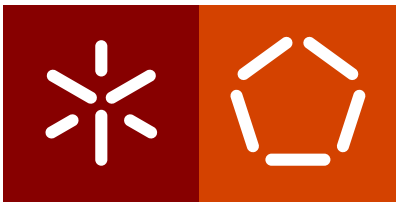


University of Minho
School of Engineering
Electronics Engineering department
Embedded systems

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 6, 2021

Contents

List of Figures

List of Abbreviations

Notation	Description	First used on page nr.
AC	Alternating Current	12
ACF	Aggregated channel feature	103
ANN	Artificial Neural Networks	107
API	Application Programming Interface	4
BN	Billions	2
BSD	Berkeley Software Distribution	73
BSP	Board Support Package	13
CAGR	Compound Annual Growth Rate	3
CLI	Command Line Interface	13
CNN	Convolutional Neural Network	100
COTS	Commercial off-the-shelf	5
CPS	Cyber—Physical Systems	1
CPU	Central Processing Unit	103
CV	Computer Vision	15
DB	Database	iv
DBMS	Database Management System	107
DC	Direct Current	12
DDL	Data Definition Language	118
DML	Data Manipulation Language	118
DOOH	Digital Out-Of-Home	2
DPM	Deformable part models	103
ER	Entity-Relationship	109, 110
ERD	Entities-Relationships diagram	vi

Notation	Description	First used on page nr.
GIF	Graphics Interchange Format	v
GPU	Graphics Processing Unit	104
GUI	Graphical User Interface	125
HD	High-Definition	102
HOG	Histogram of Oriented Gradients	103
HTML	Hypertext Markup Language	85
HW	Hardware	iv
I/O	Input/Output	65
IOT	Internet of Things	102
IP	Internet Protocol	15
IPC	Inter-Process Communication	59
IR	infrared	123
KNN	K-nearest neighbor	107
LAN	Local Area Network	62
MAC	Media Access Control	62
MDO	Marketing Digital Outdoor	iv, 2
MDO-L	MDO Local System	9
MDO-RC	MDO Remote Client	9
MDO-RS	MDO Remote Server	9
MW	Mircowave	122
NB	Naive Bayes	107
OCR	Optical Character Recognition	100
OMT	Object-Modeling Technique	58
OOSE	Object Oriented Software Engineering	58
OS	Operating System	13
OSI	Open Systems Interconnection	v
PCB	Printed Circuit Board	5

Notation	Description	First used on page nr.
PGID	Process Group ID	68
PID	Process ID	68
PIR	Passive Infrared	122
POSIX	Portable Operating System Interface	58
PWM	Pulse-Width Modulation	96
R&D	Research and Development	3
RDBMS	Relational Database Management System	14
ROI	Region Of Interest	107
SID	Session ID	68
SoC	System-on-a-Chip	12
SQL	Structured Query Language	109
SVM	Support Vector Machine	103
SW	Software	iv
TCP	Transmission Control Protocol	63
TCP/IP	Transmission Control Protocol/Internet Protocol	9
UDP	User Datagram Protocol	63
UI	User Interface	6
UML	Unified Modeling Language	57
USD	United States Dollar	102
WAL	Write-Ahead Log	111
WLAN	Wireless local Area Network	62

1. Introduction

The present work, within the scope of the Embedded Systems course, consists in the project of a Cyber–Physical Systems (CPS), i.e., a system that provides seamless integration between the cyber and physical worlds [1]. The Waterfall methodology is used for the project development, providing a systematic approach to problem solving and paving the way for project's success.

In this chapter are presented the project's context and motivation, the problem statement — clearly defining the problem, the market research — defining the product's market share and opportunities, the project goals, the project planning and the document outline.

1.1. Context and motivation

COVID pandemics presented a landmark on human interaction, greatly reducing the contact between people and surfaces. Thus, it is an imperative to provide people with contactless interfaces for everyday tasks. People redefined their purchasing behaviors, leading to a massive growth of the online shopping. However, some business sectors, like clothing or perfumes, cannot provide the same user experience when moving online. Therefore, one proposes to close that gap by providing a marketing digital outdoor for brands to advertise and gather customers with contactless interaction.

Scenting marketing is a great approach to draw people into stores. Olfactory sense is the fastest way to the brain, thus, providing an exceptional opportunity for marketing [2] — "75% of the emotions we generate on a daily basis are affected by smell. Next to sight, it is the most important sense we have" [3].

Combining that with additional stimuli, like sight and sound, can significantly boost the marketing outcome. Brands can buy advertisement space and time, selecting the videoclips to be displayed and the fragrance to be used at specific times, drawing the customers into their stores.

Marketing also leverages from better user experience, thus, user interaction is a must-have, providing the opportunity to interact with the customer. In this sense, when users approach the outdoor a gesture-based interface will be provided for a brand immersive experience, where the user can take pictures or create GIFs with brand specific image filters and share them through their social media, with the opportunity to gain

several benefits.

1.2. Problem statement

The first step of the project is to clearly define the problem, taking into consideration the problem's context and motivation and exploiting the market opportunities.

The project consists of a Marketing Digital Outdoor (MDO) with sound and video display, and fragrance emission selected by the brands, providing a gesture-based interface for user interaction to create pictures and GIFs, brand-specific, and share them on social media. It is comprised of several modes:

- normal mode (advertisement mode): the MDO will provide sound, video and fragrance outputs.
- interaction mode: When a user approaches the device, the MDO will go into interaction mode, turning on and displaying the camera feed and waiting for recognizable gestures to provide additional functionalities, such as brand-specific image filters.
- multimedia mode: in this mode the facial detection is applied, enabling the user to select and apply different brand-specific image filters and take pictures or create a GIF.
- sharing mode: after a user take a picture or create a GIF, it can share it across social media.

Brands can buy advertisement space and time, selecting the videoclips to be displayed and the fragrance to be used at specific times, drawing the customers into their stores. Customers can be captivated by the combination of sensorial stimuli, the gesture-based interaction, the immersive user experience provided by the brands — feeling they belong in a TV advertisement, and the opportunity to gain several benefits, e.g., discount coupons.

1.3. Market research

A Digital Outdoor is essentially a traditional outdoor advertising powered up by technology. The pros of a digital outdoor compared to a traditional one is mostly the way that it captivates the attention of consumers in a more dynamic way. It can also change its advertisement according to certain conditions, such as weather and/or time. Some researches tells that the British public sees over 1.1 Billions (BN) digital outdoor advertisements over a week [4], which can tell how much digital marketing is valued nowadays.

When talking about numbers, "At the end of 2020, despite the COVID wipe-out, the Digital Out-Of-Home (DOOH) market was estimated to be worth $41.06 BN, but by 2026, nearly two out of three (65%) advertising executives predict this will rise to between $50 BN and $55 BN. A further 16% expect it to be worth between $55 BN and $60 BN, and 14% estimate it will be even bigger" [5].

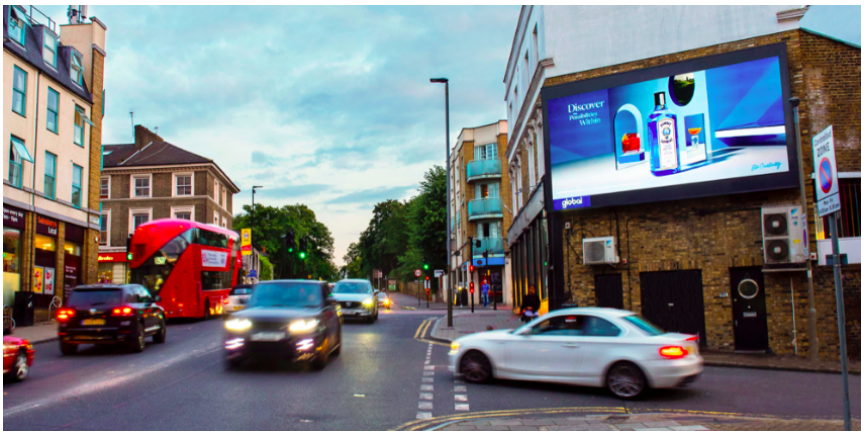


Figure 1.1.: Example of a Digital Outdoor, withdrawn from [4]

Scent market is the art of taking a company's brand identity, marketing messages, target audience and creating a scent that amplifies these values. That's because "a scent has the ability to influence behavior and trigger memories almost instantaneously. When smell is combined with other marketing cues, it can amplify a brand experience and establish a long lasting connection with consumers" [6].

Ambient scent uses fragrance to enhance the experience of consumers with different purposes, whereas scents in scent branding are unique to each company's identity. According to a Samsung study: "when consumers were exposed to a company scent, shopping time was increased by 26% and they visited three times more product categories" [7]. Also, "the digital scent technology market is expected to grow from $1.0 BN in 2021 to $1.5 BN by 2026, at a Compound Annual Growth Rate (CAGR) of 9.2%." [8].

The market growth can be attributed to several factors, such as expanding application and advancements in e-nose technologies, increasing use of e-nose devices for disease diagnostic applications, emerging Research and Development (R&D) activities to invent e-nose to sniff out COVID-19, and rising use of e-nose in food industry for quality assurance in production, storage, and display.

1.4. Project goals

The project aims to develop a CPS for multi-sensory marketing with contactless user interaction. The key goals identified and the respective path to attain them are:

1. devise a device with audio and video outputs, as well as fragrance diffusion: understand audio and video streaming and study fragrance nebulizer technologies.
2. create a contactless user interface based on gestures through computer vision: identify user gestures through computer vision and match them to interface callbacks; a virtual keyboard may be required

Figure 1.2.: Attractive Opportunities in the Digital Scent Technology Market, withdrawn from [8]

for user input.

3. devise a distributed architecture to convey brand advertisement information to the local device: understand distributed architectures and apply them for optimal data flow; create a remote client-server model to convey information from the brands to the device through remote cloud database services; devise adequate data frames to convey information to the local device; create a local server to respond to the remote server requests.
4. apply facial detection to the camera feed and subsequently apply image filters specific to each brand: understand facial detection algorithms and apply them to the camera feed; apply image filters on top of the identified faces through a specialized Application Programming Interface (API).
5. enable image and GIFs sharing to social media for increased brand awareness: understand how to use social media APIs for media sharing.

1.5. Project planning

In Appendix A is illustrated the Gantt chart for the project (Fig. A.1), containing the tasks' descriptions. It should be noted that the project follows the Waterfall project methodology, which is meant to be iterative.

The tasks are described as follows:

- Project planning: in the project planning, a brainstorming about conceivable devices takes place, whose viability is then assessed, resulting in the problem statement (Milestone 0). A market research

is performed to assess the product's market space and opportunities. Finally, an initial version of the project planning is conceived to define a feasible timeline for the suggested tasks.

- Analysis: in this phase an overview of the system is conceived, presenting a global picture of the problem and a viable solution. The requirements and constraints are the elicited, defining the required features and environmental restrictions on the solution. The system architecture is then derived and subsequently decomposed into subsystems to ease the development, consisting of the events, use cases, dynamic operation of the system and the flow of events throughout the system. Finally, the theoretical foundations for the project development are presented.
- Design: at this stage the analysis specification is reviewed, and the HW and SW and the respective interfaces are fully specified. The HW specification yields the respective document, enabling the component selection, preferably Commercial off-the-shelf (COTS), and shipping. The SW specification is separately performed in the subsystems identified, yielding the SW specifications documentation (milestone).
- Implementation: product implementation which is done by modular integration. The HW is tested and the SW is implemented in the target platforms, yielding the SW source code as a deliverable (milestone). The designed HW circuits are then tested in breadboards for verification and the corresponding Printed Circuit Board (PCB) is designed, manufactured and assembled. After designing the PCB, the enclosure is designed to accomodate all HW components, manufactured and assembled. Lastly, the system configuration is performed, yielding prototype alpha of the product.
- Tests: modular tests and integrated tests are performed regarding the HW and SW components and a functional testing is conducted.
- Functional Verification/Validation: System verification is conducted to validate overall function. Regarding validation, it is conducted by an external agent, where a user should try to interact with the designed prototype.
- Documentation: throughout the project the several phases will be documented, comprising several milestones, namely: problem statement; analysis; design; implementation; and final.

1.6. Report Outline

This report is organized as follows:

- In Chapter 1 is presented the project's context and motivation, the problem statement, the market research, the project goals, and project planning.
- In Chapter 2, the product requirements are derived — defining the client expectations for the product

— as well as the project constraints — what the environments limits about the product. Based on the set of requirements and constraints, a system overview is produced, capturing the main features and interactions with the system, as well as its key components. Then, the system architecture is devised, comprising both hardware and software domains. Next, the system is decomposed into subsystems, presenting a deeper analysis over it, comprising its user mock-ups, events, use cases diagram, dynamic operation and flow of events. A budget estimation is conducted to evaluate the project's costs for both the scale-model and real-scale prototypes.

- Chapter 3 lays out the theoretical foundations for project development, namely the project development methodologies and associated tools, concurrency, the communications technologies and client–server architecture, daemons and device drivers, fragrance diffusion technologies, computer vision for facial detection and gesture recognition, database management, motion detection, camera recording, image filtering APIs, GIF generation, social media APIs, the User Interface (UI) framework, and the file transfer protocols.

- Lastly, the appendices (see Section 3.18) contain detailed information about project planning and development.

2. Analysis

In the analysis phase, the product requirements are derived — defining the client expectations for the product — as well as the project constraints — what the environment limits about the product. Based on the set of requirements and constraints, a system overview is produced, capturing the main features and interactions with the system, as well as its key components.

Then, the system architecture is devised, comprising both hardware and software domains. Next, the system is decomposed into subsystems, presenting a deeper analysis over it, comprising its user mock-ups, events, use cases diagram, dynamic operation and flow of events.

2.1. Requirements and Constraints

The development requirements are divided into functional and non-functional if they pertain to main functionality or secondary one, respectively. Additionally, the constraints of the project are classified as technical or non-technical.

2.1.1. Functional requirements

- Advertising through a screen and speakers;
- Have fragrance diffusion;
- Take pictures and GIFs;
- Detect a user in range of the device;
- Contactless user interaction through gesture recognition;
- Camera feed and facial detection;
- Apply brand-specific image filters;
- Enable sharing multimedia across social media;
- Provide a remote user interface for brands to purchase and configure the advertisements;
- Provide a remote user interface for company staff to monitor and control the MDO local system.

2.1.2. Non-functional requirements

- Low power consumption;
- Provide user-friendly interfaces;
- Have low latency between local system and remote server;
- Use wireless communication between the local and remote systems.

2.1.3. Technical constraints

- Use device drivers;
- Use Makefiles;
- Use C/C++;
- Use Raspberry Pi as the development board;
- Use compatible HW with the development board;
- Use buildroot;
- Social media APIs for sharing multimedia
- Image filtering through specialized APIs.

2.1.4. Non-technical constraints

- Project duration: one semester (circa 20 weeks);
- Pair work flow;
- Limited budget;
- Scale model prototype.

2.2. System overview

The system overview presents a global view of the system, considering its main features, components and interactions. It is not intended to be complete, but rather provide a basis for the outline of the system architecture. Fig. 2.1 presents the MDO system overview.

Considering the system interactions, three main actors were identified:

1. Brand: represents the brands contracting the advertisement services;
2. Administrator: the development company staff, which can monitor and control the outdoor (administrative privileges).

