: incoming/outgoing connection nodes, through which incoming or outgoing traffic flows for the **Remote Client**, the **Local System** and the **DB Server**.

The MDO-RS: DB Server package is comprised of the following artifacts:

- Comm Manager package: manages incoming and outgoing connections to the **DB Client** (in package **MDO-RS: App Manager**). It periodically checks all connections statuses. All connections consist of TCP/IP sockets.
- DB Manager package: handles the received queries, issuing transactions for the databases and returns its response. It is also responsible for periodically checking timestamps, and when there is a match, update the local system with the relevant information.
- Databases: contains the actual data stored.

Local System

Fig. 1.16 depicts the **Local System** SW architecture, encapsulated in the package MDO-L: AppManager.

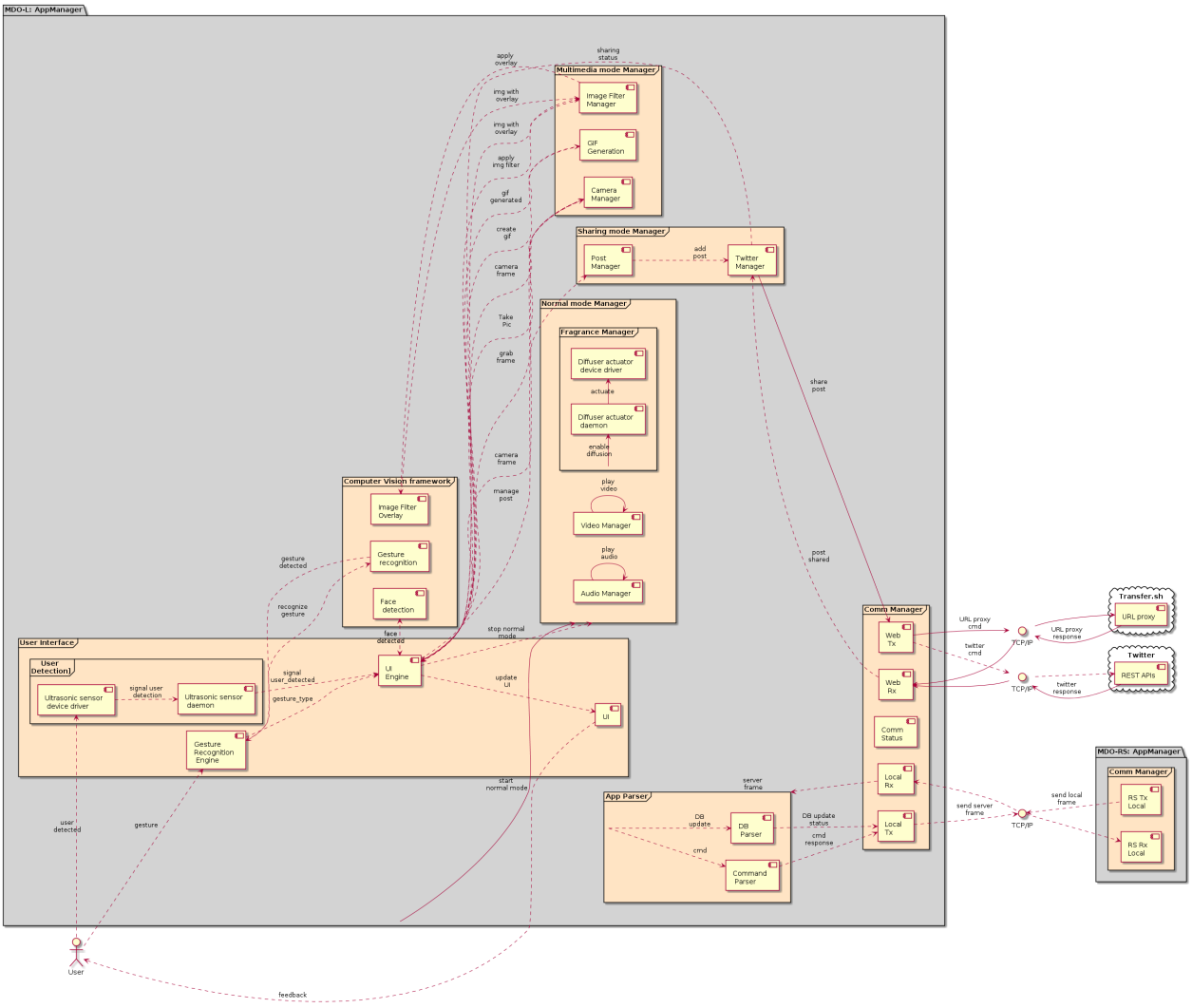

Figure 1.16.: SW architecture: component diagram — **Local system**

It interacts with:

- Remote Server (package **MDO-RS: AppManager**): to retrieve updates on its operation or **Admin** commands
- Twitter (via its **Representation State Transfer (REST) Application Programming Interfaces (APIs)**): to share posts on it
- **transfer.sh** — an Uniform Resource Locator (URL) proxy server: to ease file transfer between the

Remote Server and the Local System.

The MDO-RS: AppManager package is comprised of the following artifacts:

- User Interface package: contains the **UI**, the **UI Engine**, the **Gesture Recognition Engine**, and the **User Detection package**. The **UI** provides user feedback and **UI Engine** captures the events that drive the **Local System**'s logic. When a user approaches the **Local System**, the **Ultrasonic sensor device driver** captures this event and passes it to the user space where the **Ultrasonic sensor daemon** logs it, which, in turn, signals this event to the **UI Engine**. When the **User** performs gestures, the **Gesture Recognition Engine** requires a service to the **Gesture recognition** component (**Computer Vision Framework**) to recognize the gesture. Also, when the **User** is detected, the **UI Engine** requests the **Normal mode manager** to stop running, and requests **Face detection** to track people's faces in the camera.
- Comm Manager package: manages incoming and outgoing connections to the **Remote Server** (package **MDO-RS: App Manager**), and the internet for each web service it needs to interact (**Twitter** and **transfer.sh**). It periodically checks all connections statuses. All connections consist of TCP/IP sockets.
- Computer Vision framework package: manages the computer vision related tasks, namely, gesture recognition, face detection, and image filter overlay.
- Multimedia Mode Manager package: manages the multimedia mode related tasks, namely, image filtering, Graphics Interchange Format (GIF) generation, and camera management.
- Sharing Mode Manager package: manages the sharing mode related tasks, namely, post management, and social media management, in this case, **Twitter**.
- Normal Mode Manager package: manages the normal mode related tasks, namely, fragrance diffusion, video and audio outputs. The fragrance diffusion is requested to the device driver using a daemon to bridge user-space and kernel-space.
- App Parser package: manages the parsing for database queries and requested commands.
- TCP/IP sockets: incoming/outgoing connection nodes, through which incoming or outgoing traffic flows for the **Remote Server**, and the web services for **Twitter** and the **URL proxy server**.

1.3.2. Deployment specification

The deployment specification maps the software artifacts derived in the component diagrams to the respective computational node where it will be executed. This step pertains to the implementation phase; nonetheless, it is illustrated here as it clarifies the SW architecture, as well as it guides some design decisions.

Fig. 1.17 depicts the deployment diagram for the MDO system. In blue are represented the computational nodes of interest, namely:

- Linux PC: it is the host device, where are executed the packages (in grey) **MDO-RC: AppManager**, the **MDO-RS: AppManager**, and the **MDO-RS: DB Server**. All connections between packages in the **Linux PC** and **Raspberry Pi** are TCP/IP sockets in a client-server architecture. The design and subsequent implementation of these packages are not limited by the embedded device constraints, thus, the focus is on functionality and ease of implementation. It is important to note the design is decoupled, i.e., the **Remote Client** and **Remote Server** can be executed in different computational nodes, where, the former runs on a desktop computer and the latter on a cloud-based service or mainframe, thanks to the client-server architecture based on TCP/IP sockets. Thus, for practicality reasons, both subsystems are implemented on the same **Linux PC**, but it should be straightforward to migrate them to different and distinct platforms.
- Raspberry Pi: it is the embedded device, where is executed the package **MDO-L: AppManager**. It interacts with **Remote Server** and **Web services — Twitter** and **Transfer.sh** — using TCP/IP sockets in a client-server architecture. The design and subsequent implementation of this package is limited by the embedded device constraints, thus, the focus in on functionality and performance.
- Web: it represents the remote computational nodes where the **Web services — Twitter** and **Transfer.sh** are executed. The **Local System** interacts with these services using TCP/IP sockets in a client-server architecture through the exposed APIs.

1.3.3. Database design

As aforementioned in Section **??**, the conceptual design of a database consists of a high-level description of the data to be stored in the database and its constraints, which is usually carried out using the Entity-Relationship (ER) model. Thus, in Section **??** the key concepts of the ER model are presented, which can be reviewed to assist the comprehension of the subsequent database design produced.