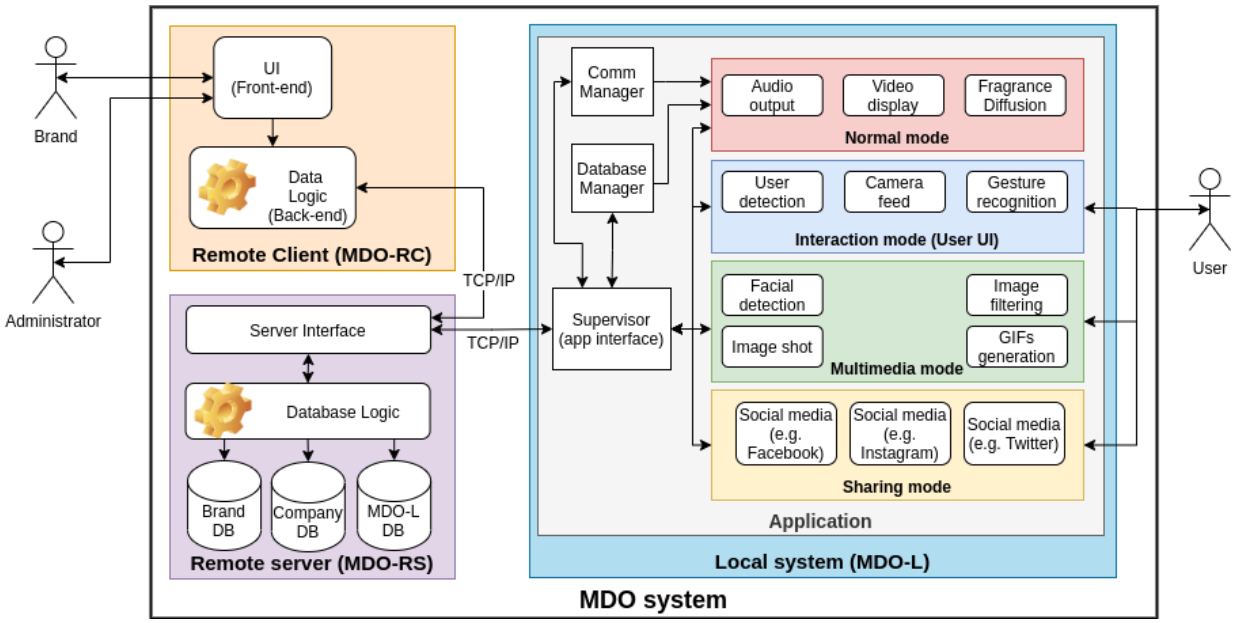


Figure 2.1.: MDO system overview

3. User: the user (the target audience of the advertisement) interacting with the system.

Considering the data flow across the **MDO system**, three main subsystems were identified: **MDO Remote Client (MDO-RC)**, **MDO Remote Server (MDO-RS)**, and **MDO Local System (MDO-L)**. The rational behind this initial decomposition is explained next.

2.2.1. MDO Remote Client

The Brand and Administrator members require a remote UI (front-end) to interact with the system: the former to configure the advertisements being displayed at the MDO and purchase them; the latter to remotely monitor and control the operation of the MDO. Thus, it is clear that an authentication mechanism must be provided for the remote UI.

The data is then dispatched to the back-end, where it is processed and feed back to the UI user and/or sent to the remote server, via Transmission Control Protocol/Internet Protocol (TCP/IP) comprising the data logic component of the UI.

2.2.2. MDO Remote Server

Although the MDO-RC could communicate directly with the MDO-L, this is not desirable or a good architecture mainly due to: communications failure could result in data loss, compromising the system's integrity; the

remote client and the local system become tightly coupled, meaning the remote client must be aware of all the available local systems; if the data storage in the local system fails, the remote client would have to provide the backup information.

Thus, a remote server component is included, providing the access and management of the system databases, pertaining to the Brand, Company, and MDO Local system. The first two provide the historical information of the **Brand** and **Administrator** entities, and the last one the information related to all of the **MDO-L** systems in operation.

The main functions of the **MDO-RS** are:

- UI requests responses: when a UI user requests/modifies some information from the database, the server must provide/update it.
- MDO-L monitoring and control: provide command dispatch and feedback to the **Administrator** staff for remote monitoring and control of the device.
- MDO-L update: periodically check for start times of each MDO-L device and transfer the relevant data to it.

The server interface is the responsible for managing the requests and respective responses from the remote client and for periodically send the update data to all MDO-L devices.

2.2.3. MDO Local system

The MDO local system (MDO-L) is the marketing device, interacting with the user to display multi-sensory advertisements. As aforementioned in Section 1.2, it is comprised of four modes:

- normal mode: the MDO provides sound, video and fragrance outputs. It is the default mode.
- interaction mode: When a user approaches the device, the MDO will go into interaction mode, turning on and displaying the camera feed and waiting for recognizable gestures to provide additional functionalities, such as brand-specific image filters. This is the **User** UI.
- multimedia mode: in this mode the facial detection is applied, enabling the user to select and apply different brand-specific image filters and take pictures or create a GIF.
- sharing mode: after a user take a picture or create a GIF, it can share it across social media.

The user interaction is considered to be a higher priority activity than the advertisements, so when a **User** interacts with the system, the **normal mode** is overriden by the **Interaction mode**, thus, halting the advertisements.

The MDO-L application communicates with the remote server (**MDO-RS**) through the **Supervisor** via TCP/IP to handle requests from **Administrator** members to monitor and control the device through the **Supervisor** or to update the advertisements. Additionally, the **Supervisor** oversees the application mode

and the communication (**Comm Manager**) and database (**Database manager**) managers to handle system events.

2.3. System architecture

In this section, the system architecture is devised in the HW and SW components, using the system overview as a starting point.

2.3.1. Hardware architecture

Fig. 2.2 illustrates an initial hardware big picture that fulfils the system's goals, meeting its requirements and constraints. As it can bee seen, the diagram is divided in four distinct parts: **External Environment**, **Local System**, **Remote Server** and **Remote Client**.

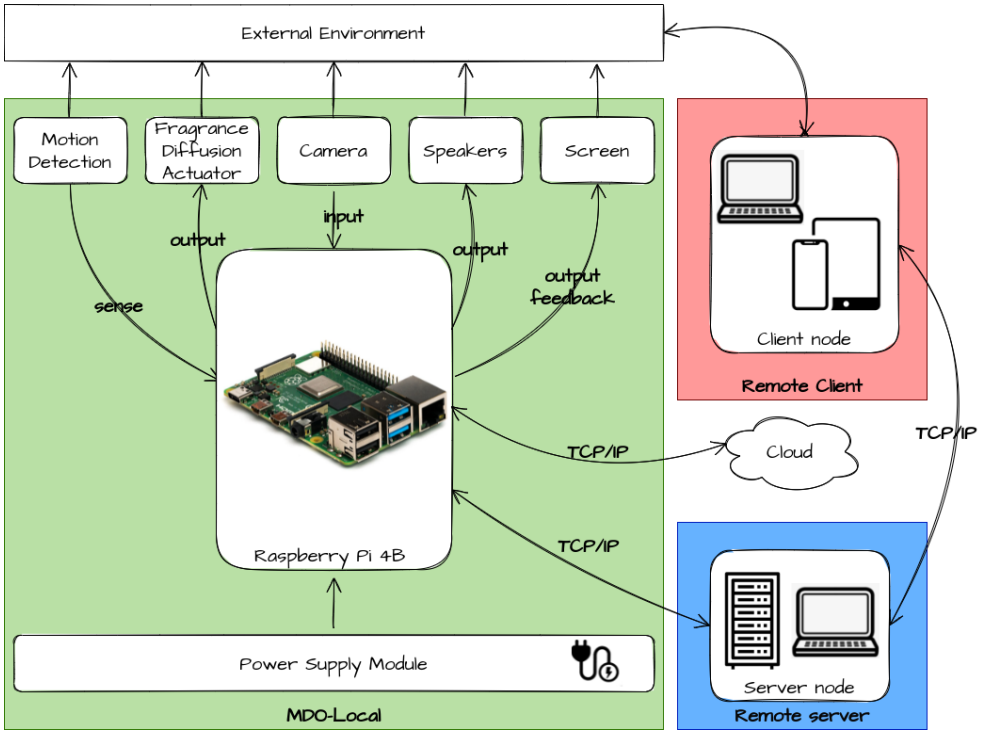


Figure 2.2.: HW architecture diagram

Firstly, the **External Environment** represents all the environment that interacts with the system. In this case, these are all its users — normal users, brands and company staff (Administrator role).

Secondly, the **Local System** is composed of the main controller, which is the Raspberry Pi 4B. This System-on-a-Chip (SoC) is responsible to control all the **Local System** and to establish a connection with the **Remote Server** through its included WiFi module. Additionally, the **Local system** also communicates with the Cloud to share contents on social media, and, potentially, to access image filtering APIs. The **Local System** is powered through a Alternating Current (AC)-Direct Current (DC) power converter, and, potentially, a step-down converter — **Power Supply Module**. The main board has several blocks connected to it:

- Motion Detection: used to detect the users and switch from normal mode to interaction mode;
- Fragrance Diffusion Actuator: used to diffuse the fragrance into the air;
- Camera: used to capture image that is then processed;
- Speakers: used to reproduce advertisements sounds;
- Screen: used to display video clips of advertisements.

In third place, the **Remote Server** has a server node running in another machine that can be one computer or a main frame. The remote server stores all databases which the **Remote Client** and **Local System** may need to access and serves as a proxy server to enable the **Admin** users to control and monitor the **Local System**.

Lastly, the **Remote Client** runs the MDO management application, which can be deploy to a computer (like the Raspberry Pi), a tablet or a smartphone.

2.3.2. Software architecture

In this section the SW architecture for MDO-RC, MDO-RS, and MDO-L subsystems is presented, defining its SW stack.

MDO remote client

Fig. 2.3 illustrates the SW architecture for the remote client, representing its SW stack. It is comprised of the following layers:

- Application: contains the remote client application. The **Brand** and **Admin** members interact with the UI, which is the visual part of the interface. The **UI engine** is notified and handles all UI events — internal or external — providing the **UI** with feedback for its users. The relevant commands are then parsed — **Parser** component — and responded. The commands are then translated to the appropriate DB queries and responded through the **DB Manager**. The **Comm Manager** is responsible for encapsulating the DB queries into the respective TCP/IP frames to be sent to the **Remote Server** as well as unwrap the incoming server responses.

- Middleware: contains the TCP/IP framework supporting these communication protocols as part of OSI model for internet applications. It manages the incoming/outgoing TCP/IP frames by providing the adequate protocol handshaking and queueing and timing aspects of the bytes to send/receive.
- OS & BSP — Operating System (OS) & Board Support Package (BSP): it contains the low-level and communication drivers required to handle input (keyboard/touch), output (screen) and communication to the **Remote Server**.

It should be noted that for desktop and mobile applications, the **Middleware** and **OS & BSP** layers are usually abstracted by the OS, thus, the relevant APIs should be used.

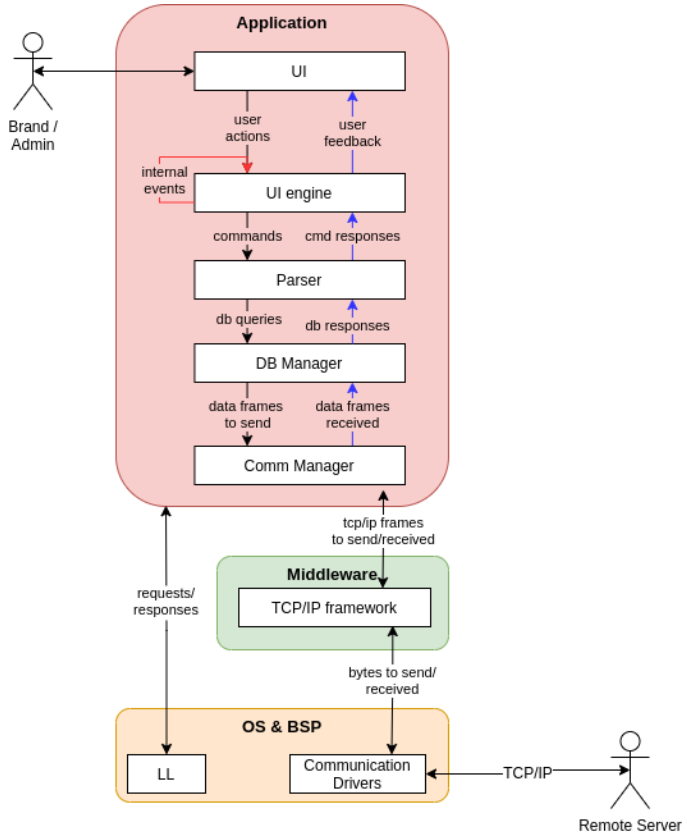


Figure 2.3.: SW architecture diagram: remote client

MDO remote server

Fig. 2.4 illustrates the SW stack for architecture for the remote server. It is comprised of the following layers:

- Application: contains the remote server application. It provides a Command Line Interface (CLI) to handle **Remote client** requests. The CLI engine is notified and handles all UI events — internal or external — providing the appropriate feedback. The relevant commands are then parsed — **Parser**

component — and responded: DB queries are handled by the **Relational Database Management System (RDBMS)** issuing DB transactions; other commands received from the **Remote Client** are handled internally and translated, being dispatched to the **Local System** by the **Comm Manager** (via **Communication drivers**). Internal events can also trigger the **RDBMS** to issue database transactions for the **Remote Client** or **Local System**. The **Comm Manager** is responsible for wrapping/unwrapping the data frames received by or sent to the **Remote Client** or **Local System**.

- Middleware: contains the RDBMS framework supporting the management of the relational databases using database transactions.
- OS & BSP — OS & BSP: it contains the **Communication** drivers to the handle requests from the **Remote Client**, and the **File I/O** drivers to manipulate DB transactions from/to storage.

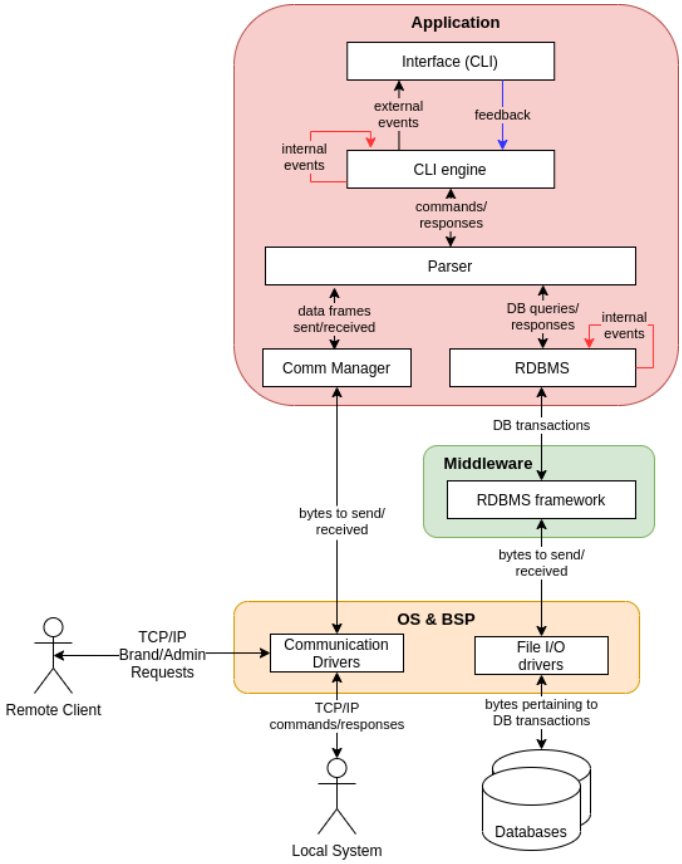


Figure 2.4.: SW architecture diagram: remote server

It should be noted that the **Remote Server** main functions are:

- provide relational databases for easier management of all entities and respective data in the system;
- decompose the relationship many-to-many, between the remote clients and local systems — many remote clients may want to connect to different local systems;

- decouple the architecture as the **Remote Client** should not know the Internet Protocol (IP) address of every local system it may potentially try to access, acting as a proxy server.