Fig. 1.18 depicts the Entities-Relationships diagram (ERD) for the conceptual design of the MDO system's database, with the primary keys shown in red and the foreign keys in blue, comprising the relevant entities and its relationships, namely:

- User: models an User in the application (**Brand** or **Admin**). It has a **role**, a **name**, an **email**, and a **pass** (which is encrypted before being stored). An **User** manages **0** or many **UserStations** and owns **0** or many **Ad**s.
- UserStations: it unfolds the relationship many-to-many between an **User** and a **Station**. An **User** manages **0** or many **UserStations**, and a **UserStations** contains one or many **Station**s. It imports the

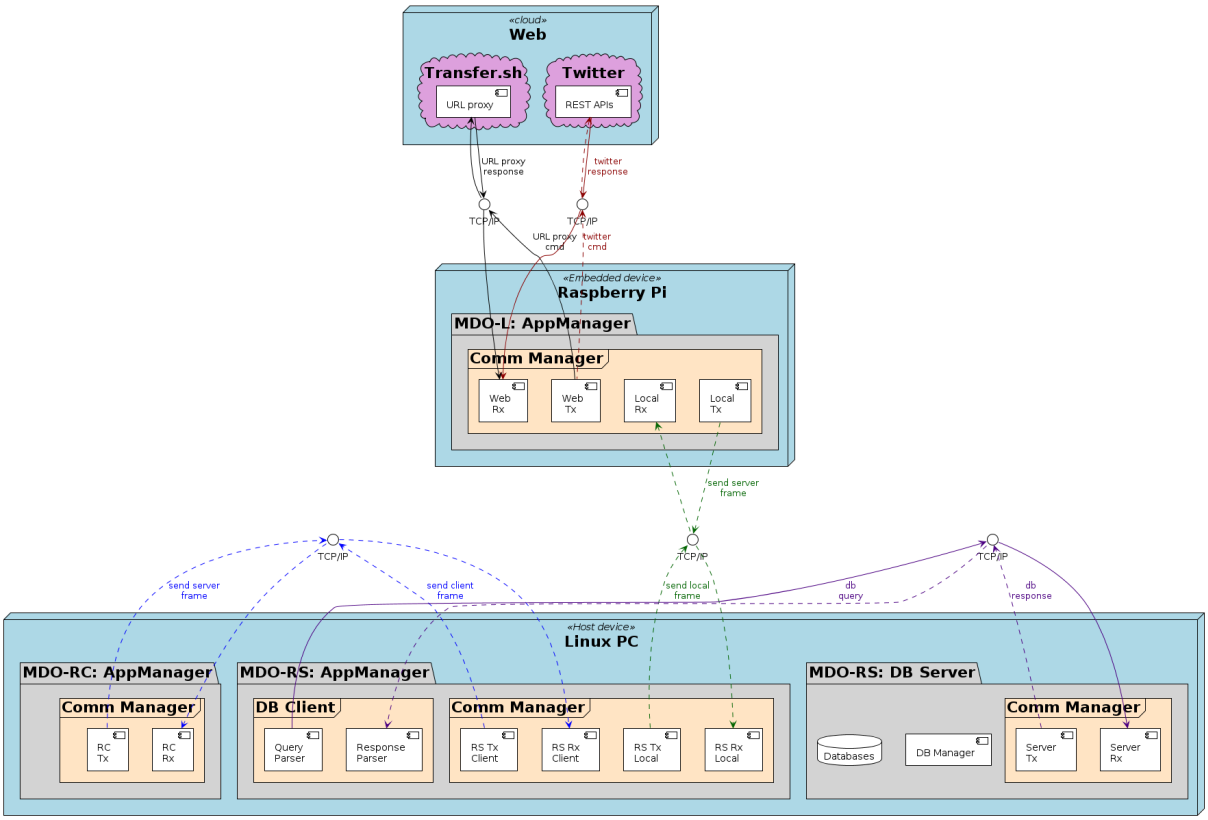

Figure 1.17.: Deployment diagram

User.id as a foreign key to enable referencing an **User**.

- Station: it models a physical MDO station. It contains an **id**, a **name**, a **location** and an **IP** address. It imports the **UserStations.id** as a foreign key to enable referencing an **UserStations**.
- TimeTable: it models a weekly timetable that every **Station** contains. It contains one or many **TimeSlots**. It imports the **Station.id** as a foreign key to enable referencing a **Station**.
- TimeSlot: it models a slot of time available for ads reproduction. It contains the **duration** (in minutes), the **cost**, and a flag signaling if it is **rented** or not. It imports the **TimeTable.id** as a foreign key to enable referencing a **TimeTable**.
- Ad: it models an advertisement. It imports the **User.id** as a foreign key to reference an **User**, a **Fragrance.id** to reference the associated **Fragrance**, and a **TimeSlot.id** to reference a **TimeSlot**. The **Fragrance** is optional, but, as the foreign key must not contain a null value, a default value of **0** is used to signal that no fragrance is used.
- MediaFile: it models a media file that must be attached to the **Ad**. It contains an **id**, file specifications (**filename**, **filesize**, and **filetype**), **mdata** for storing the file and an optional **description**. It imports the **Ad.id** as a foreign key to enable referencing an **Ad**.

- Fragrance: it models a **Fragrance**. It is added to the database by the **Admin** and it can be selected by the **Brand** from the **FragranceList** available for each **Station**. It contains an **id**, a **name**, an **intensity** to define the actuation time, a maximum capacity (**vol_ml_max**) and a current capacity (**vol_ml_level**), and an optional **description**. It imports the **FragranceList.id** as a foreign key to reference the **FragranceList** available for each **Station**.
- FragranceList: it represents the list of fragrances available for each **Station**. It imports the **Station.id** as a foreign key to reference its associated **Station**.

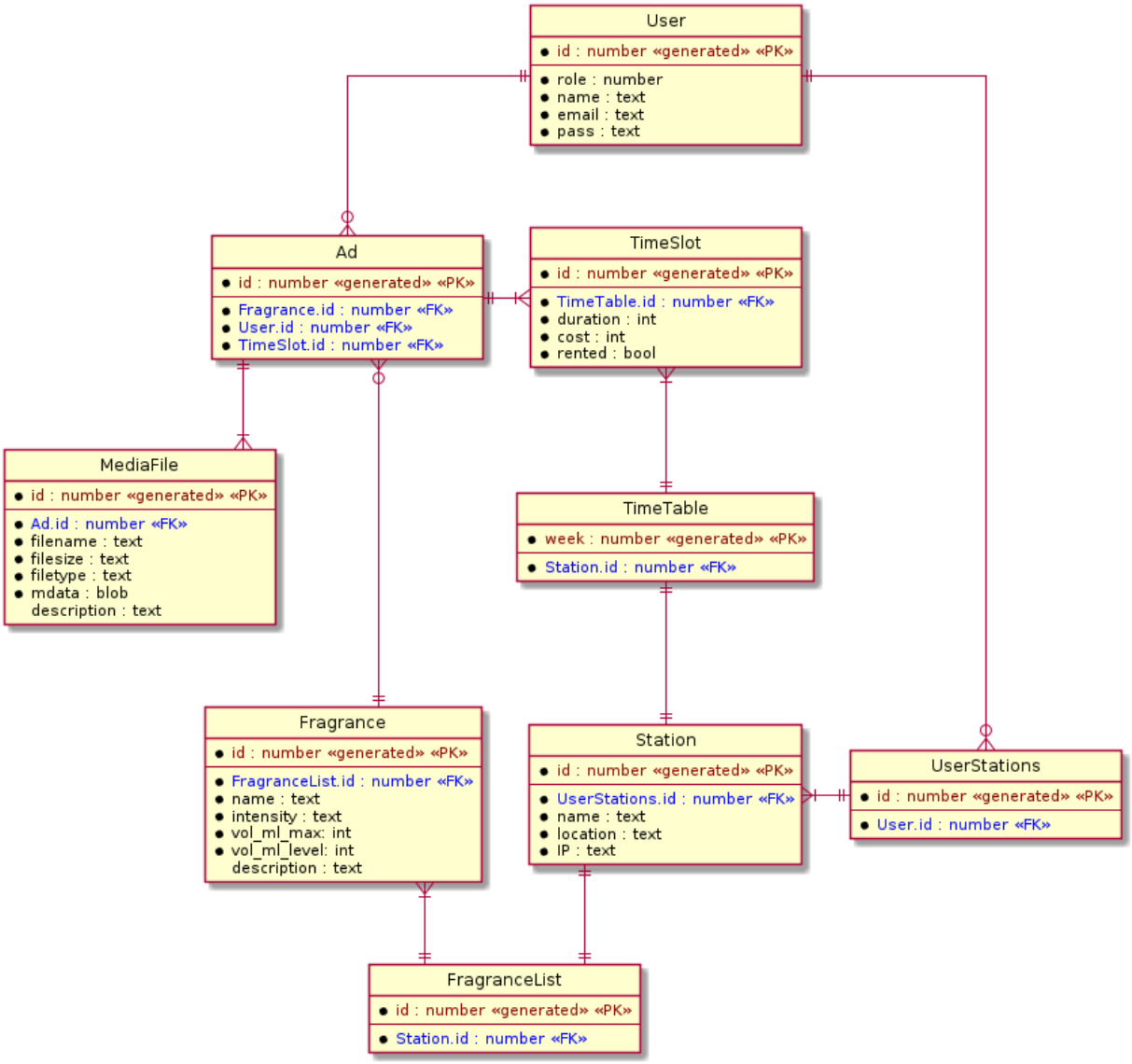

Figure 1.18.: MDO Entity-Relationship Diagram: conceptual design

1.3.4. Data formats

After devising the SW architecture the data flow across the several subsystems became clearer, with data frames being conveyed through TCP/IP sockets. Thus, it is important to define the data formats for internal representation and communication between the subsystems, as illustrated in Fig. 1.19. In gray are depicted the several subsystems, in white the data frames and in blue the legend.

The several subsystems will communicate using a basic data frame — **rxFrame** — which contains a **header**, the data length, the actual data, and an **ACK** (acknowledge) signal. The **header** defines the frame type which can be **DB** (database queries), **CMD** (commands sent to be executed), and **AD** (ads to be reproduced — only for **Local System**). It is important to note: the usefulness of **data**'s datatype being **void ***, as it is generic, and thus enables the encapsulation of other data frames — e.g., the **AdFrame** is encapsulated within a **rxFrame**; the **ACK** signal is a valid terminator for all data frames — helping to identify bigger data frames than the maximum ones, and to prevent ill-intentioned data frames usage to corrupt the system — of type **int** to prevent collisions with normal messages sent.

The **Remote Server** contains an additional data frame — **serverFrame** — which encapsulates a received **rxFrame** and pairs it with the sender socket descriptor information to enable appropriate message routing.

The **AdFrame** contains the information about the advertisement to be reproduced, namely its type, intensity, start time, duration, media length, and media URLs. The first identifies the fragrance within the **Local System**, the second is used to calculate the fragrance diffusion timing, the start time is the elapsed time from the start of the day, the duration is the reproduction time of the ad (in minutes), and the **mediaURLs** contain the URLs from where the media files can be downloaded and stored in the **Local System**.