MDO local system

Fig. 2.5 illustrates the SW stack for architecture for the **Local System**. It is comprised of the following layers:

- Application: contains the local system application. It provides a UI to handle **User** interaction. The **Interface** engine is notified and handles all UI events — internal or external — through gesture recognition, providing the appropriate feedback. The relevant commands are then parsed — **Supervisor** component — and responded: DB queries are handled by the **Database manager** issuing DB transactions for internal databases; commands received from the **Remote Server** to monitor or control the system are handled internally and responded back by the **Comm Manager** (via **Communication drivers**); mode management is performed. Internal events can also trigger the **Database manager** to issue database transactions to update the **Local System**. The **Comm Manager** is responsible for wrapping/unwrapping the data frames received by or sent to the **Remote Server**.
- Middleware: contains: the DB framework supporting the management of the internal databases using database transactions; the Computer Vision (CV) framework that handles gesture and facial detection; image filtering and GIF frameworks for multimedia; social media framework.
- OS & BSP — OS & BSP: contains: the **Communication** drivers to the handle requests from the **Remote Server** and for social media sharing, and, potentially the API calls to cloud-based image filtering frameworks, depending on the application profiling; the **File I/O** drivers to manipulate internal DB transactions from/to storage; audio, video and fragrance diffuser actuator drivers for normal mode; the camera driver for camera feed; the detection sensor driver to signal a **User** is in range, triggering the switch from normal mode to interaction mode.

The **Local system** is a **soft real-time** system, as no mandatory deadlines must be met.

2.4. Subsystem decomposition

In this section the system is decomposed into subsystems and, for each subsystem, a more detailed analysis is performed yielding its user mock-ups, events, use case diagram, dynamic operation and the flow of events throughout the subsystem.

As aforementioned, the subsystems identified are: **Remote Client** (MDO-RC), **Remote Server** (MDO-RS), and **Local System** (MDO-L).

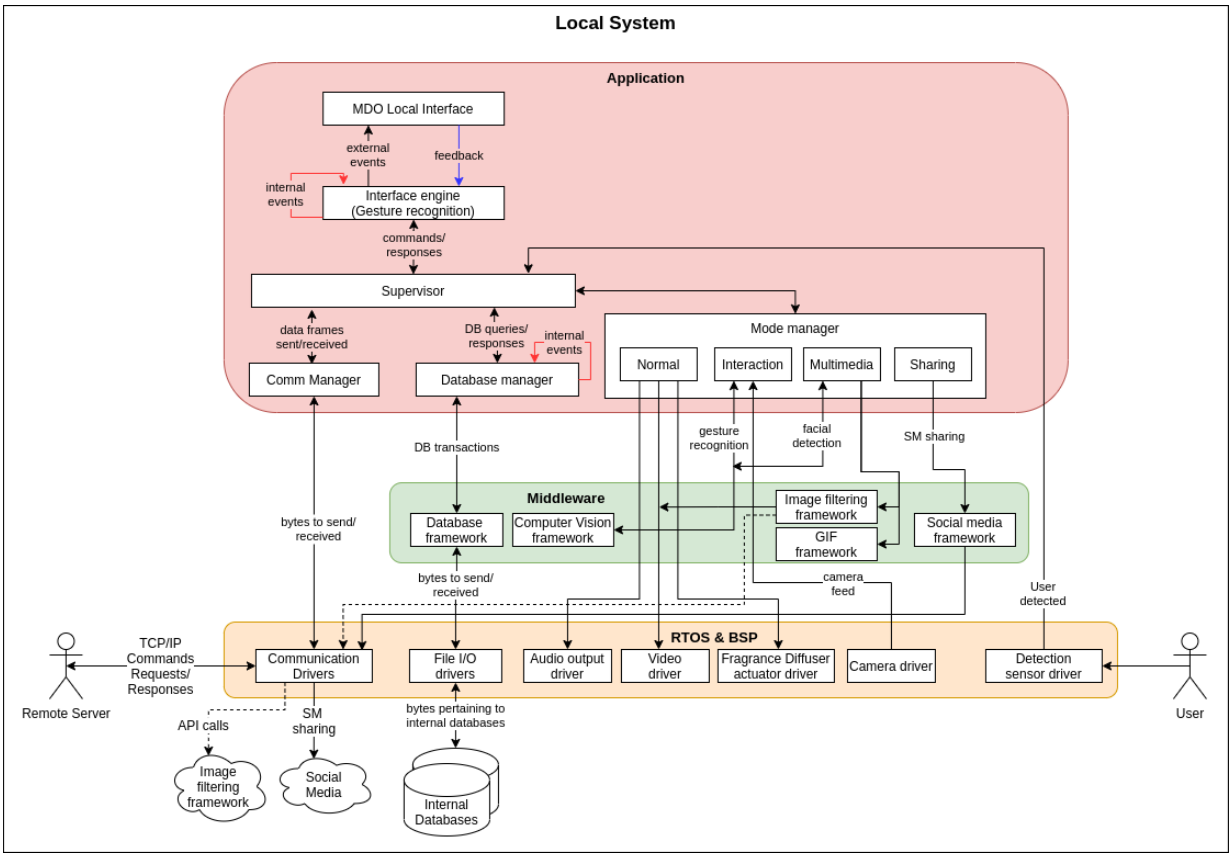


Figure 2.5.: SW architecture diagram: local system

2.4.1. Remote Client

In this section the remote client is analyzed, considering its events, use cases, dynamic operation and the flow of events.

User mock-ups

In Fig. 2.6 is illustrated the user mock-ups for the remote client. It intends to clarify how does the UI works for the two actors: Brands and Company (staff).

The initial state of the MDO-RC's UI is depicted in thick border outline: the 'Sign In' window. If the **User** makes a mistake in its username and/or password, it will be shown an error message. Also, the 'Sign In' window has an option to recover the password, triggering the dispatch of an e-mail. If the **User** still remembers its credentials, the app flows through one out of two possibilities: if the user is an admin, goes to the admin main menu, otherwise if the user is a brand, it will appear the brand main menu.

Firstly, the **Admin** workflow:

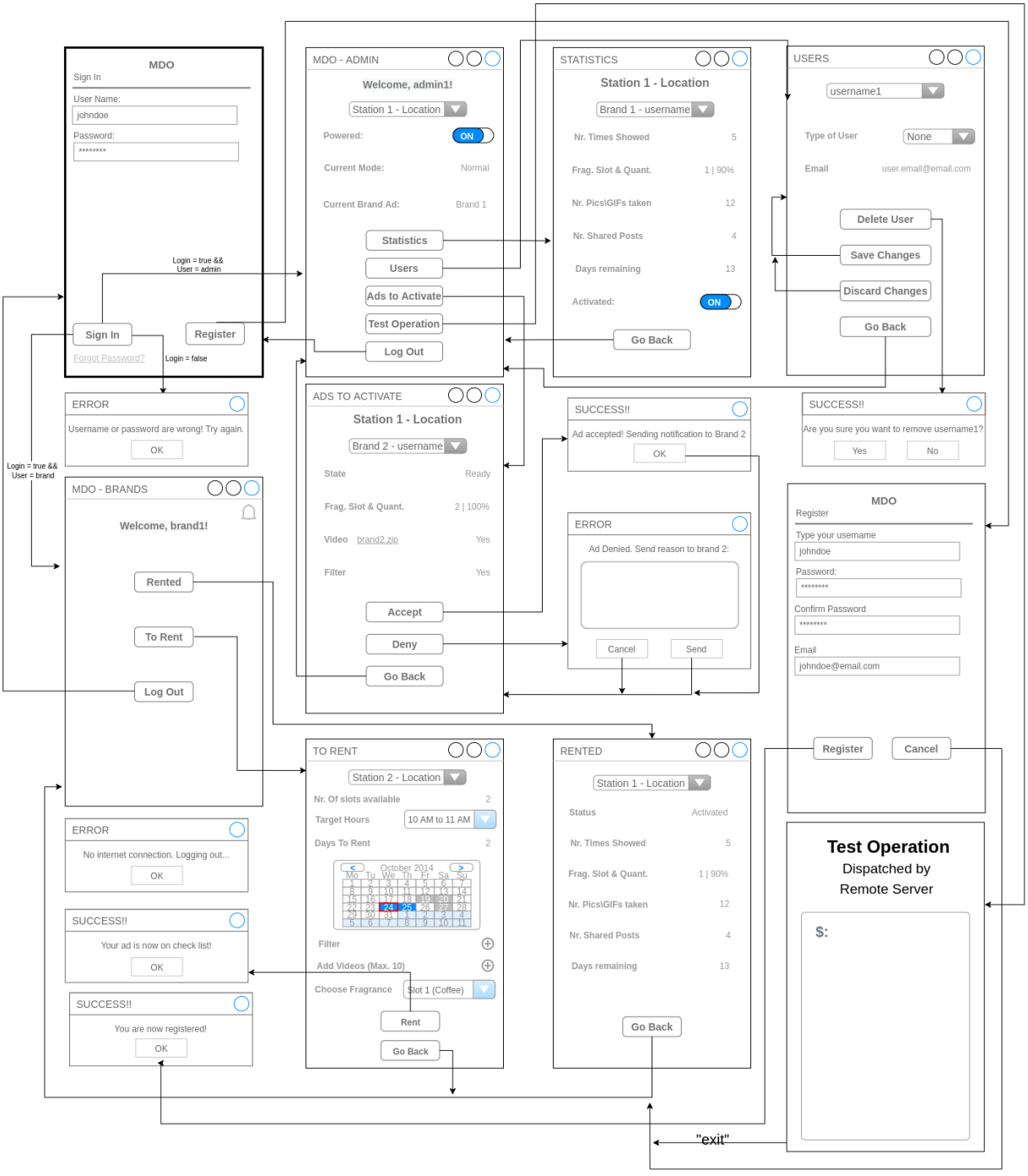


Figure 2.6.: User mockups: remote client

- The **Admin** main menu contains a drop down button with all the available stations. Choosing one of them, the **Admin** can turn it On/Off, see it's current mode and the current brand ad being displayed. Also, the **Admin** can log out and choose between two different paths:

- Statistics: It is possible to see various statistics of all different brands that are currently playing on the station: the number of times that the ad was shown, the number of pictures/GIFs and shared posts, the fragrance slot and quantity (percentage) and the days remaining for the rent to end. It is also possible to deactivate the advertisement if something wrong occurs and go back to the previous menu.
- Users: In this window, the **Admin** can manage all users and see their information, changing their type or deleting them from the database.
- Ads to Activate: In this window, the **Admin** can handle all the ads that the brands are intending to rent. For that, the **Admin** needs to validate the ads': verify video's content (checking if all videos are appropriate), if it has a filter, a fragrance and a time slot. After that, the **Admin** can either accept or deny the ad. If it accepts the ad, it is shown a success message and the ad is added to the station with its preferences. Otherwise, the **Admin** indicates the denial reason, which is subsequently sent to the **Brand**'s email.

Secondly, the **Brand** workflow:

- The **Brand** main menu contains a welcome message, a notification bell to see if another ad was accepted or denied and three buttons - Rented, To Rent and Log Out. The 'Log Out' button logs the **Brand** out of its account, the other two buttons switch to different widgets:
  - Rented: The **Brand** can see all statistics of all its rented ads on different stations that it rented. The statistics are: status, number of times the ad was shown, the fragrance slot and quantity (percentage), the number of pictures/GIFs taken, the number of shared posts and the number of days remaining to end its rent.
  - To Rent: The **Brand** can rent ads in the same station or in other stations. For that, the **Brand** selects the target hours and then a calendar displays the available dates. Then after choosing the days, the **Brand** needs to upload a filter and a compressed multimedia archive with a maximum of ten videos. Finally, the **Brand** needs to select the fragrance and select 'Rent'. After that, a success message will be shown and the ad will enter in a waiting queue for an **Admin** to validate.

It is also possible to register a new user through the 'Register' button. This opens a window to type a username, a password, confirm the password and the e-mail. If everything is in order, the user is created with the default user type of Brand.

Finally, at any time, it can occur the loss of internet connection, which triggers an error message informing the automatic log out of the account.

Table 2.1.: Events: remote client

Event	System response	Source	Type
Login	The system verifies if the user credentials are correct and what type of user is and asks for data from databases	User	Asynchronous
Verify internet connection	Periodically verify internet connection	Remote Client	Synchronous
Statistics	Request to the Remote Server all the information to show statistics from all stations and brands	User (Admin)	Asynchronous
Accept/Deny ad	Send information to the Remote Server if the ad is either accepted or denied and if so, why	User (Admin)	Asynchronous
Power On/Off Station	Send command to Remote Server to Power On/Off a certain station	User (Admin)	Asynchronous
Rented	Request to the Remote Server all the information to show statistics from all stations the brand rented	User (Brand)	Asynchronous
Rent	Send to the Remote Server all the information of rent from the brand, all the videos and the filter	User (Brand)	Asynchronous
Test Operation	The System dispatches the command kine provided by the Remote Server	User (Admin)	Asynchronous
Forgot Password	Send e-mail to the user that has forgotten his password	User	Asynchronous

Events

Table 2.1 presents the most relevant events for the Local system, categorizing them by their source and synchrony and linking it to the system's intended response.

Use cases

Fig. 2.7 depicts the use cases diagram for the **Remote Client**, describing how the system should respond under various conditions to a request from one of the stakeholders to deliver a specific goal.

The **Admin** and the **Brand** interact with the **Remote Client** and this last interacts with the **Remote Server** to process commands, such as query databases or power on/off machines.

The **Admin** can Manage the Station, which includes Power On/Off Station, Manage Ads to Activate, Enable/Disable an Ad and test its operation. It can also manage users, removing or modifying them. All these use cases are processed from the **Remote Client** and are requested to the **Remote Server**.

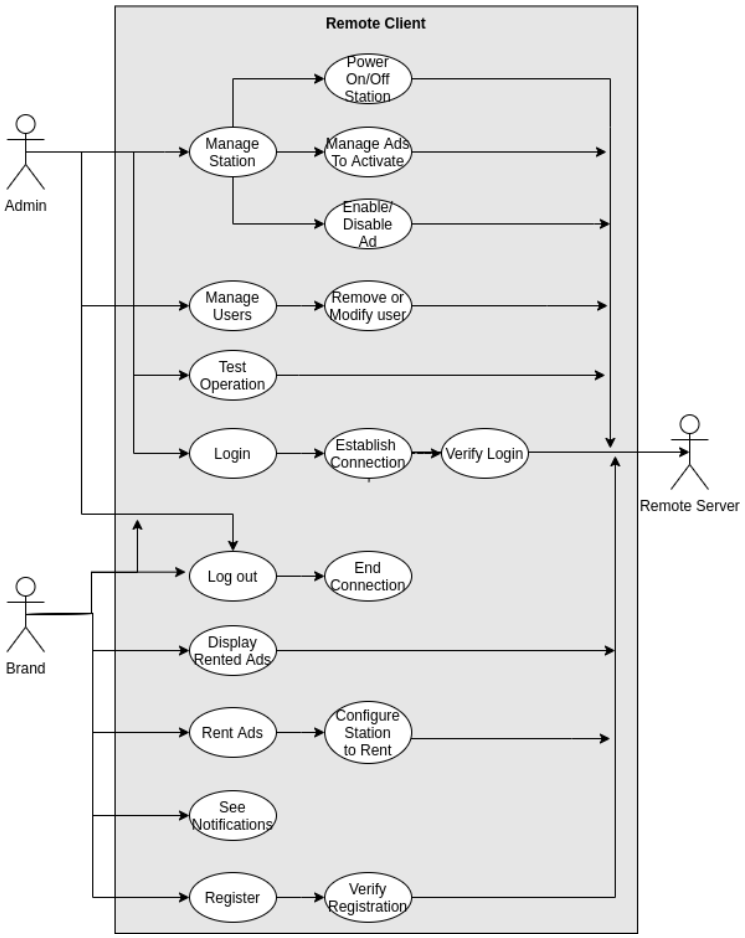


Figure 2.7.: Use cases: remote client

The **Brand** can see Rented Ads, Rent Ads, See notifications and register. All these cases are also processed from the **Remote Client** and are requested to the **Remote Server**.