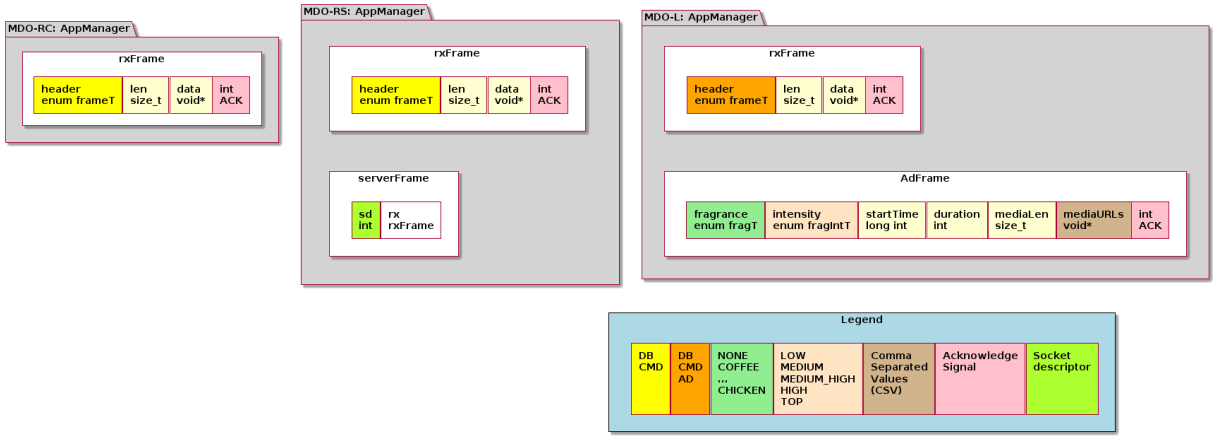

Figure 1.19.: Data formats

1.3.5. Static architecture — Class diagrams

In this section the static architecture is derived from the SW architecture, i.e., the class diagrams are outlined, using the component diagrams as a starting point, for each subsystem.

Remote Client

Fig. 1.20 depicts the class diagram for the **Remote Client** subsystem. The **UIApp** and **UIWindow** handle all the events made by the user with the UI. The **CommManager** handles the communication between this system and the **Remote Server**, which has an enumerator to classify the connection status (**ConnStatus**) and also the class **DBManager** handles all the databases through its functions and enumerators **DB_OPER** and **DB_OBJ**. Then, there's the normal classes that are mandatory to have: **User** and **Admin** that have a relation (because an Admin is a User), **Station** with its **TimeTable**, **Date** and also **Ad** to handle the ads.

There are some functions that are important to be referenced, such as **serialize()** - this function serializes all the members of the class in order to be more easier to build queries to the databases, that's why there is a function with that name in almost all classes. Also, the function **buildQuery** is important because it builds the queries to send to the databases having in mind the type of query and the table that it wants to access. In this way, it is more easier to handle all the application, with no need to save too much data unnecessarily.

Remote Server

Fig. 1.21 depicts the class diagram for the **Remote Server** subsystem. The **CLI** is the Command Line Interface and handles almost everything in this subsystem. It has several members, the more important ones are the send and receive vectors that have the type of data **serverFrame** containing the rxFrame and the socket to which the server is connected, this facilitates the work when sending and receiving because there is already the knowledge of the socket to or from the message is being received or sent. Allied to this, there's all the conditional variables and mutexes that maintain the stability of all variables and avoid collisions between classes, communications and so on.

The **CommManager** handles all the communication and has the most important functions: the **send** thread, the **receive** thread and the **connect** thread.

The **Parser** class has the finality to parse and execute all the commands that are received either to handle the database or to communicate to the remote client or the local system.

The **DBClient** is used to handle all the mechanisms of all the databases, making queries, receiving responses and so on. Finally, the **Crypt** class is to encrypt something that can be necessary, such as files, passwords or some commands.

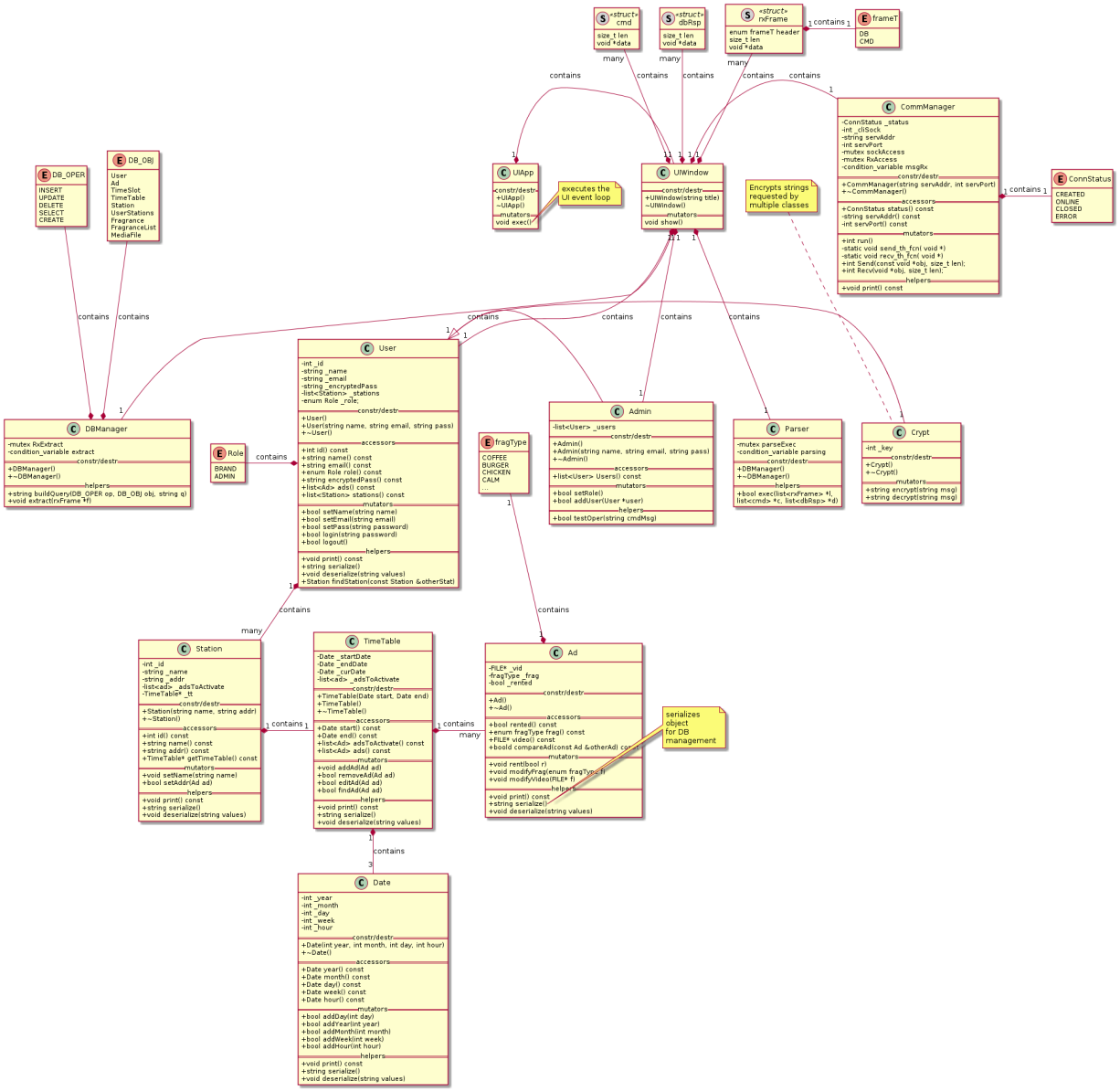

Figure 1.20.: Class diagram: **Remote Client**

Local System

Fig. 1.22 and Fig. 1.23 depicts the class diagram for the **Local System**.

1.3.6. Threads specification

In this section the threads specification are performed for the several subsystems, assigning them the respective priority. A color scheme is used to define thread priority, with red being the highest and green

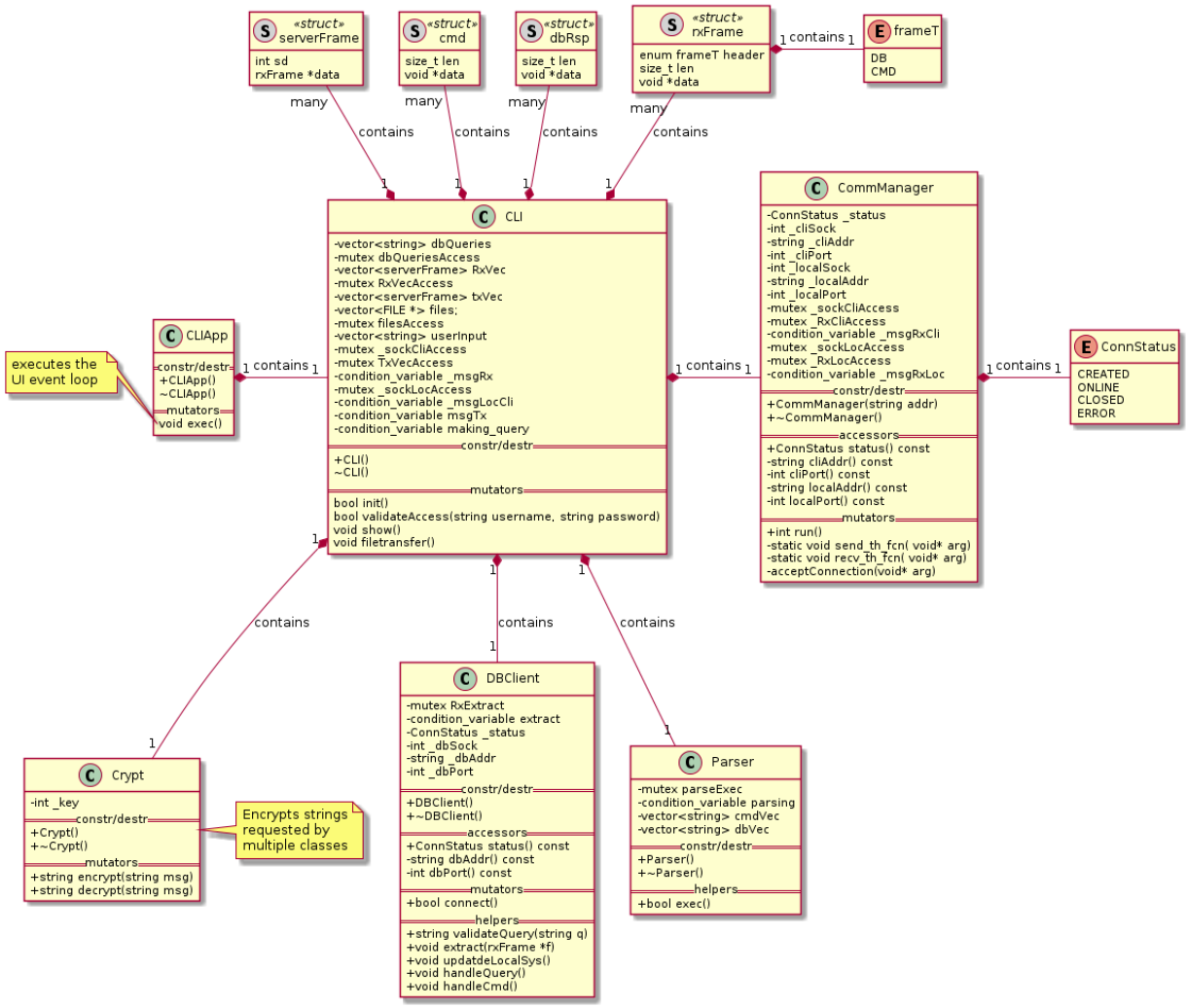

Figure 1.21.: Class diagram: Remote Server

the lowest. Additionally, the threads are ranked from highest (at the top) to lowest (at the bottom) in each frame.