There are some use cases that are common to the **Admin** and to the **Brand**: Login and Logout.

Dynamic operation

Fig. 2.8 depicts the state machine diagram for the **Remote Client**, illustrating its dynamic behavior.

There are two main states:

- Initialization: the application is initialized. The settings are loaded and if invalid they are restored. The WiFi communication is setup, signaling the communication status and if valid, an IP address is returned.
- Execution: after the initialization is successful, the system goes into the **Execution** macro composite state with several concurrent activities, modeled as composite states too. However, it should be noted

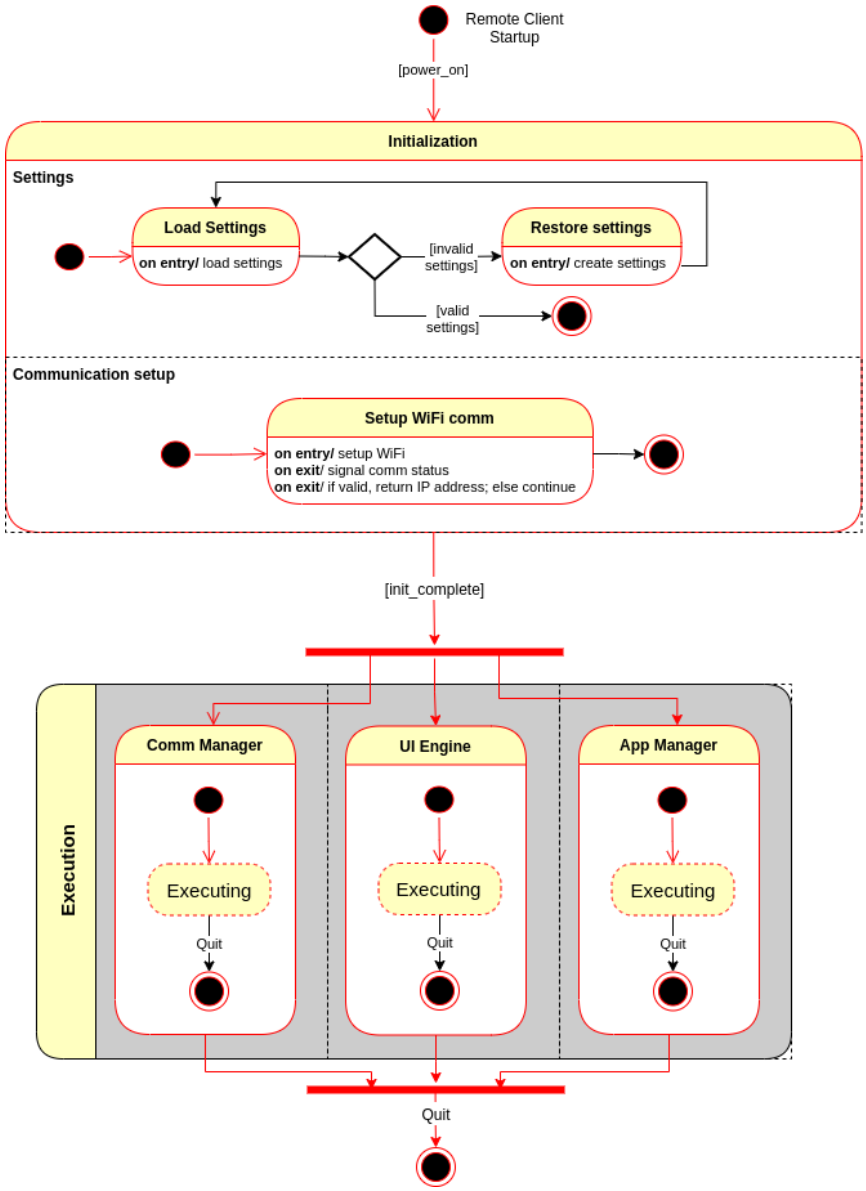


Figure 2.8.: State Machine Diagram: remote client

that there is only one actual state for the device, although at the perceivable time scale they appear to happen simultaneously. These activities are communication management (**Comm Manager**), interface management (**UI Engine**) and application manager (**App Manager**), and are executed forever until system's power off. They are detailed next.

Communication Manager

Fig. 2.34 depicts the state machine diagram for the **Comm Manager** component. Upon successful initialization the **Comm Manager** goes to **Idle**, listening for incoming connections. When a remote node tries to connects, it makes a connection request which can be accepted or denied. If the connection is accepted and the node authenticates successfully the **Comm Manager** is ready for bidirectional communication. When a message is received from the remote node, it is written to **TX msg queue** and the **Supervisor** is notified. When a message must be sent to the remote, it is read from the **TX msg queue** and sent to the recipient. If the connection goes down, it is restarted, going into **Idle** state again.

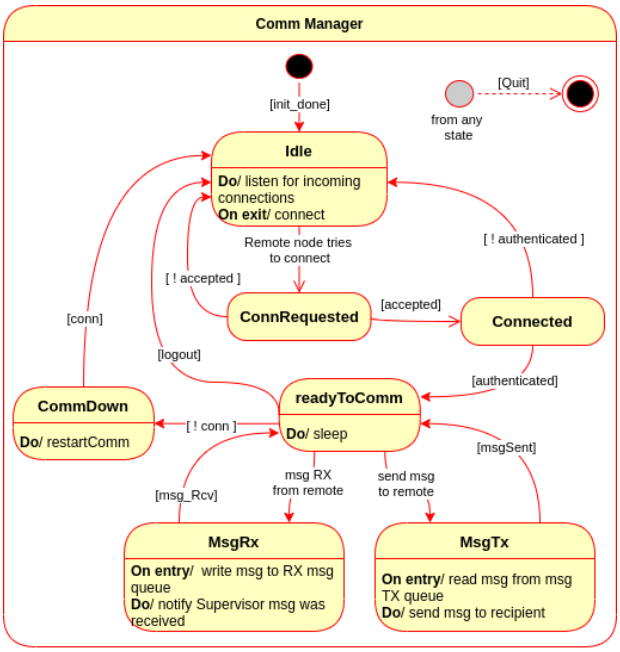


Figure 2.9.: State Machine Diagram: remote client – Communication Manager

App Manager

Fig. 2.10 depicts the state machine diagram for the **App Manager** component. Upon successful initialization the **App Manager** goes to **Login**, waiting for some action.

A user can register itself by pressing the 'Register' button which leads to **Register** state: if succeeds, it returns to **Login** state. If the 'Login' button is pressed, the system goes to **Validation** state, determining its type:

- Admin – **Admin Mode**: the **Admin** has several can view statistics (**Statistics**), manage all users (**Users**), manage all ads to actvivate (**Ads To Activate**) and test operations on the machines (**Test Operation**).

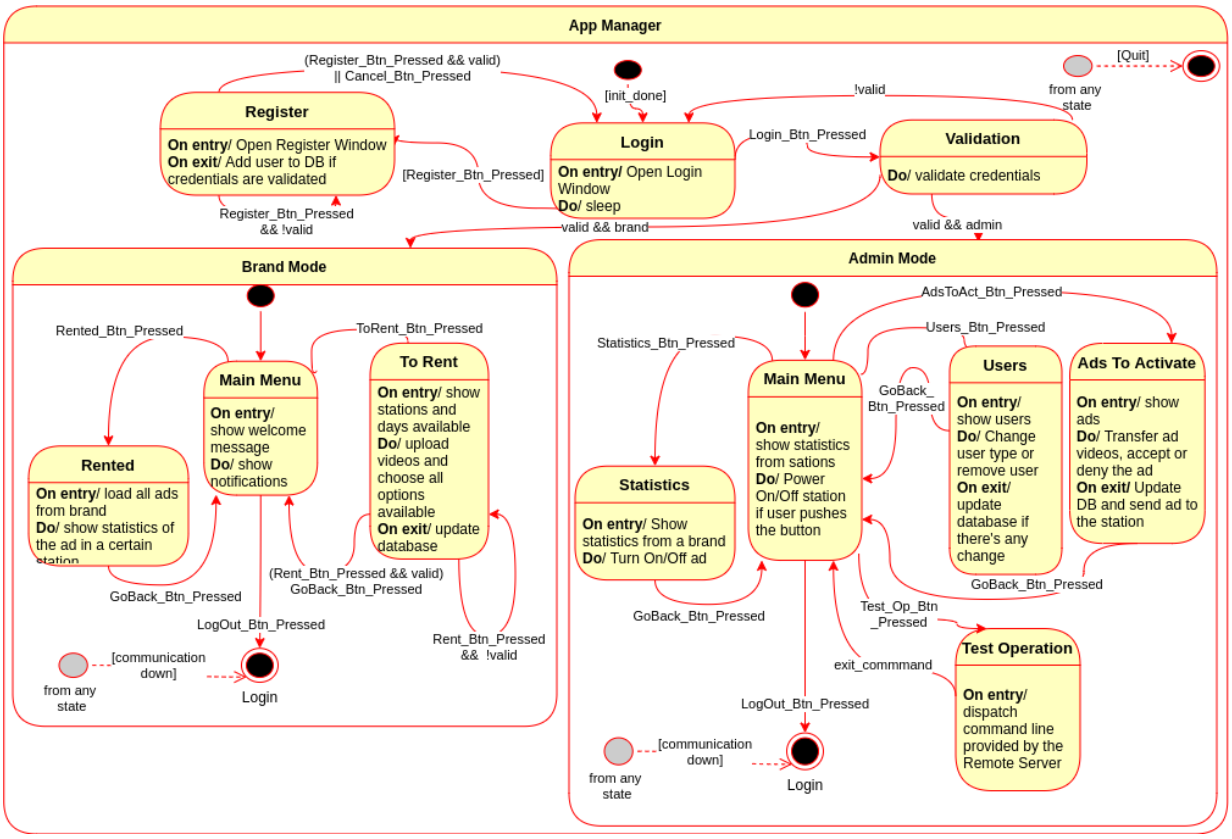


Figure 2.10.: State Machine Diagram: remote client – App Manager

- Brand – **Brand mode**: the **Brand** can see all its ads (**Rented**), see notifications and messages (**Main Menu**) and rent new ads (**To Rent**).

These two states are terminated by pressing the 'Log Out' button, which redirects to **Login** state.

If, in any state, a critical error occurs, that can cause an unexpected quit of the **App Manager**, leading to the application abnormal shutdown.

Flow of events

The flow of events throughout the system is described using a sequence diagram, comprising the interactions between the most relevant system's entities. It is usually pictured as the visual representation of an use case. The main sequence diagrams are illustrated next (Fig. 2.11 through Fig. 2.16).

«««< HEAD

As it can be seen in Fig. 2.11 to Fig. 2.17, the user interacts with the **UI**, then this last interacts with the **UI Engine**, interacting with the **Remote Client Back-End** in order to process and execute all the information and commands needed. There's an alternate way to go to the user, that's because on the authentication

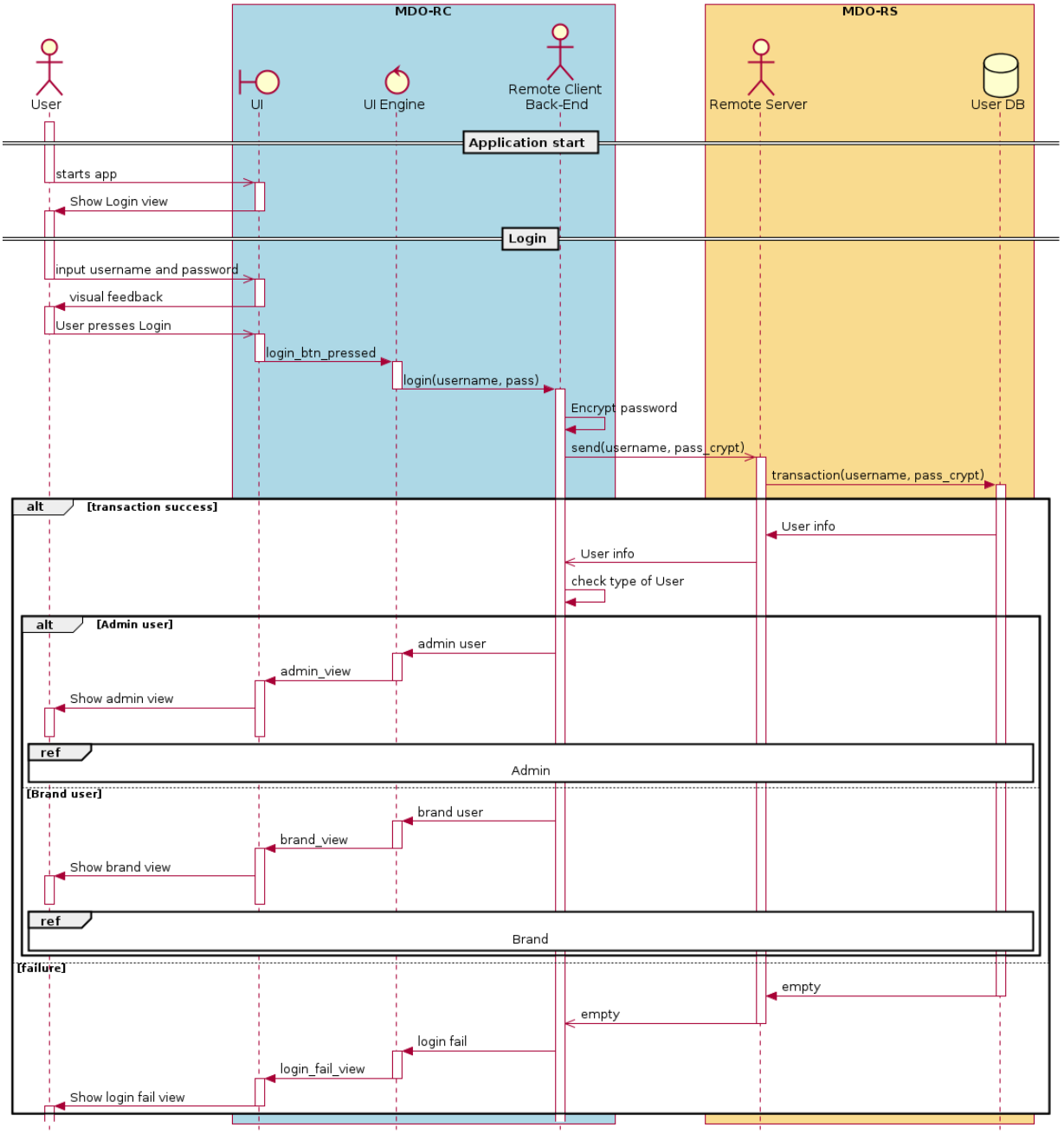


Figure 2.11.: Sequence Diagram: remote client — Login

the **Remote Client Back-End** will discover if the user is an **Admin** or a **Brand**. On both cases, it shows its main menu and it can end the sequence through the 'Logout'. In each one of the cases there's alternative sequences to occur, depending in what the **User** decides to do. Also, in each alternative choice, the **Remote Client** can interact with different **Databases**, either to update them or to ask some info.