Remote Server

Fig. 1.24 illustrates the thread specification and the associated priorities for the Remote Server. There are four logical groups:

1. CLI: the command line interface has a thread to transfer files because it can be necessary to transfer them to both subsystems. This thread has low priority because it is not a critical feature that need to happen as soon as possible and, depending on the file, it can take a while until its transfer, so, it can have a lower priority and take its time to transfer.

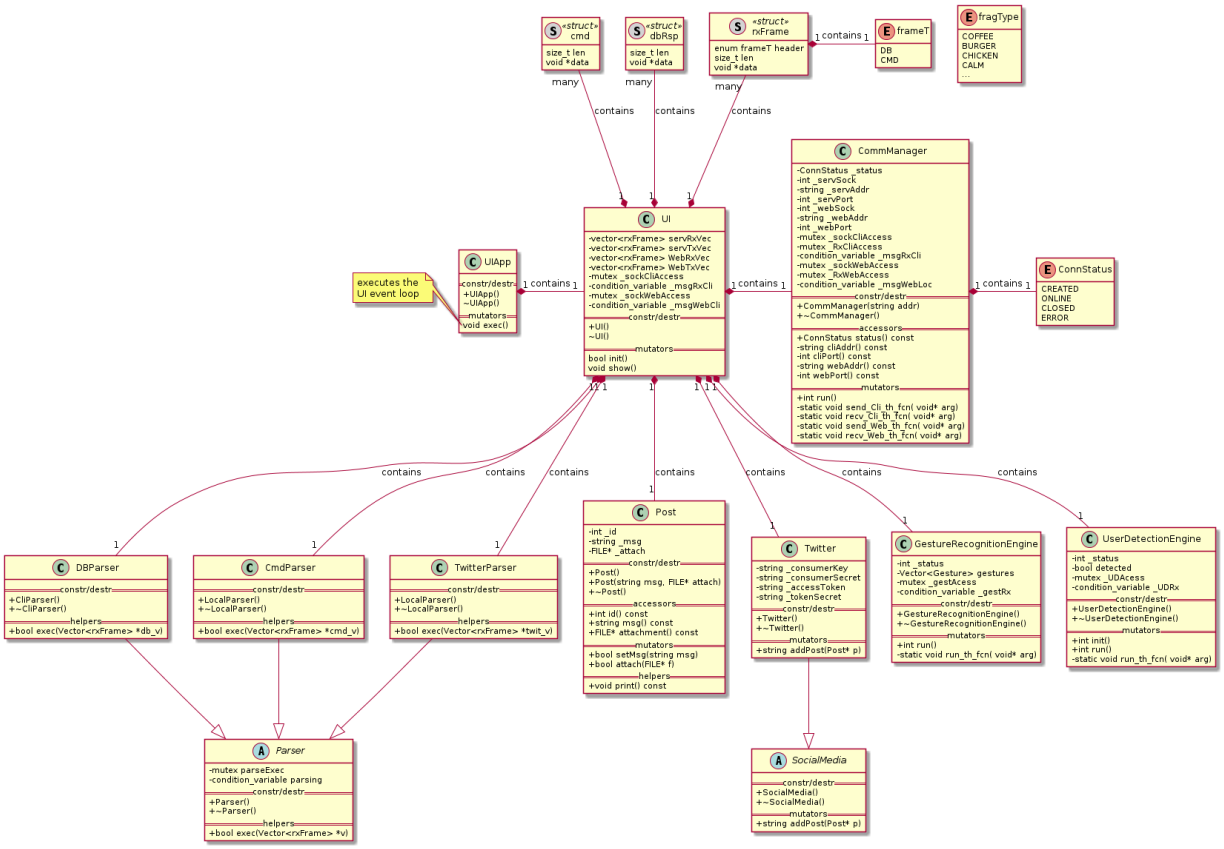

Figure 1.22.: Class diagram: **Local System** — part 1/2

2. Parser: the parser has one thread that is its execution. This thread has high priority because it is mandatory to parse and execute the commands that are received, as soon as they are received.
3. Comm: the communication manager has three threads that are important: the **receive**, the **send**, and the **accept connection**. The one with most priority is the **receive** thread, because it is necessary to receive the messages to then parse and execute them. Then, the other two threads have medium priority because receiving and parsing messages is more important than this.
4. DB: the database management has three threads that have high priority, that's because it is important to always have the databases updated. These three threads make queries to the databases, but they are controlled one by another in order to avoid the risk to make queries at the same time or some queries to be lost. There's another thread that handles the commands sent to the server or between the two subsystems

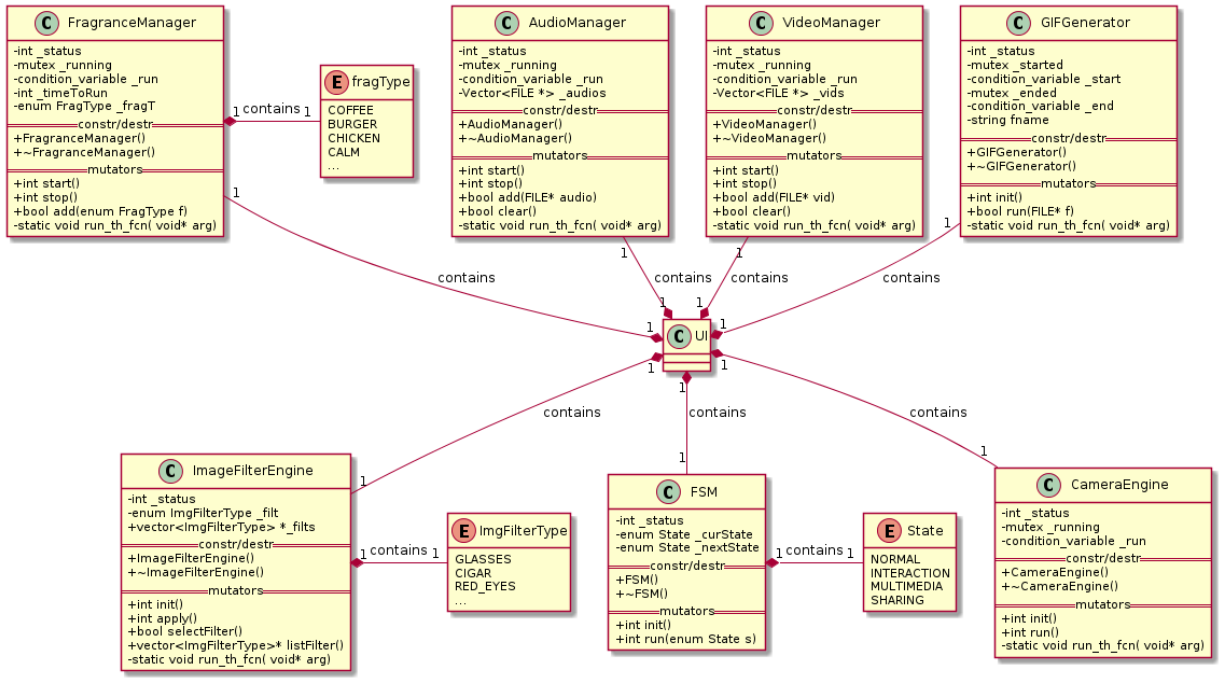

Figure 1.23.: Class diagram: **Local System** — part 2/2

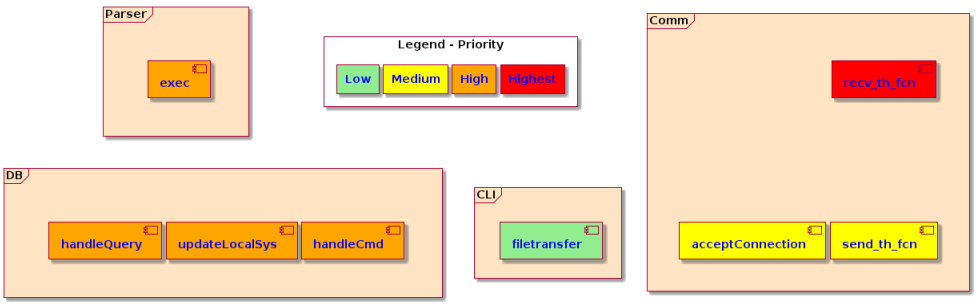

Figure 1.24.: Thread specification: **Remote Server**

Local System

Fig. 1.25 illustrates the thread specification and the associated priorities for the **Local System**. There are four logical groups:

1. User Interface: the **UserDetection** detects when a user is in range and is the top priority thread, as it will determine the device's mode, stopping normal mode and going into interaction mode.
   The **FrameGrabber** and **UI** threads are high priority. The first grabs frames from the camera and stores it in a frame buffer which is available to the **UI**, **ImgFilterOverlay**, and **GestureRecognition** threads. The second is responsible for providing user feedback on the Liquid Crystal Display (LCD) display.

The **ImgFilterOverlay**, and the **GestureRecognition**, and the threads are medium priority. The first is responsible for detecting faces on the camera frame buffer and applying the selected image filter overlay, and the second for recognizing **User** gestures.
It should be noted that only the **UserDetection** and **UI** threads should be permanently running, while the others are only running while an **User** is detected.

2. Comm: these threads are responsible for communications handling. **LocalRx** is the top priority thread, as it needs to check if an relevant command or database update has arrived. It stores the incoming data frames into a First-In, First-Out (FIFO) buffer.
   The **AppParser** and the **LocalTx** are high priority threads. The first consumes incoming data frames and dispatches it for processing. The second transmits outgoing data frames to the **RemoteServer**.
   The **TwitterShare** and **FileTransfer** are low priority tasks, as they don't impact the user experience or device operation significantly. The first is responsible for sharing posts on Twitter. The second handles file transferring between **RemoteServer** and the **LocalSystem**.
3. Normal mode: these threads are only running when normal mode is on. **AudioMan** and **VidMan** are high priority tasks, and they are responsible for Ad's audio and video management, respectively.
   FragMan is a medium priority task, as variations in the fragrance diffusion are less significant for user experience. It is responsible for diffusing the fragrance.
4. Multimedia mode: the **GifGenerator** is a low priority thread, responsible for generating GIFs. It should only run after a user request was acknowledged.

1.3.7. Flowcharts

In this section are presented the most important flowcharts of all the systems. These flowcharts are important to implement because it gives to one the essentials to build all the system.

Remote Client

In the **Remote Client** there are several functions to implement. However, most of them have particularly the same implementation method. This is because the class diagrams are designed in order to replicate most of the code and to have many things in common, turning everything more easy to implement and to execute.

In Fig. 1.26 is the flowchart of the **Login** feature. After the login button being pressed, it is necessary to verify if all fields are filled. If so, the password is encrypted in order to compare with the passwords from the database (that are all encrypted). It is made a **query** to the users database to select a count and the role of all users that have the same username and password. If the count is equal to one, then the user

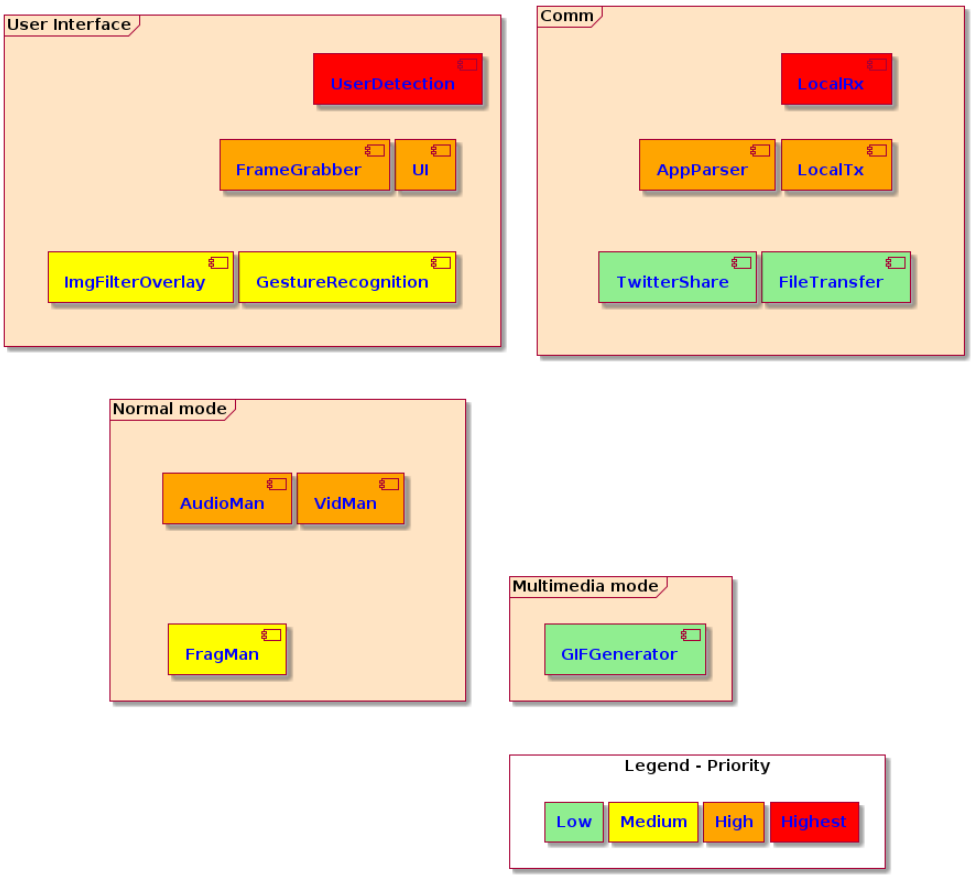

Figure 1.25.: Thread specification: **Local System**

has its credentials validated and it is shown the view according to its role. If not, then there's an error in the login.

In Fig. 1.27 is depicted the flowchart for registering a user. Basically, after the button being pressed and all the fields are filled, an user object is created. To know if the user can be created, it is made a **query** to the Users database to know if there is already an user with that username or that password. If not, then it is made another **query** to add that user to the users database and the registration is complete.

In Fig. 1.28 is depicted a flowchart of the delete user feature. Basically, the user is deleted using its id, and for that it is made a **query** to the users database to delete the row with that user id. After that query, the delete process is completed

In Fig. 1.29 is the flowchart of how it behaves the rent ad feature. Basically an ad object is created and added to the timetable object. After that, it is made a **query** to the time table database to add the ad to the specific time slots. Then, if everything occurs as expected, the Ad is then added to the database by a **query**. If that query don't occur as expected, then the timetable previously added needs to be removed

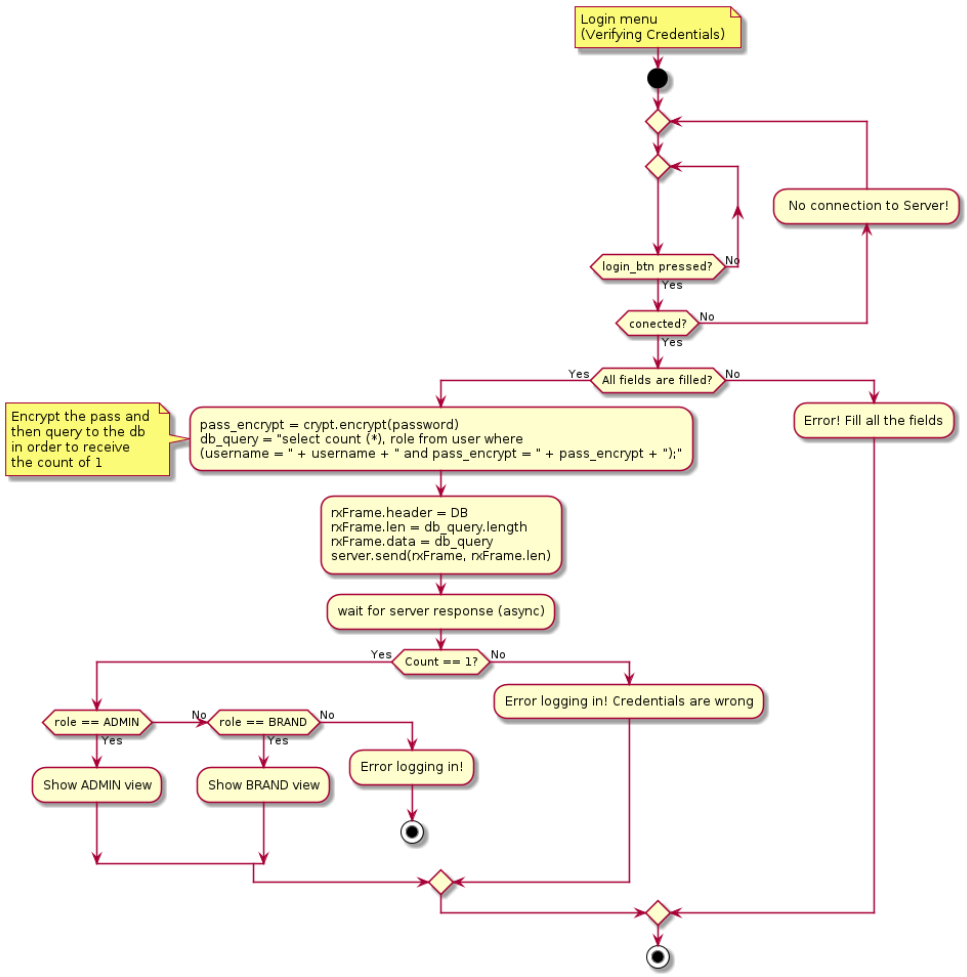

Figure 1.26.: Flowchart: **Remote Client** — Login

from the database.

Remote Server

The **Remote Server** is the connection node between the **Remote Client** and the **Local System**. Also, it has an important influence in the databases. So, there are several functions to implement. The most important functions to implement are the thread functions. Some of them can be analogous which can be good because most of the code can be replicated.

Fig. 1.30 shows how the receive thread is implemented. Basically this thread waits to receive data from a socket and after that, add that received command to the vector RxVec. In this situation it is mandatory to use mutexes to avoid accessing RxVec at the same time in different threads. Also it is used a semaphore to handle the number of threads that can use this vector. The send thread is particularly the same thing,