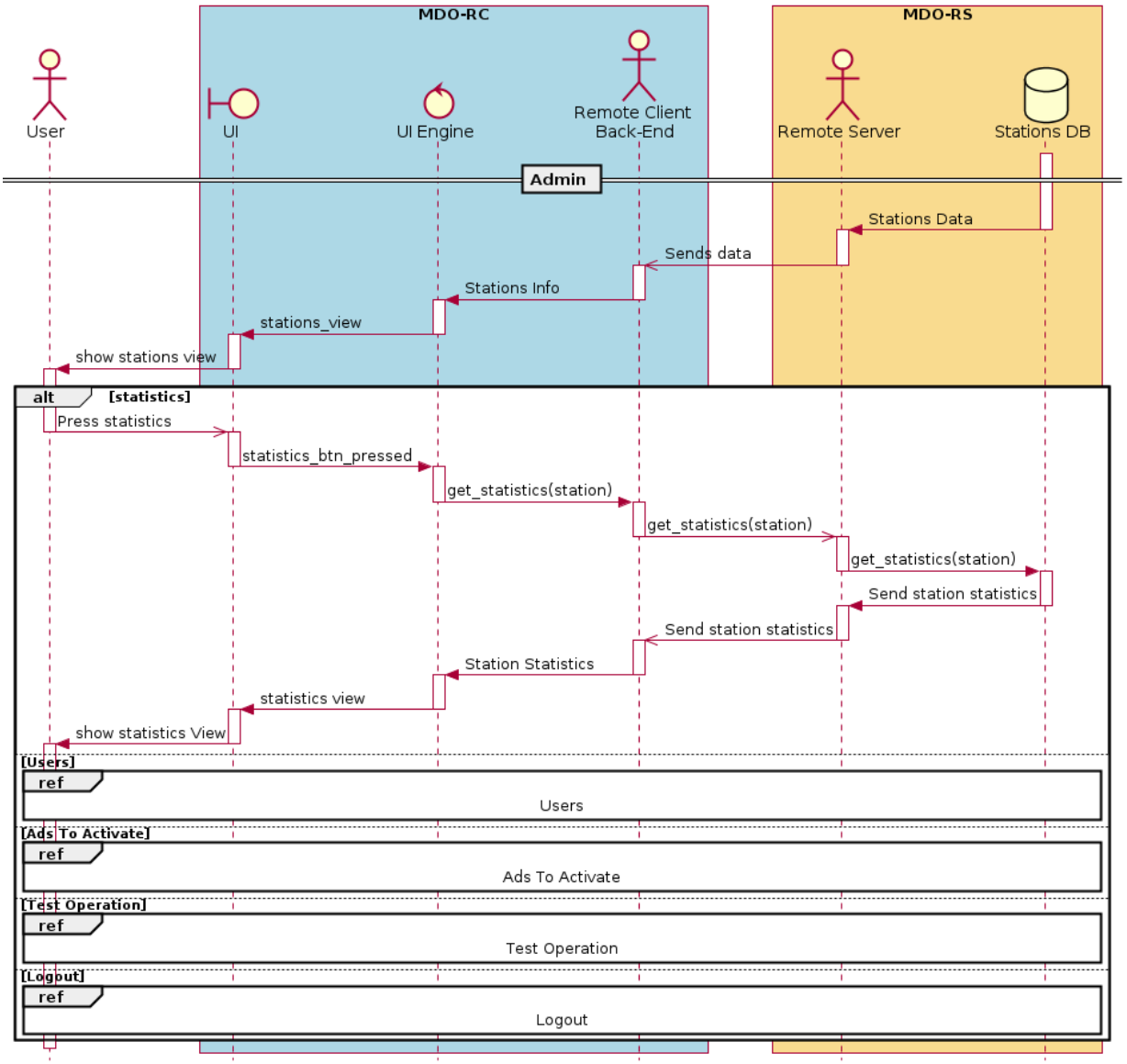


Figure 2.12.: Sequence Diagram: remote client – admin statistics

======= As it can be seen, the **user** interacts with the **UI**, whose events are tracked by the **UI Engine** triggering the appropriate callback and dispatching data to the **Remote Client Back-End** for adequate processing.

There are two flow paths, pertaining to type of **User** – **Admin** or **Brand** – as a result of the **User authentication**. On both cases, it shows its main menu and it can end the sequence through the 'Logout'.

In each one of the cases there's alternative sequences to occur, depending of what the **User** decides to do. Also, in each alternative choice, the **Remote Client** can interact with different **Databases**, either to

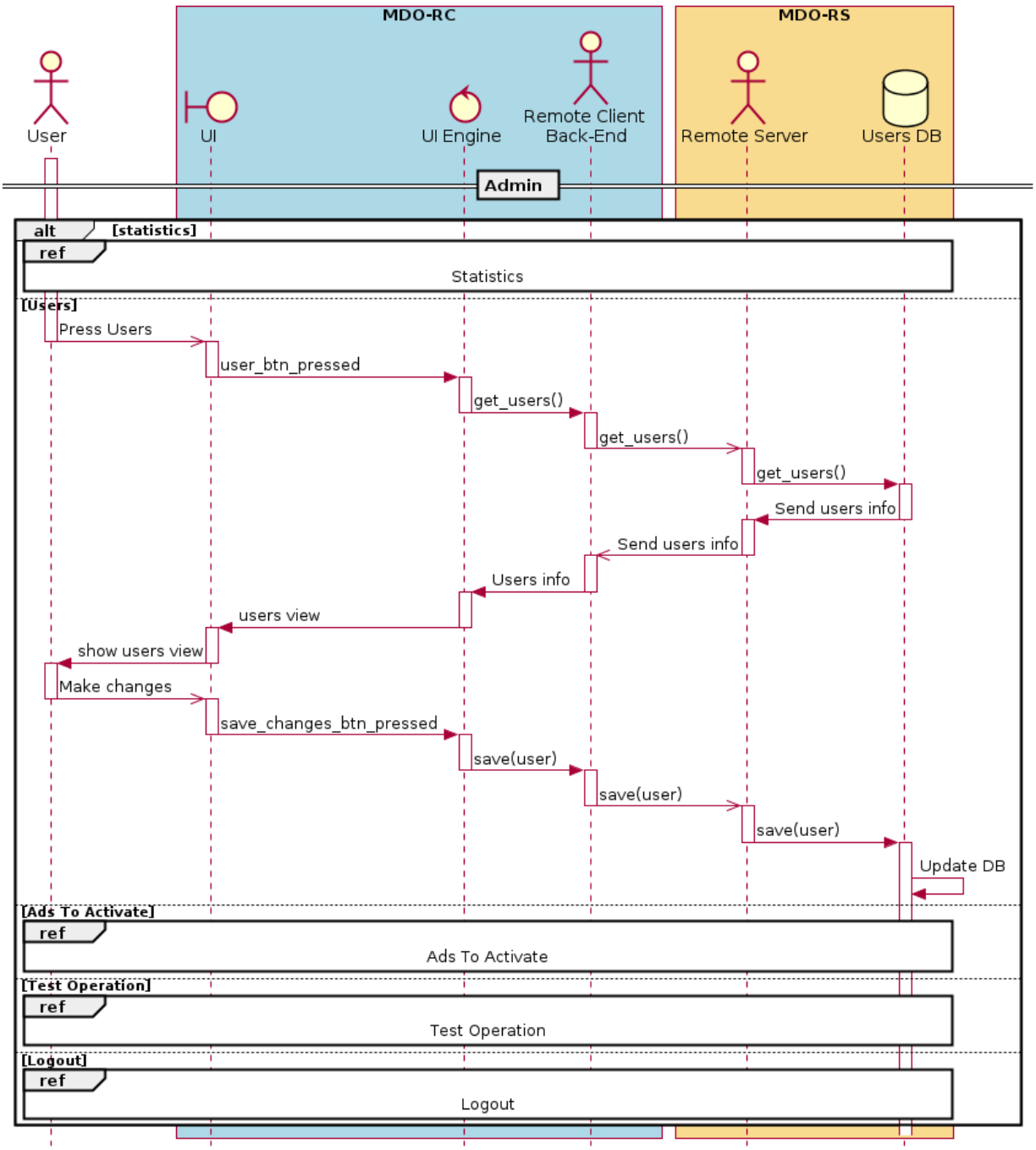


Figure 2.13.: Sequence Diagram: remote client — admin users

query or update them. »»»> 1b64a1fa2e419d79801eaaa806f4d9675f06902e

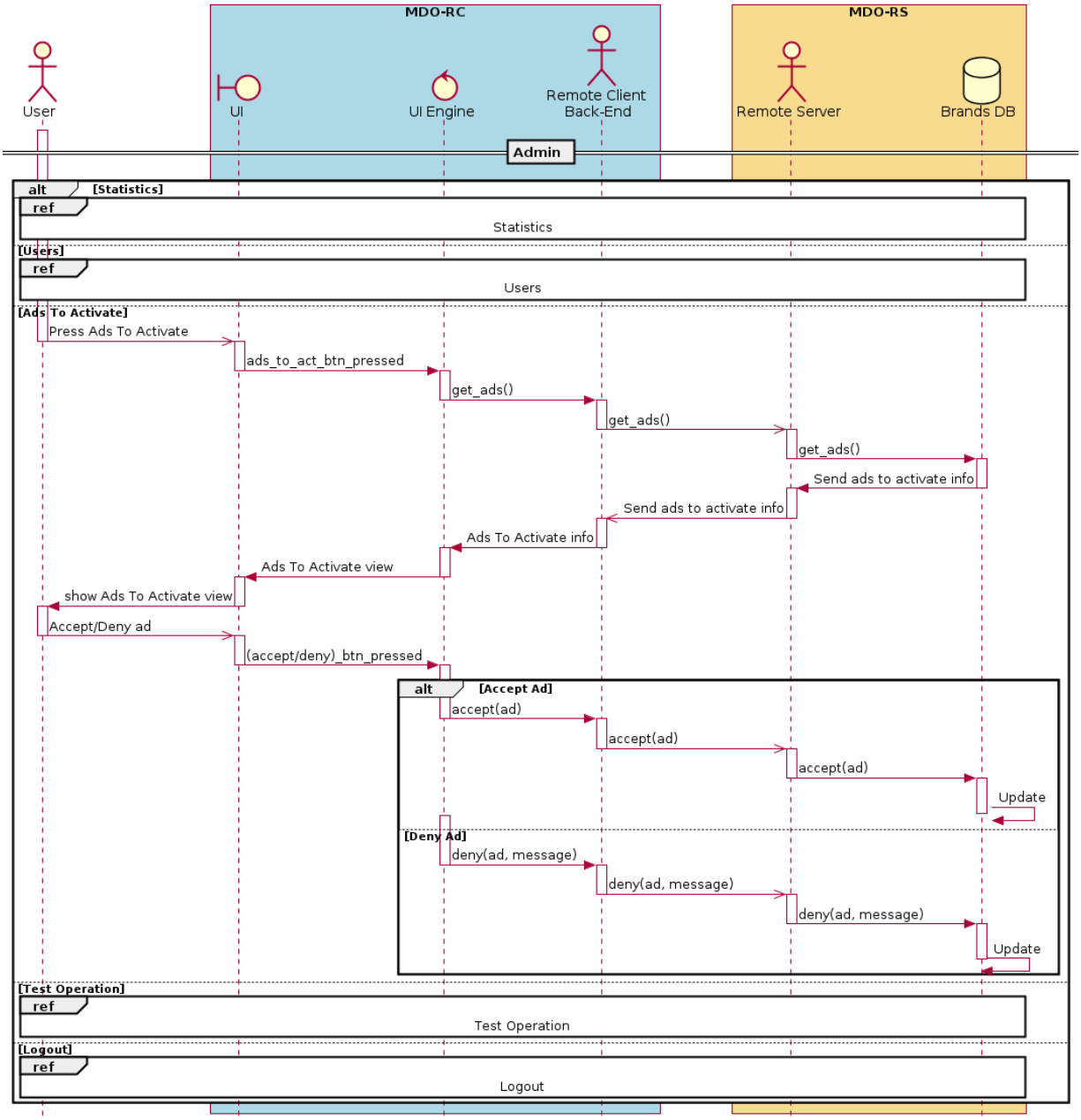


Figure 2.14.: Sequence Diagram: remote client — admin ads to activate

2.4.2. Remote server

In this section the remote server is analyzed, considering its events, use cases, dynamic operation and the flow of events.

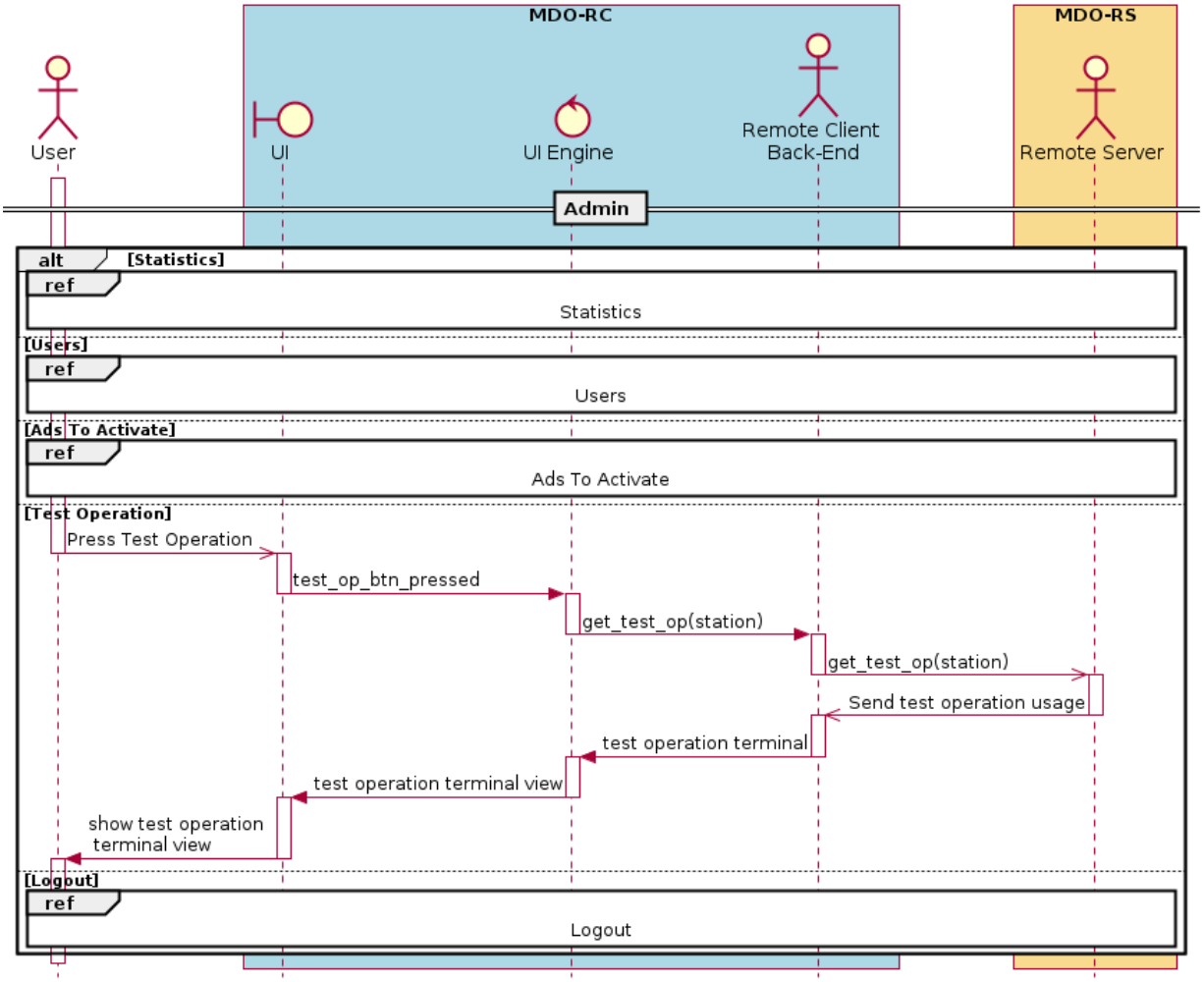


Figure 2.15.: Sequence Diagram: remote client — admin test operation

User mock-ups

Fig. 2.18 illustrates the user mock-ups for the **Remote Server**. It intends to mimic the user interaction with the **Remote server** interface, clarifying the user actions and the respective responses, as well as the workflow, defining the **Remote Server** interface.

It consists of a CLI providing basic commands to authenticate an user, perform operations over a DB and test the operation of a designated **Local System** (only available to administrator users).