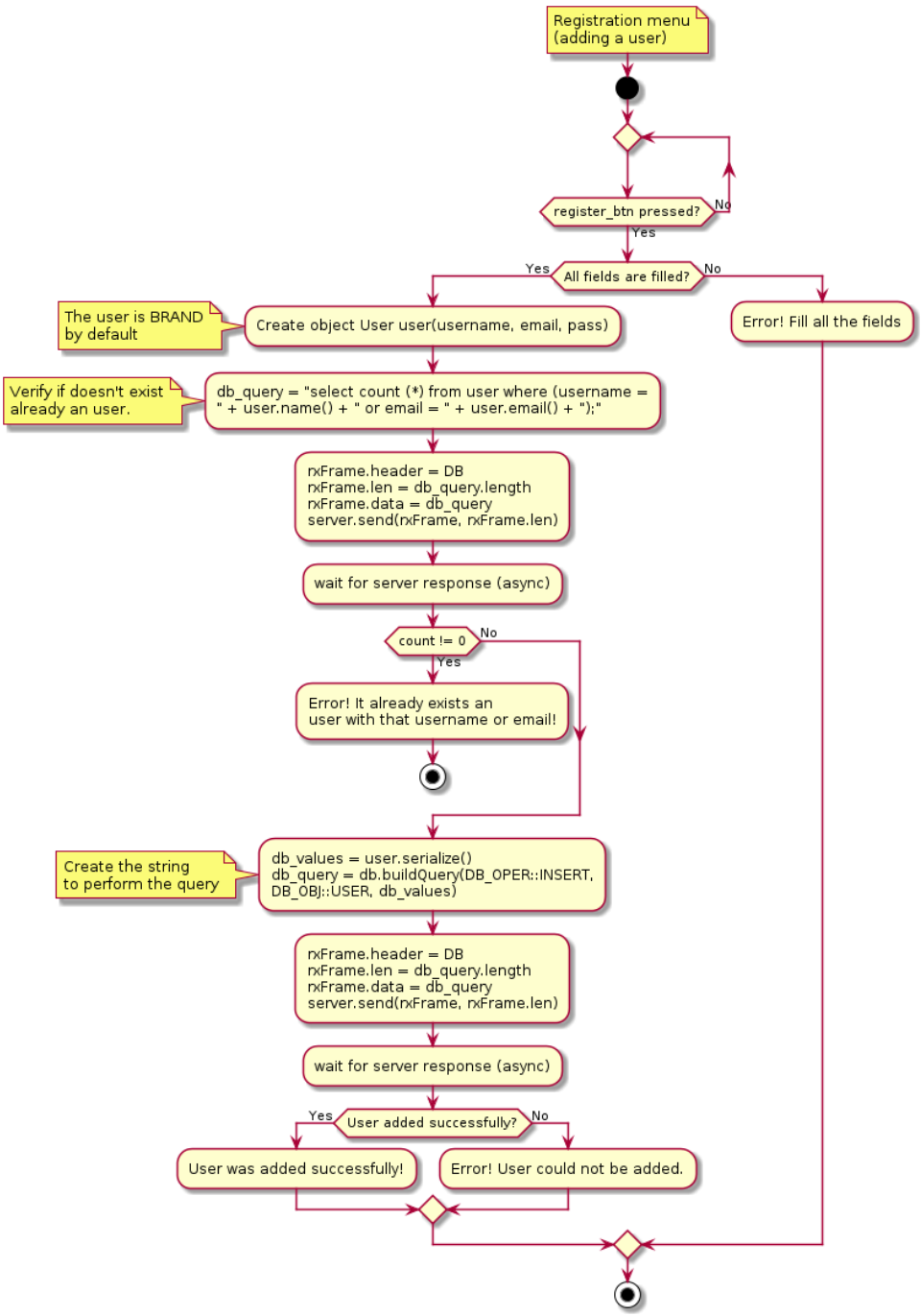

Figure 1.27.: Flowchart: **Remote Client** — Register

with the difference that it is necessary to know if there is something to send and send it. The information of the socket is already in the txVec and it is easy to send the information to the specific side.

Fig. 1.31 shows how the parser is executed. Basically this thread verifies if it exists messages received

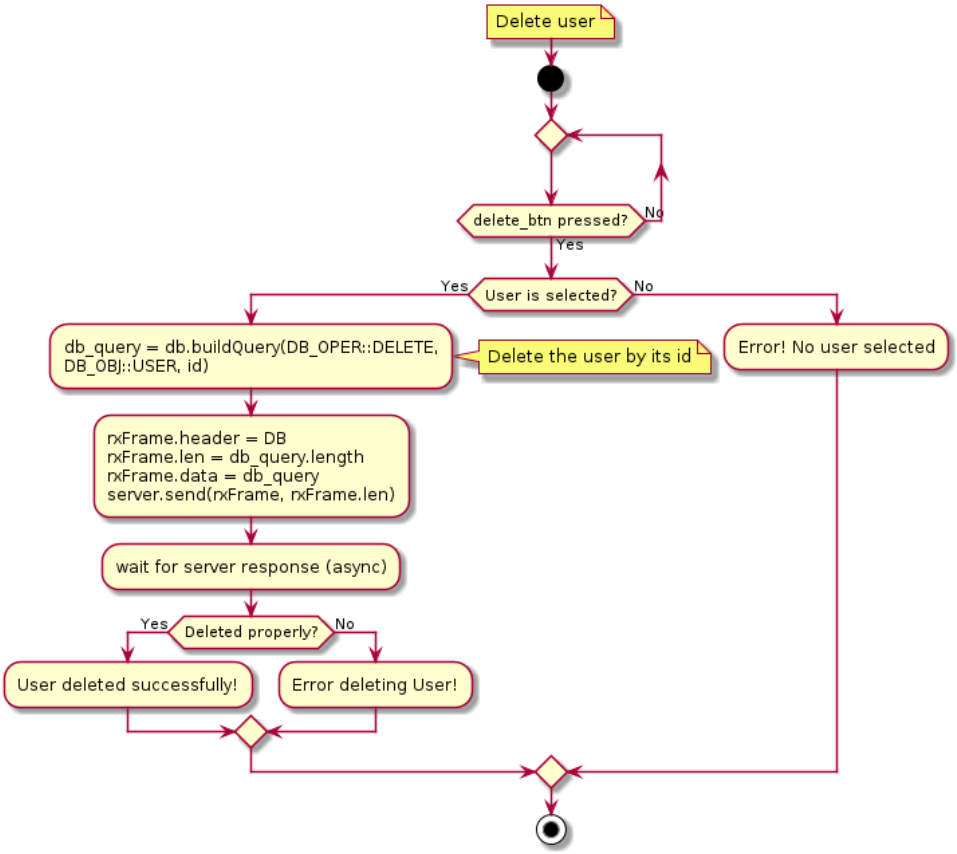

Figure 1.28.: Flowchart: **Remote Client** – Delete User

to read, if so, they are popped from the FIFO and it is dropped the semaphore that was being used. Then, by the structure rxFrame it is verified if the data is a command or a database query. If it is a command, it is signalized that it is a command and it is added to the vector, if it is a database query, it does the same but for the signal of the database handle and the db vector. It is always mandatory to avoid collisions of data, using for that the mutexes to lock the usage of the vectors.

Fig. 1.33 shows how the handleCmd thread is processed. Firstly, it is necessary to pop a command from the cmdVec and then execute it. It is necessary to know if the command is to execute in the remote server or to send to the local system. If it is to send to the local system, it is needed to make a query to the database to know the ip of the local system and then send to the machine.

Fig. 1.32 shows how the handleQuery thread is processed. Firstly, it is necessary to retrieve a query from the vector dbQueries (using the mutex to save the vector from other usage) and then make the query if no one is querying the database. After the query, it is received and stored the response to then send to the other side. So, the response is saved on the TxVec that already has the destination socket.

Fig. 1.34 illustrates how the update local system works. Basically, when some field from the database is

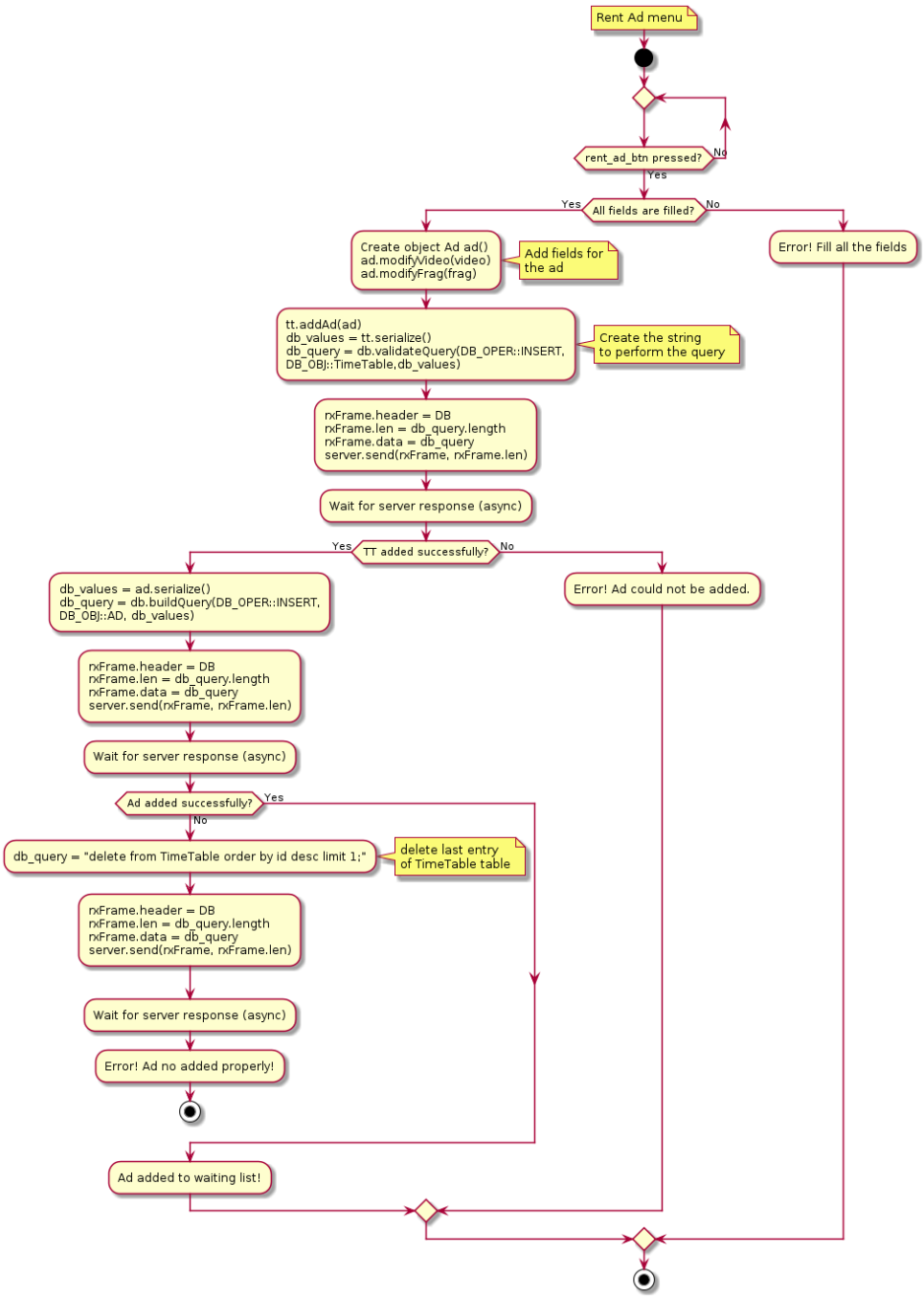

Figure 1.29.: Flowchart: **Remote Client** — Rent Ad

changed and it is necessary to inform the local system to change its behavior, the database sends a trigger that is received from this thread. After that, the thread needs to make the query to the explicit database that made the trigger to ask for all the information that changed to send to the station. After receiving the

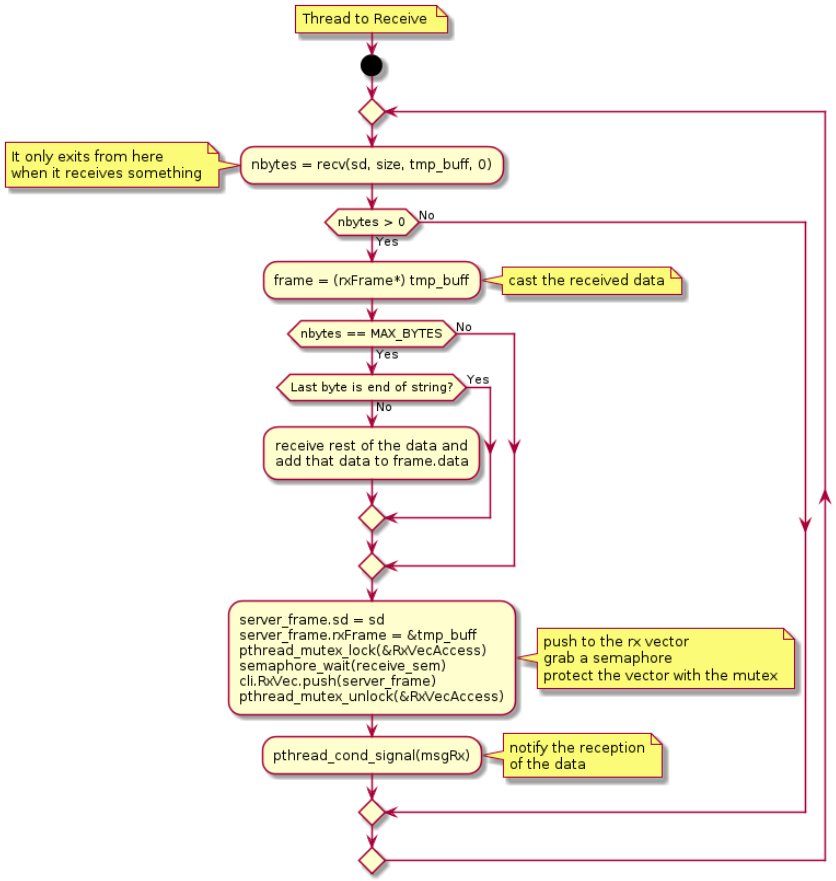

Figure 1.30.: Flowchart: **Remote Server** — Receive Thread

responses from the queries, everything is wrapped in the txVec (that also will contain to which station to send) and it is signaled the need to send information.

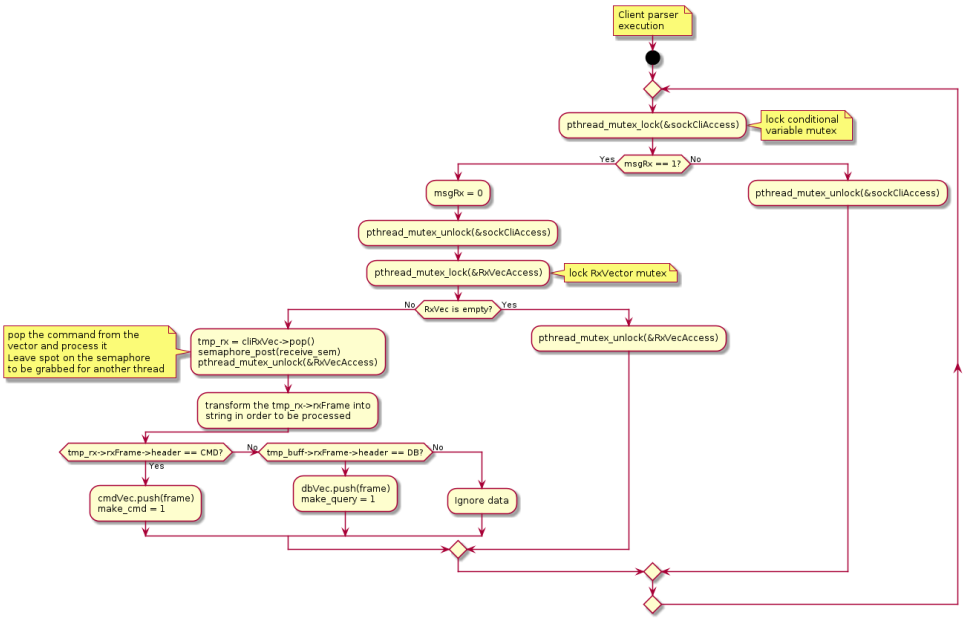

Figure 1.31.: Flowchart: Remote Server — Parser

1.4. Software interfaces definition

1.5. Start-up/shutdown process specification

1.6. Error handling specification

1.7. Test cases

1.7.1. Remote Client

1.7.2. Remote Server

1.7.3. Local System

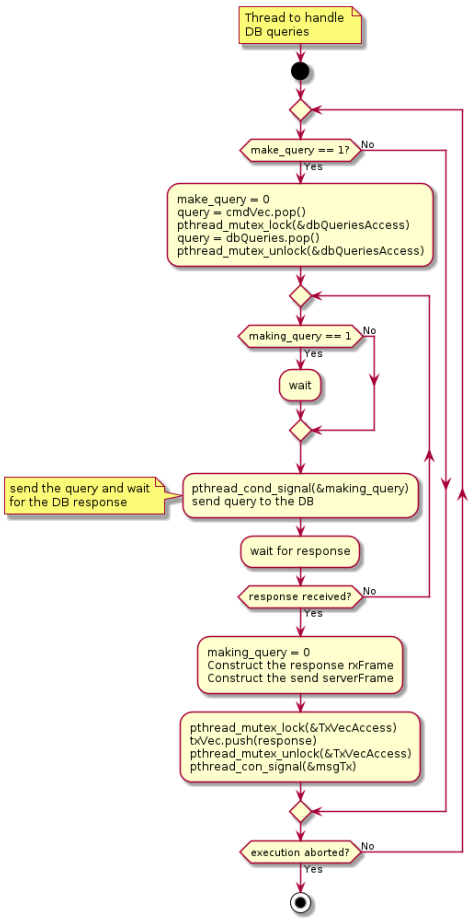

Figure 1.32.: Flowchart: **Remote Server** — handleQuery

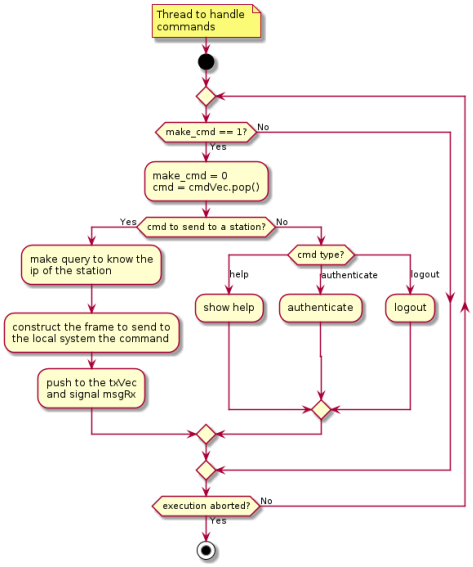

Figure 1.33.: Flowchart: **Remote Server** — handleCmd

Table 1.4.: Remote Client Test Cases

Use Case	**Type of test**	**Description**	**Expected Result**
Register	Functional	One will try to make a new register as a Brand	If the User is created and added to the database, everything is in order
Login	Functional	One will try to Login through its user and password.	If the login is successful, everything is working as expected.
Logout	Functional	One will try to log out from the account.	If it returns to the login page, it works.
Manage User	Functional	One will manage a user removing it or giving him privileges as Admin.	If the user is removed from the DB or has privileges as admin, it works well.
Manage Station	Functional	One will try to manage a station by powering on/off, manage ads or enable/ disable ad.	If the information is updated and the station reacts to the commands, the system is working well.
Test Operation	Functional	One will choose to test the operation of the machine.	If a command line is provided with the remote server, it works properly.
Display Rented Ads	Functional	One will try to display the rented ads by the user in case.	If there are displayed all the ads of that brand, then everything works properly.
Rent Ads	Functional	One will try to rent ads as a brand, inserting all data necessary.	If the remote client responds in order, saving all data in the DB, it works well.
See Notifications	Functional	One will try to watch the notifications of a brand.	If the brand has notifications, it should be displayed. This means that it works.