To test the operation of a **Local System**, an **Admin** can:

- Normal mode: add, delete, play or stop video, audio and fragrance;
- Interactive & Multimedia modes — camera: turn on/off the camera, apply facial detection, use an image filter, take a picture or create a GIF;

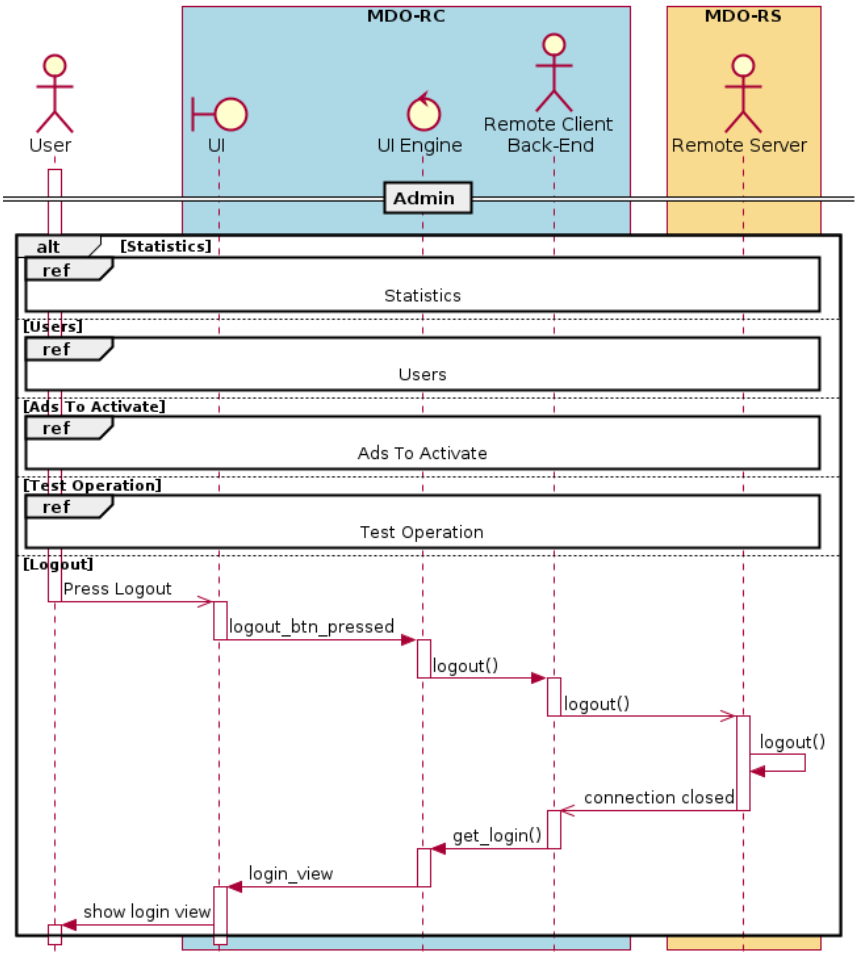


Figure 2.16.: Sequence Diagram: remote client — admin logout

- Sharing mode: share to a designated social media network a post, containing a message and attachment (picture or GIF).

Events

Table 2.2 presents the most relevant events for the **Remote Server**, categorizing them by their source and synchrony and linking it to the system's intended response.

Use cases

Fig. 2.19 depicts the use cases diagram for the **Remote Server**, describing how the system should respond under various conditions to a request from one of the stakeholders to deliver a specific goal.

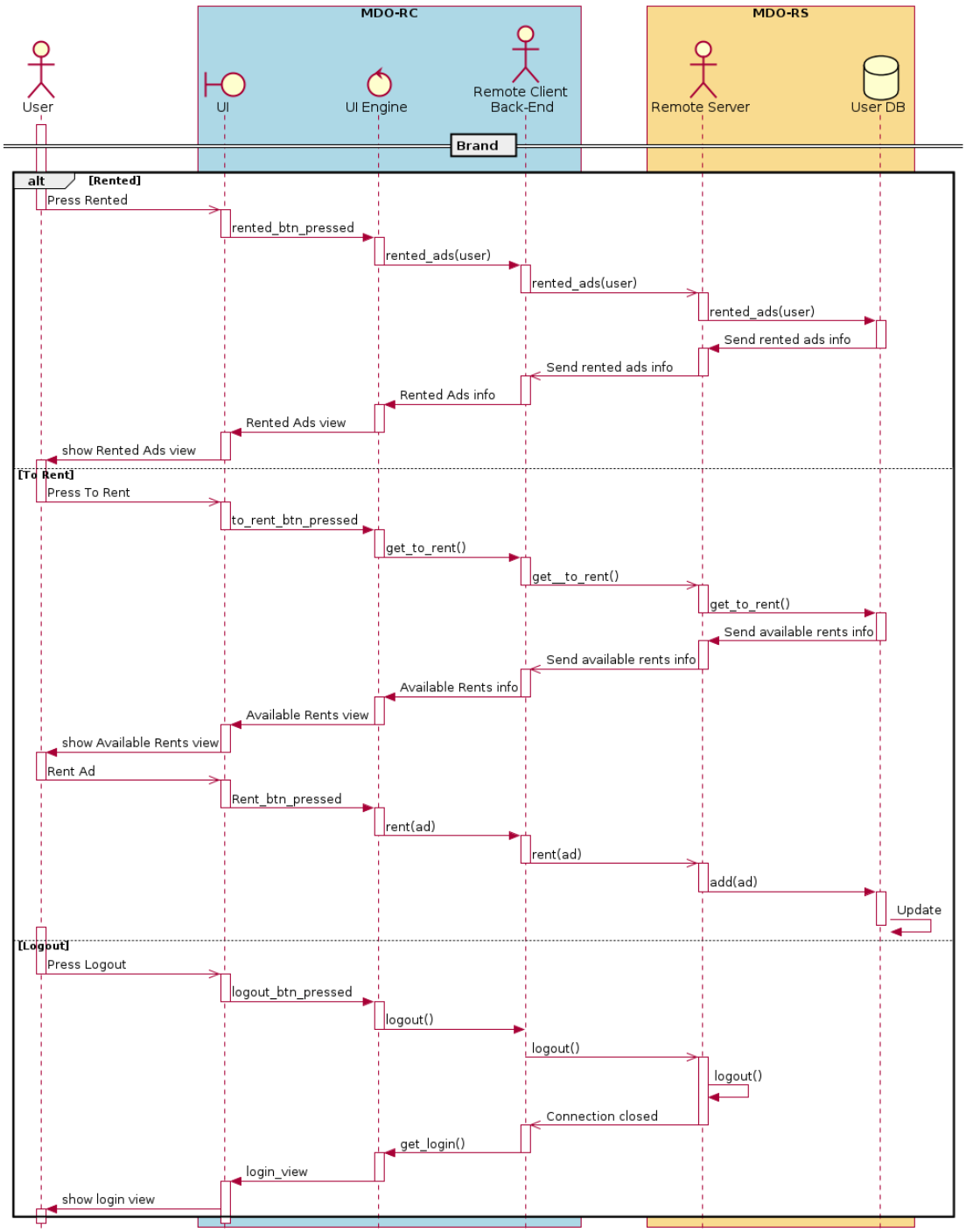


Figure 2.17.: Sequence Diagram: remote client - brand

As it can be seen, the **Remote Client** can interact through various modes: **Help**, **Authenticate User**, **Interact with databases**, **Test Operation** and **Disconnect**.

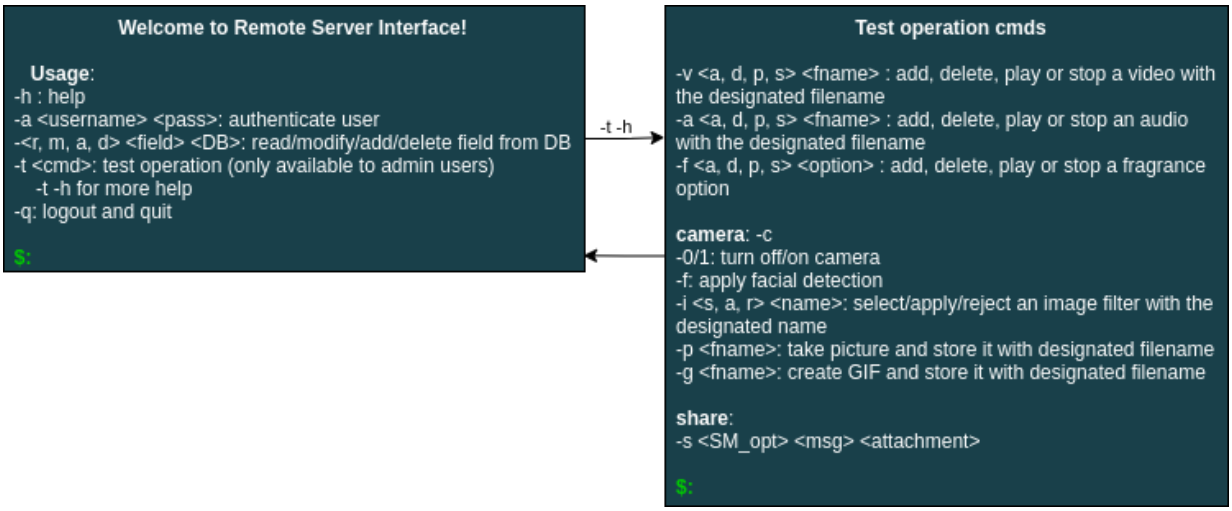


Figure 2.18.: User mock-ups: Remote Server

Table 2.2.: Events: Remote Server

Event	System response	Source	Type
Power on	Initialize RDBMS and go to Idle mode	System maintainer	Asynchronous
Connection Requested	Accept/refuse connection	Remote Client	Asynchronous
Connection Accepted	Start listening for commands	Remote Client	Asynchronous
Authenticate	Query User DB to validate user credentials. If valid, login user.	Remote Client	Asynchronous
Help	Send help information	Remote Client	Asynchronous
Logout	Logout user, close connection and go to Idle mode	Remote Client	Asynchronous
Check WiFi connection	Periodically check WiFi connection	Remote Client	Synchronous
Connection timeout	Logout user, close connection and go to Idle mode	Remote Server	Synchronous
DB management	Read/modify/add/delete data from DB	Remote Client	Asynchronous
Update stations	Update all ready-to-run stations with ads data	Remote Server	Synchronous
Command invalid	Inform RC that command is invalid	Remote Server	Synchronous
Station notification	Store station notification into DB	Local System	Asynchronous
Test Operation RC	Parse command originated from RC and, if valid, dispatch it to designated station	Remote Client (Admin)	Asynchronous
Test Operation Callback	Provide command dispatch to original RC	Local System	Asynchronous

When interacting with the databases, it is possible to read, modify, add or delete some field from a **Database**. The operation of the **Local System** can be tested — **Test Operation** — if the **User** is an **Admin**, namely: **manage audio, video, fragrance and camera** and test the **share** option.

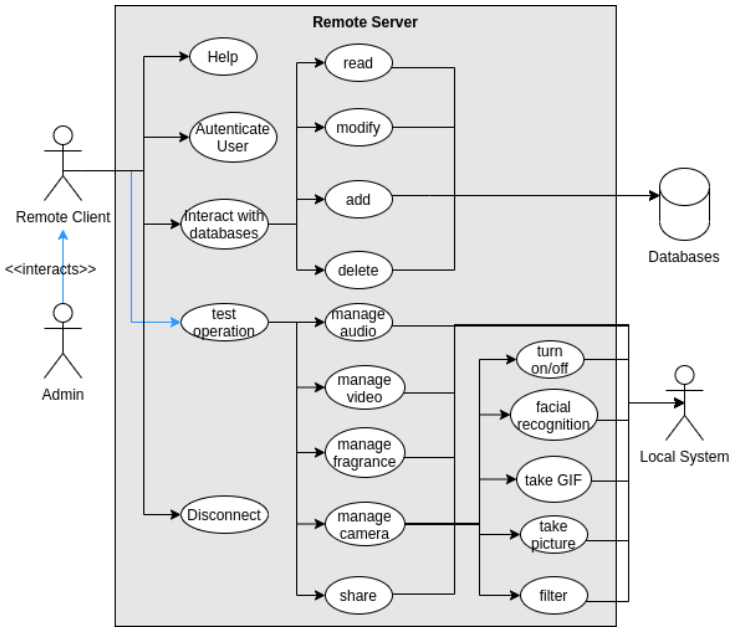


Figure 2.19.: Use cases: remote server

The **Manage Camera** use case is subdivided into: **Turn On/Off, facial detection, take GIF, take picture** and **filter**.

Dynamic operation

Fig. 2.20 depicts the state machine diagram for the **Local System**, illustrating its dynamic behavior. There are two main states:

- Initialization: the **Remote Server** is initialized. The settings are and DBs are loaded and if invalid they are restored. The WiFi communication is setup, signaling the communication status and if valid, an IP address is returned. Lastly, the RDBMS is configured and started: if any error occurs the device goes into the **Critical Error** state, dumping the error to a log file and waiting for reset; otherwise, the initialization is complete.
- Execution: after the initialization is successful, the system goes into the **Execution** macro composite state with several concurrent activities, modeled as composite states too. However, it should be noted that there is only one actual state for the device, although at the perceivable time scale they appear to happen simultaneously. These activities are communication management (**Comm Manager**), DB management (**DB manager**), and request handling (**Request Handler**), and are executed forever until system's power off. They are detailed next.

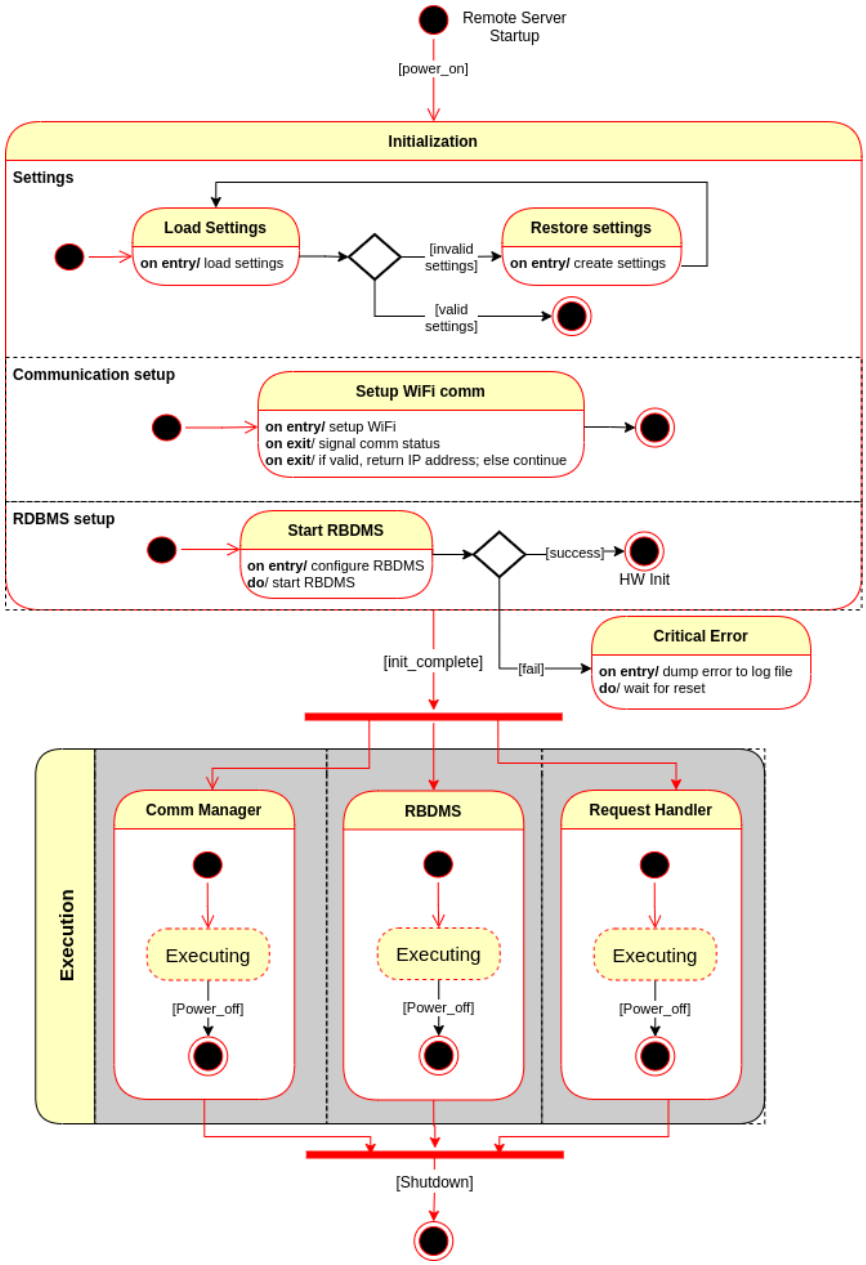


Figure 2.20.: State machine diagram: Remote server

Communication Manager

Fig. 2.21 depicts the state machine diagram for the **Comm Manager** component. Upon successful initialization the **Comm Manager** goes to **Idle**, listening for incoming connections. When a remote node tries to connects, it makes a connection request which can be accepted or denied. If the connection is accepted and the node authenticates successfully the **Comm Manager** is ready for bidirectional communication. When a message is received from the remote node, it is written to **TX msg queue** and the **Request Handler**

is notified. When a message must be sent to the remote, it is read from the **TX msg queue** and sent to the recipient. If the connection goes down, it is restarted, going into **Idle** state again.