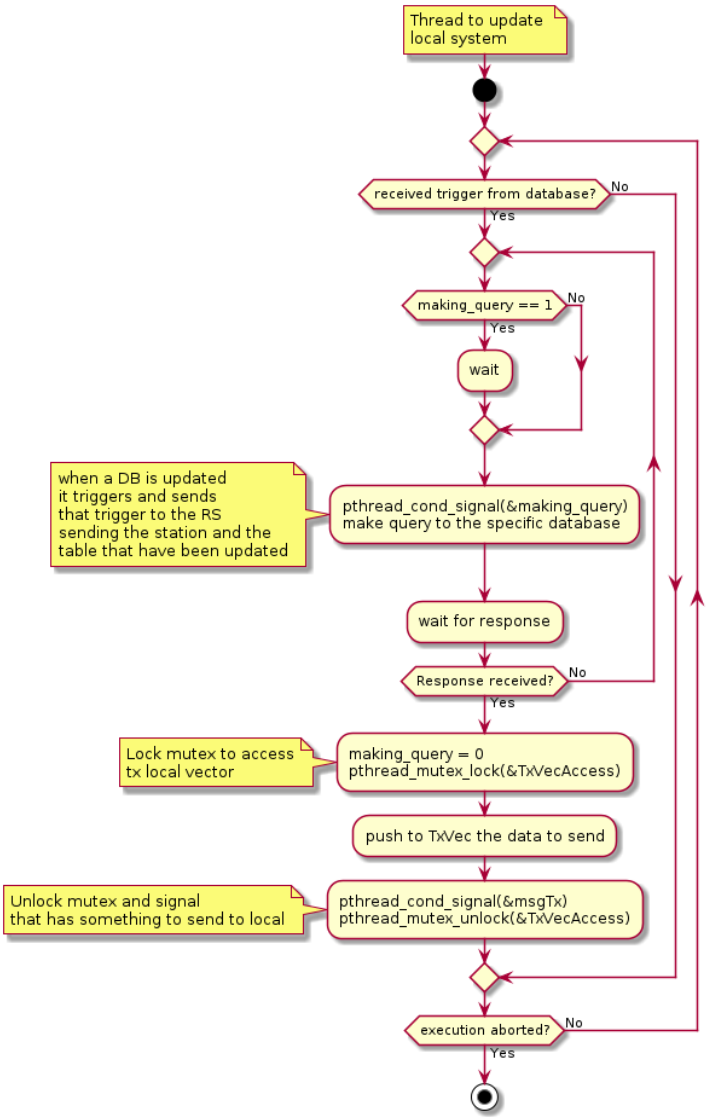

Figure 1.34.: Flowchart: **Remote Server** — updateLocalSys

Table 1.5.: Remote Server Test Cases

Use Case	Type of test	Description	Expected Result
Disconnect	Functional	One will try to disconnect.	It has to disconnect successfully.
Authenticate User	Functional	One will try to authenticate a user already present in the database.	The server needs to respond in order: accept authentication or decline if failed.
Help	Functional	One will try to get help.	Information to help the user should be provided.
Interact with databases	Functional	One will try to query a database, reading, modifying, adding or deleting something.	If the database is correctly updated according to the command, then it works.
		Test Operation	
Manage User	Functional	One will manage a user removing it or giving him privileges as Admin.	If the user is removed from the DB or has privileges as admin, it works well.
Manage Audio	Functional	One will try to manage the audio, playing an audio file that is present on the database.	The server must send the information of the audio to play to the local system. If it works, then it is verified.
Manage Video	Functional	One will try to manage the video, playing a video file that is present on the database.	The server must send the information of the video to play to the local system. If it works, then it is verified.
Manage Fragrance	Functional	One will try to activate a fragrance diffusion to the machine.	If the server sends the right information to the LS, then the LS should diffuse the fragrance.
Manage Camera	Functional	One will try to manage the camera with its various features (turn on/off, facial recognition, take GIF, take picture apply filter).	If the remote server sends the right info, then the local system should actuate in the camera according to the function that was requested to operate.
Share	Functional	One will try to share some file through social media.	If the LS responds correctly to that command, then it is working properly.

Table 1.6.: Local System Test Cases

Use Case	Type of test	Description	Expected Result
Establish Connection	Functional	One will try to establish connection to the machine	If the connection is established and the credentials are verified, everything works.
End remote Connection	Functional	One will try to end the remote connection.	If the connection is ended correctly, then it works as expected.
Select Image Filter	Functional	One will try to select an image filter and use it.	It has to apply correctly the facial detection and also apply correctly the filter.
Take Picture	Functional	One will try to take a picture.	The Local System has to take the picture and store it in order to behave properly.
Create GIF	Functional	One will try to create a GIF.	The Local System has to generate the GIF and store it in order to work properly.
Share on social media	Functional	One will try to share a picture or a GIF on social media.	If the Local System shares to the social media the picture or the GIF, it works.
Diffuse Fragrance	Functional	One will try to diffuse the fragrance.	If the fragrance is diffused, it works.
Play Video	Functional	One will try to play one video of an ad or something else.	If the video plays properly, then the machine is working as predicted.
Play Audio	Functional	One will try to play a random audio.	If the video audio plays properly, then the machine is working as predicted.
Process Commands	Functional	One will try to send some commands for the machine to process.	If the machine receives the commands properly and processes them, then it works well.
Detect Face	Functional	One will trigger the machine to detect the face of a user.	The face will be detected if the machine is working properly.
Recognize Gesture	Functional	One will make gestures to the camera in order to see the machine behavior.	The machine will recognize the gestures and act according to the gesture made.
Update Internal DBs	Functional	One will try to play an ad or diffuse a fragrance one more time.	The machine should update the number of times the ad was played or the percentage of fragrance that is still in the slot (in the DBs).

Bibliography

[1] Raspberry pi 4 tech specs, . URL https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. accessed: 2021-12-11.

[2] How hc-sr04 ultrasonic sensor works & interface it with arduino. URL https://lastminuteengineers.com/arduino-sr04-ultrasonic-sensor-tutorial/. accessed: 2021-12-11.

[3] Micro usb ultrasonic atomizing humidifier module fogger mist generator uk. URL https://www.ebay.co.uk/itm/203635455945?hash=item2f699e77c9:g:j-cAAOSwE75hVti0. accessed: 2021-12-11.

[4] Raspberry pi camera module 2. URL https://www.raspberrypi.com/products/camera-module-v2/. accessed: 2021-12-11.

[5] 10.1 inch hdmi screen lcd display with audio driver board monitor for raspberry pi banana/orange pi mini computer. URL https://www.aliexpress.com/item/33005274109.html?spm=a2g0o.detail.1000013.3.38072ea5MrfWq9&gps-id=pcDetailBottomMoreThisSeller&scm=1007.13339.169870.0&scm_id=1007.13339.169870.0&scm-url=1007.13339.169870.0&pvid=e1784609-f2f3-4547-9d9e-d5461fddc4c8&_t=gps-id:pcDetailBottomMoreThisSeller,scm-url:1007.13339.169870.0,pvid:e1784609-f2f3-4547-9d9e-d5461fddc4c8,tpp_buckets:668%232846%238110%231995&&pdp_ext_f=%7B"sceneId":"3339","sku_id":"12000023440173741"%7D. accessed: 2021-12-11.

[6] Passive vs. active speakers, which is right for you? URL https://www.aperionaudio.com/blogs/aperion-audio-blog/passive-vs-active-speakers. accessed: 2021-12-11.

[7] 5w speaker - para display (h). URL https://www.botnroll.com/pt/colunas-sirenes/2726-5w-speaker-para-display-h.html. accessed: 2021-12-11.

[8] Carregador 4x usb 5v / 2,4a (branco) - fonestar. URL https://www.castroelectronica.pt/product/carregador-4x-usb-5v-24a-branco--fonestar. accessed: 2021-12-11.

[9] Raspberry pi 4 pinout, . URL https://www.the-diy-life.com/raspberry-pi-4-pinout/. accessed: 2021-12-13.

Appendices

A. Project Planning — Gantt diagram

In Fig. A.1 is illustrated the Gantt chart for the project, containing the tasks' descriptions.

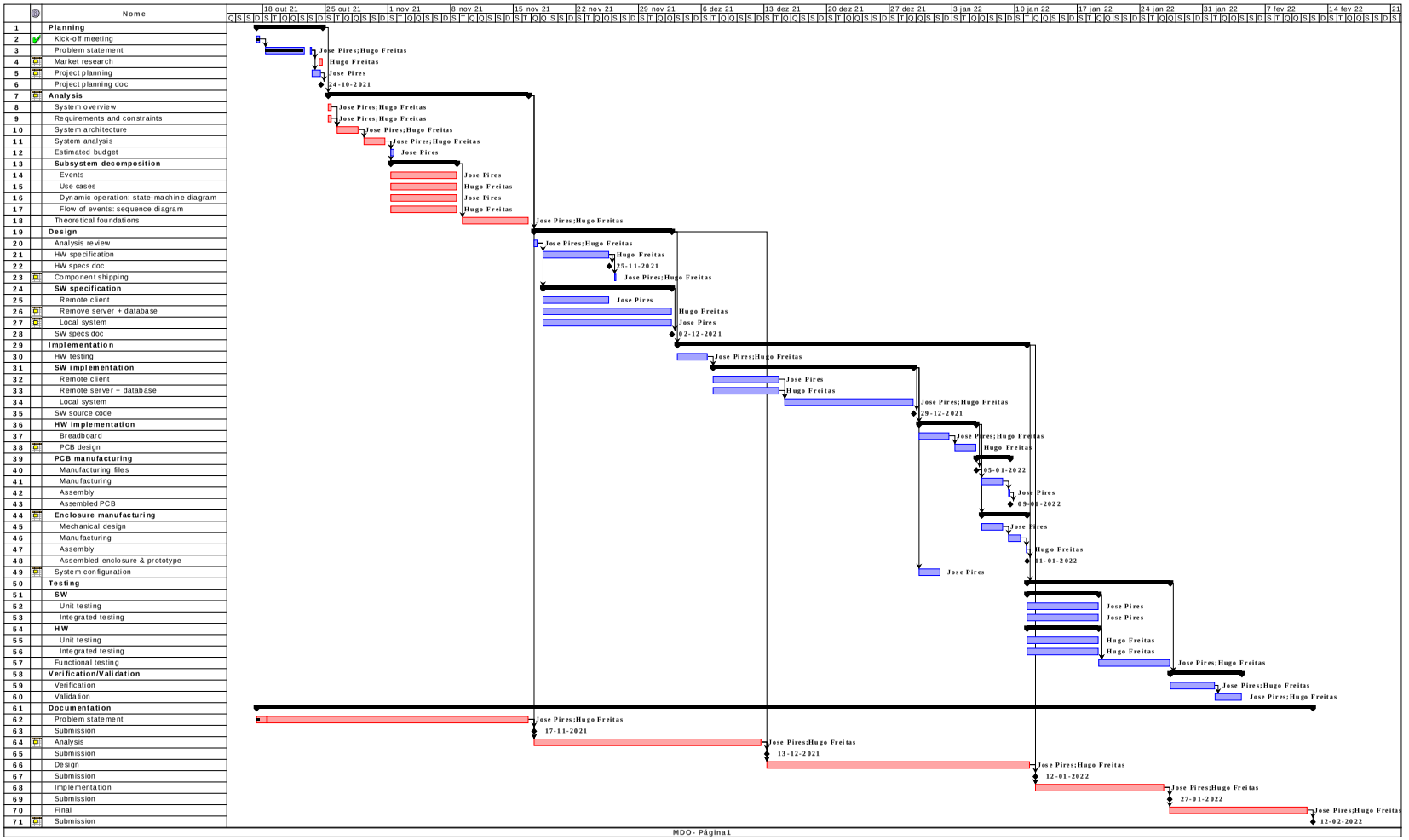

Figure A.1.: Project planning — Gantt diagram