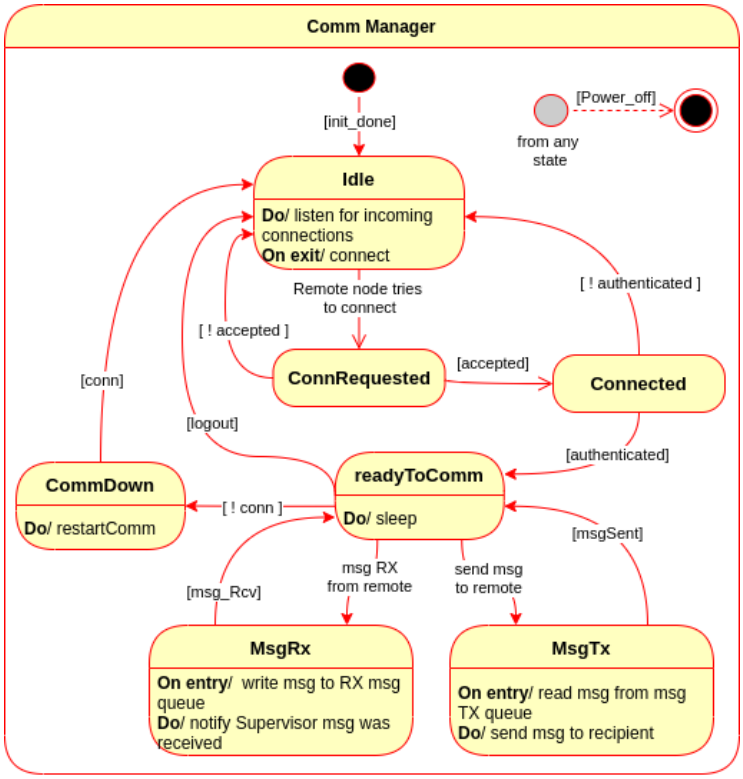


Figure 2.21.: State machine diagram: Remote Server — Comm Manager

Database Manager

Fig. 2.22 depicts the state machine diagram for the **DB Manager** component. Upon successful initialization the **DB Manager** goes to **Idle**, waiting for incoming DB requests.

When a request arrives, it is parsed, checking its validity. If the request is a DB query, a transaction is read from the respective DB to the **RX transaction queue** and the **Supervisor** is notified that there is a transaction to read. Otherwise, if the request is a DB update the transaction is written from the **TX transaction queue** to the DB and the **Supervisor** is notified that the DB was updated.

Alternatively, the RDBMS can be triggered to update a station (**update_station** event), retrieving the station and operation data to update. A data frame is composed and a server request is created, signaling it to the **Request Handler** to process it.

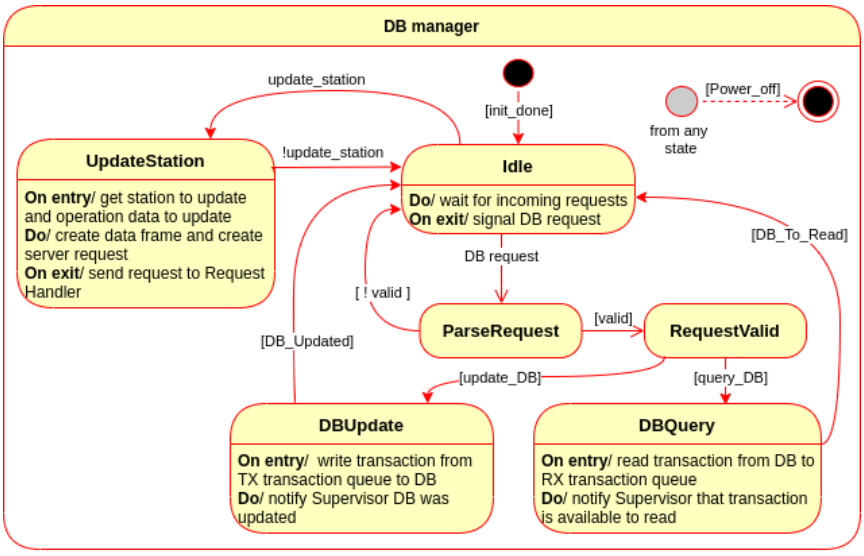


Figure 2.22.: State machine diagram: Remote Server – DB Manager

Request Handler

Fig. 2.23 depicts the state machine diagram for the **Request Handler** component, which handles incoming requests from the **Remote Client**, **Local System**, or internally (to update stations). When a request arrives, it is parsed, and, if valid, the appropriate callback is triggered, processing the request and returning its output.

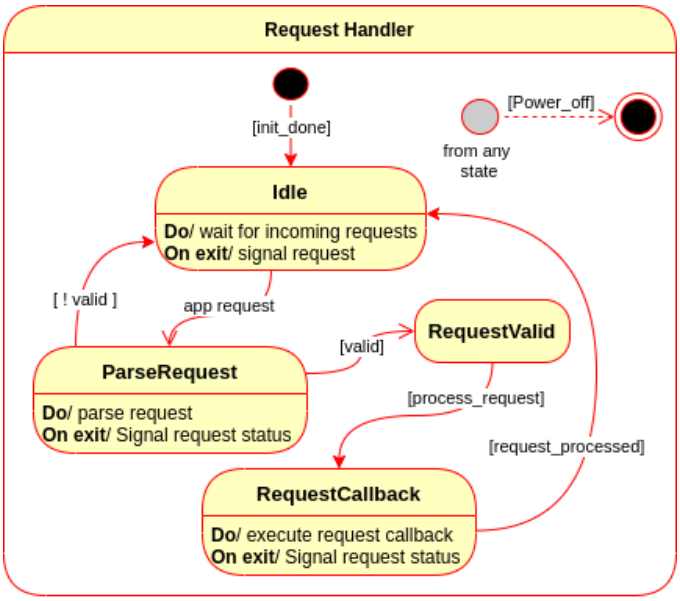


Figure 2.23.: State machine diagram: Remote Server – Request Handler

Flow of events

The flow of events throughout the system is described using a sequence diagram, comprising the interactions between the most relevant system's entities. It is usually pictured as the visual representation of an use case. The main sequence diagrams are illustrated next (Fig. 2.24 through Fig. 2.30).

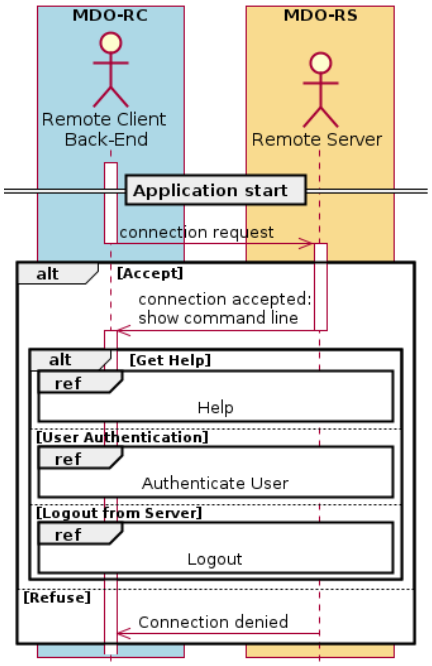


Figure 2.24.: Sequence diagram: Remote Server

The main interaction in all diagrams occurs between the **Remote Client** and the **Remote System**.

In first place, Fig. 2.24 shows the **Application Start**, starting with a connection request that it can be either accepted or denied.

In second place, Fig. 2.25 shows the **Authentication** process, where the **Remote Client** sends the username and the password to the **Remote Server**. The latter searches for the username in the **User DB**, if it is not found, the authentications fails, otherwise, the **Remote Server** compares the passwords and sends the authentication status to the **Remote Client**. It is only possible to access the **Manage DBs** and the **Test Operation** if the authentication succeeds.

In Fig. 2.26 the **Remote Client** requests to the **Remote Server** to manage a **Database**'s field. The **Remote Server** makes the request to the specific **Database**, receives the response and returns it to the **Remote Client**.

In Fig. 2.27, Fig. 2.28 and Fig. 2.29 it can be seen the **Test Operation**. As said previously, only the **Admin** can access this part, where it can test the functionality of any **Local System**. The **Admin** sends

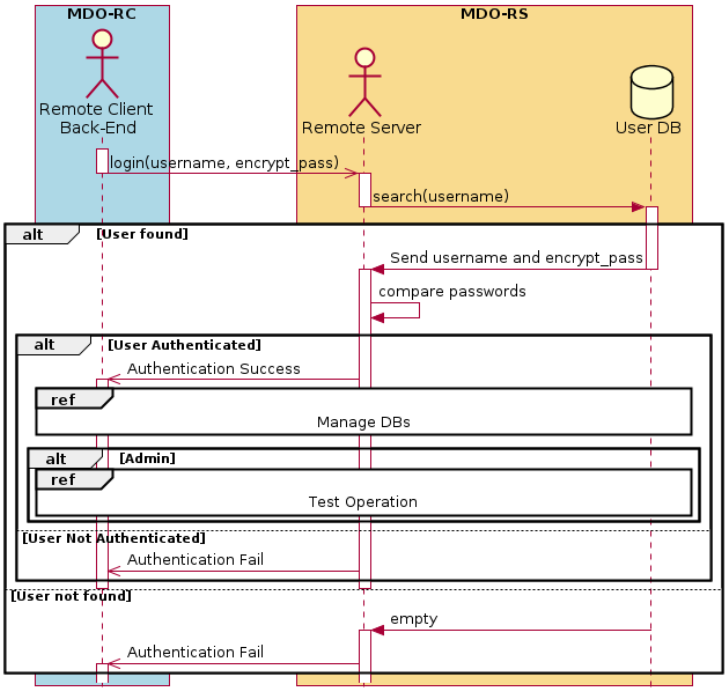


Figure 2.25.: Sequence diagram: Remote Server – Authentication

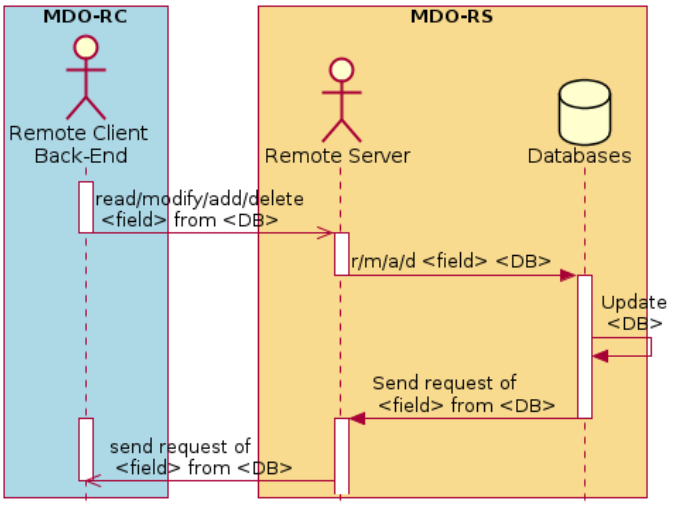


Figure 2.26.: Sequence diagram: Remote Server – Manage Databases

the request to the **Remote Client** that then sends it to the **Remote Server** with the specific station. Then, the **Remote Server** interacts with the specific station, making the specific **Local System** act according to the command that was sent. For every test operation there's always a command feedback to indicate to the **User** its execution status. The references **Apply Filter**, **Take Picture** and **Create GIF** in Fig. 2.28 and **Sharing Mode** in Fig. 2.29 are the responsibility of the **Local System** part and will be explained there.

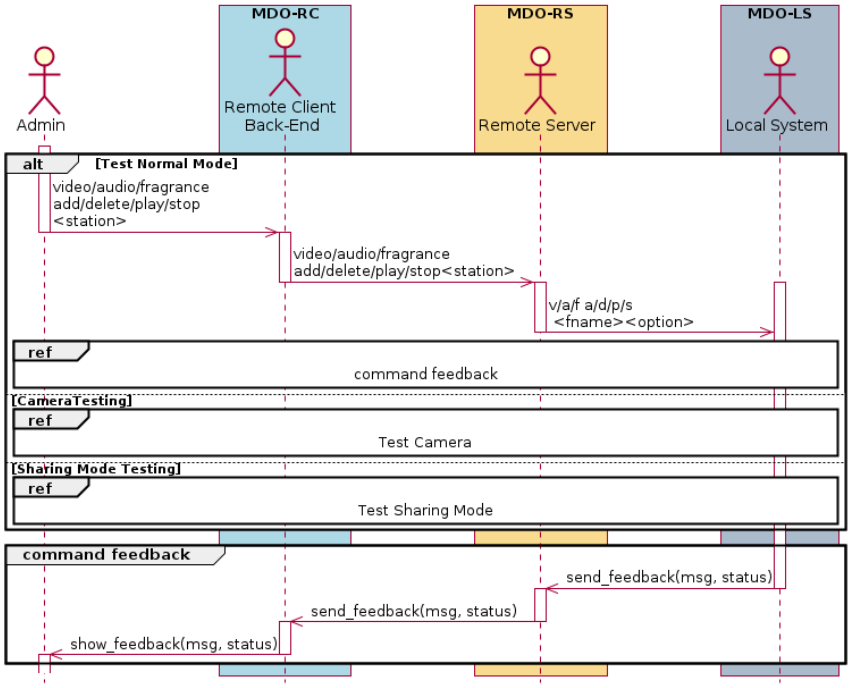


Figure 2.27.: Sequence diagram: Remote Server — Test Operation

2.4.3. Local system

In this section the local system is analyzed, considering its events, use cases, dynamic operation and the flow of events.

User mock-ups

Fig. 2.31 illustrates the user mock-ups for the local system. It intends to mimic the user interaction with the local system, clarifying the user actions (gestures) and the respective responses, as well as the workflow, comprising its four modes.

The initial state of the MDO-L's UI is depicted in thick border outline, after a **User** has been detected — **Interaction mode**. On the left it is the camera feed and on the right the commands ribbon, containing the hints to use the system and the available options. As it can been, the **User** can choose an option by hovering with pointing finger over the desired option for a designated amount of time (e.g., 3 seconds).

The workflow can be as follows:

- If the **User** selects the **Image filter** option, the **Image filtering** view is shown, presenting the options to select filters (which can be scrolled through palm raising/lowering), to cancel or accept the image filter. If a filter is selected **filter1_pressed**, it is applied, and if accepted it will return to **Interaction**

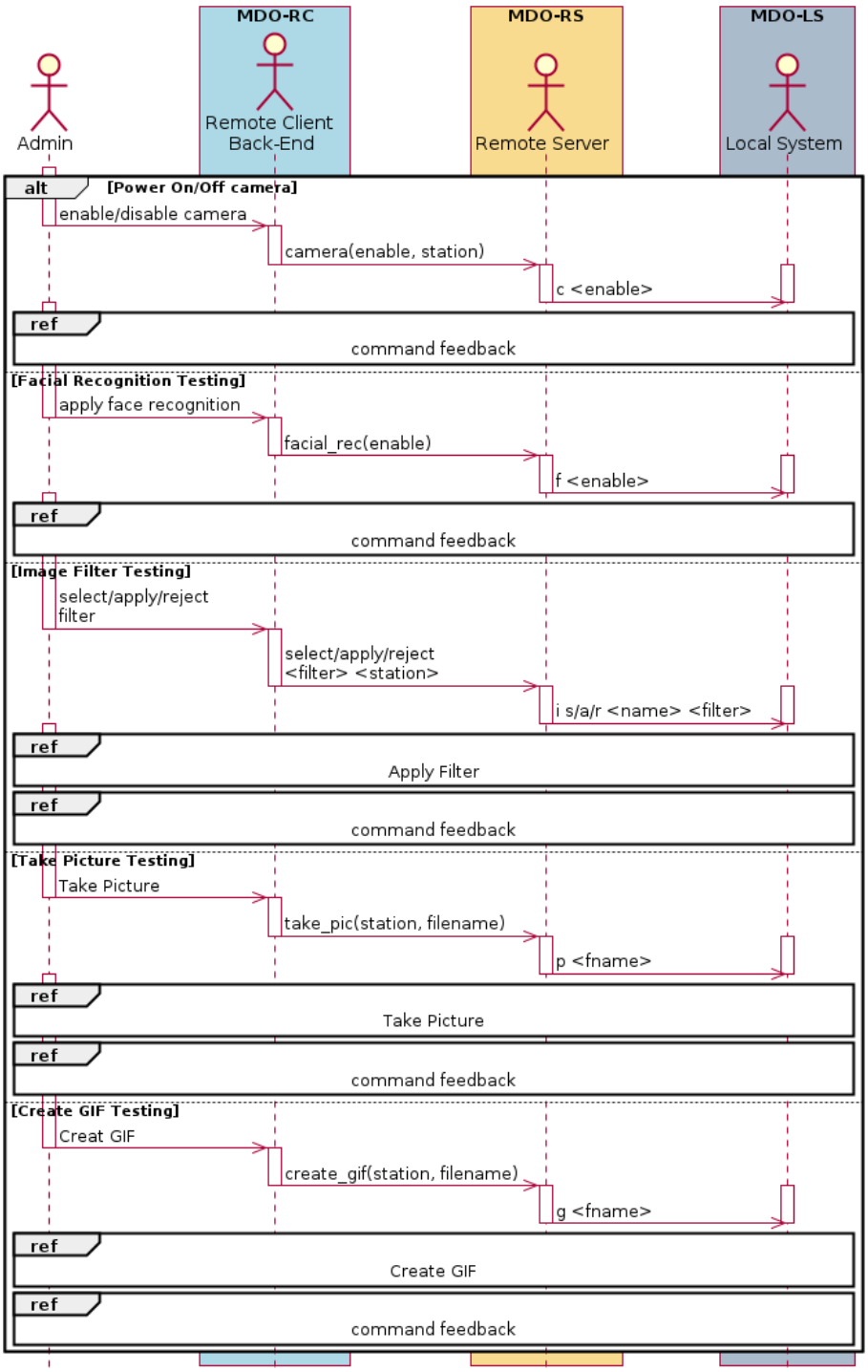


Figure 2.28.: Sequence diagram: Remote Server – Test Operation Camera

mode, keeping the filter on.

- If the User selects the Take Pic option, Picture mode is started with a timer to allow the User to get

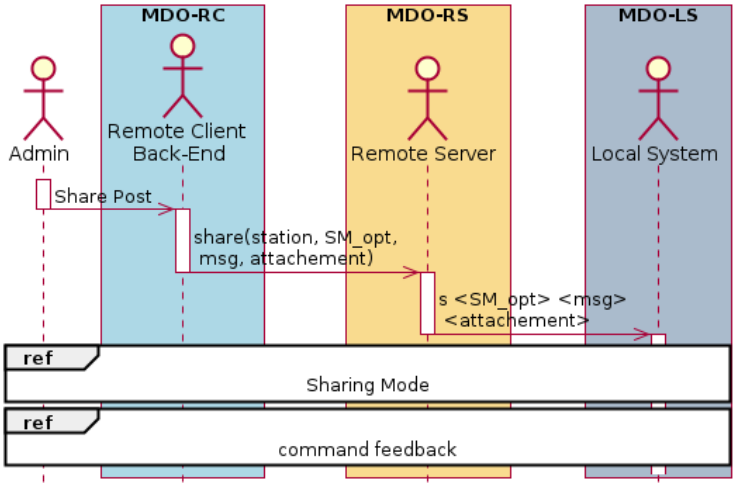


Figure 2.29.: Sequence diagram: Remote Server – Test Operation Share

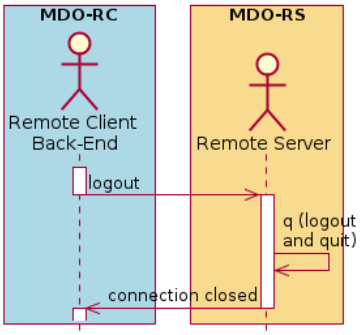


Figure 2.30.: Sequence diagram: Remote Server – logout

ready before actually taking the picture. The **User** can **Cancel** – returning to main menu – or **Share** – starting **Sharing mode**.

- If the **User** selects the **Create GIF** option, **GIF mode (setup)** is started with a timer to allow the **User** to get ready before actually creating the GIF. After the **setup_timer** is elapsed, the **GIF mode (operation)** starts, displaying a dial with the GIF duration until being complete. When the **gif_timer** elapses, the GIF is created, enabling the **User** to **Cancel** – returning to main menu – or to **Share** – starting **Sharing mode**.
- Lastly, in the **Sharing mode**, the **User** can **Cancel** – returning to main menu – or select the social media network. After selecting the social media, the **User** can edit the post by entering its customized message and, if **Share** is pressed, a message box will appear displaying the status of the post sharing – **Success** or **Error**.

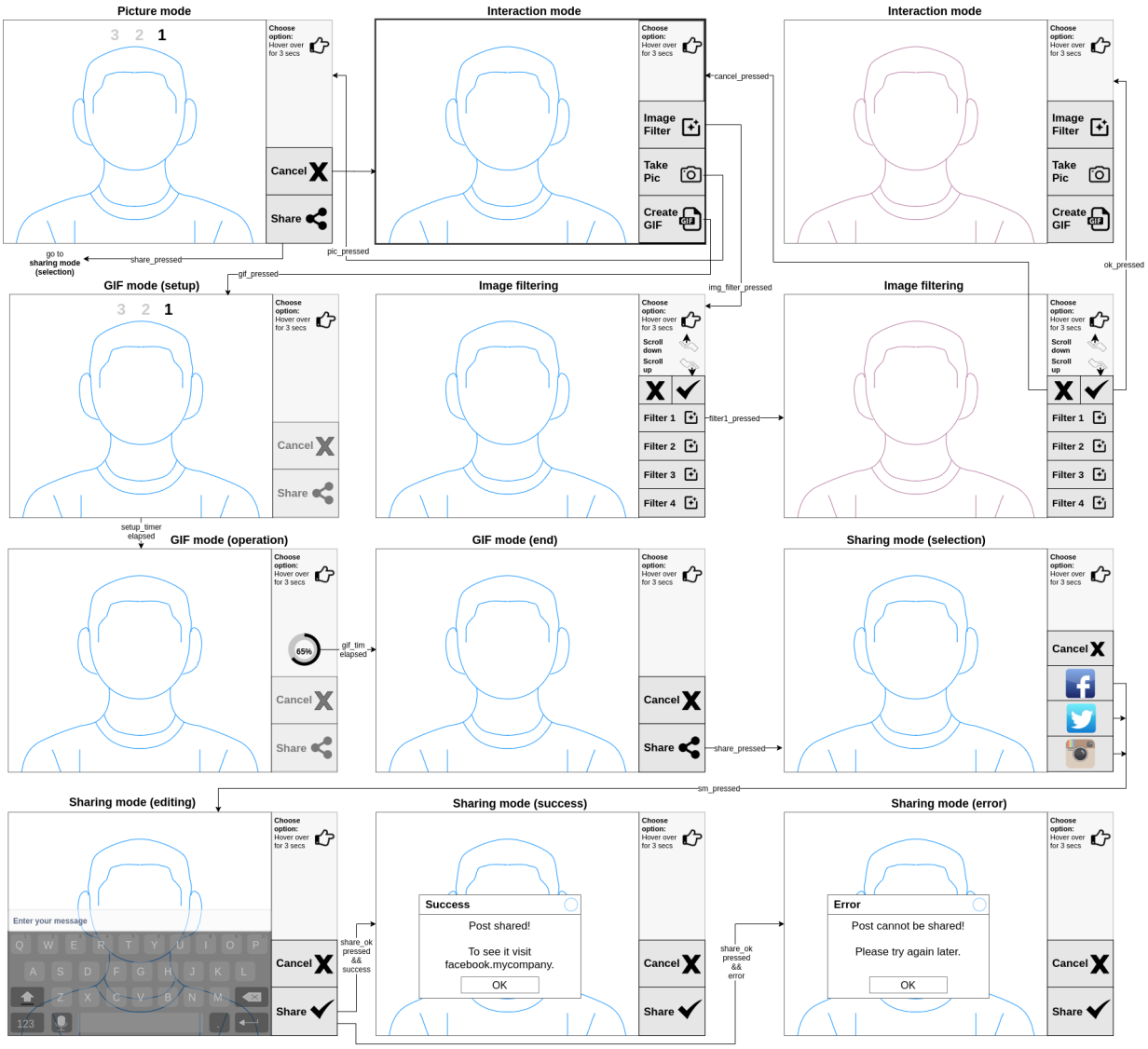


Figure 2.31.: User mock-ups: local system

Events

Table 2.3 presents the most relevant events for the **Local system**, categorizing them by their source and synchrony and linking it to the system's intended response. A further division is done separating UI events from the remaining ones.

Table 2.3.: Events: local system

Event	System response	Source	Type
Power on	Initialize sensors and go to Normal mode	System maintainer	Asynchronous
User detected	Turn on camera feed and go to Interaction mode	User	Asynchronous
Command received	Parse it and respond	Remote Server	Asynchronous
Database update	Request update of internal databases to Remote Server	Database manager	Asynchronous
Enable fragrance diffuser	Enable fragrance diffusion for a predefined period of time	Local System	Synchronous
Video ended	Playback the next video on the queue	Local System	Synchronous
Check WiFi connection	Periodically check WiFi connection	Local System	Synchronous
	UI events		
Option selected	Track the option selected and inform the UI engine	User	Asynchronous
Image filter pressed	Go to Image filter view	User	Asynchronous
Filter selected	Detect User's face and apply filter	User	Asynchronous
Pic pressed	Go to Picture mode	User	Asynchronous
Pic setup elapsed	Take picture	Local System	Synchronous
GIF pressed	Go to GIF mode	User	Asynchronous
GIF setup elapsed	Go to GIF operation	Local System	Synchronous
GIF operation elapsed	Finish GIF	Local System	Synchronous
Share mode pressed	Go to Sharing mode (selection)	User	Asynchronous
Keyboard pressed	Give feedback to user	User	Asynchronous
Share post pressed	Upload post to designated social media	User	Asynchronous
Share post status	Inform user about shared post status	Cloud	Asynchronous

Use cases

Fig. 2.32 depicts the use cases diagram for the **Local System**, describing how the system should respond under various conditions to a request from one of the stakeholders to deliver a specific goal.

The **Admin** interacts with the **Remote Client** (through its UI) requesting the **Remote Server** to process commands, getting the state of the device, adding a video or selecting the fragrance. Additionally, the **Admin** may test the operation of the device: play video, test audio, nebulize fragrance or test the camera. This last one tests the main functionalities the **User** also utilizes, namely: select image filter, apply/reject image filter, take picture, create GIF or share multimedia on the social media.

A precondition for the interaction of the **Admin** with the **Local System** is the establishment of a remote connection between the **Remote Server** and the **Local system**, verifying its credentials. However, there is another important use case for this remote connection: the update of the **Local System**'s internal databases from the **Remote server** which will sent appropriate commands for this purpose.

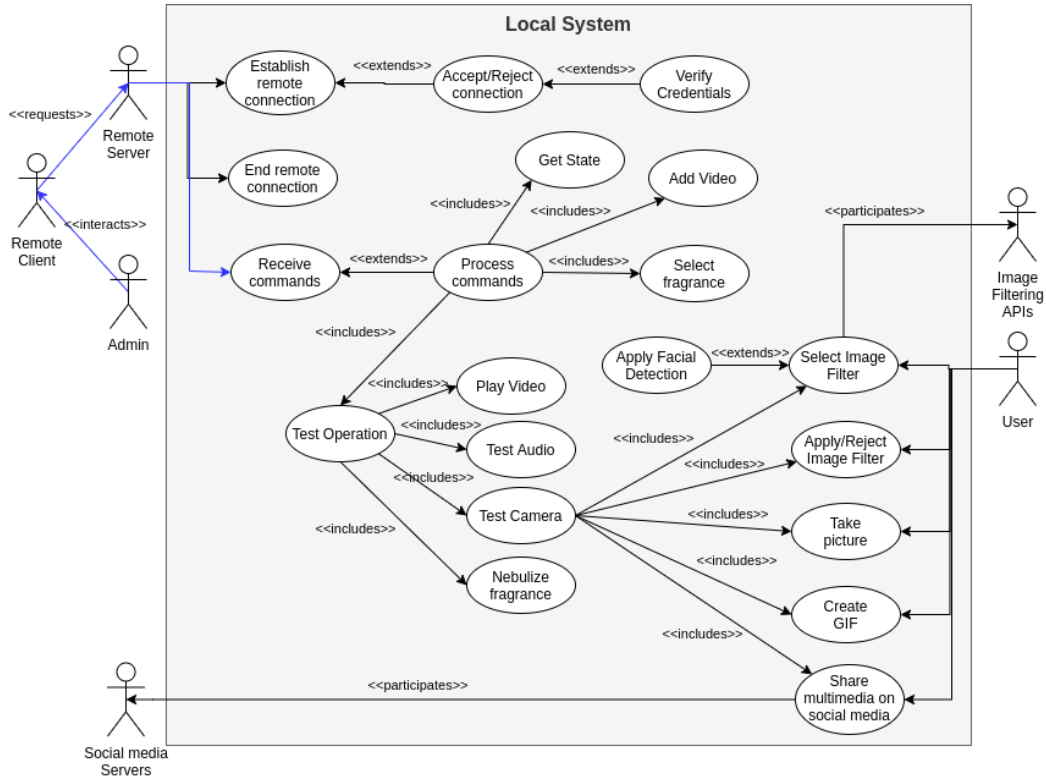


Figure 2.32.: Use cases diagram: local system

Dynamic operation

Fig. 2.33 depicts the state machine diagram for the **Local System**, illustrating its dynamic behavior.

There are two main states:

- Initialization: the device is initialized. The settings and DBs are loaded and if invalid they are restored. The WiFi communication is setup, signaling the communication status and if valid, an IP address is returned. Lastly, the HW is initialized, checking its presence, configuring it and testing the configuration: if any error occurs the device goes into the **Critical Error** state, dumping the error to a log file and waiting for reset; otherwise, the initialization is complete.
- Execution: after the initialization is successful, the system goes into the **Execution** macro composite state with several concurrent activities, modeled as composite states too. However, it should be noted

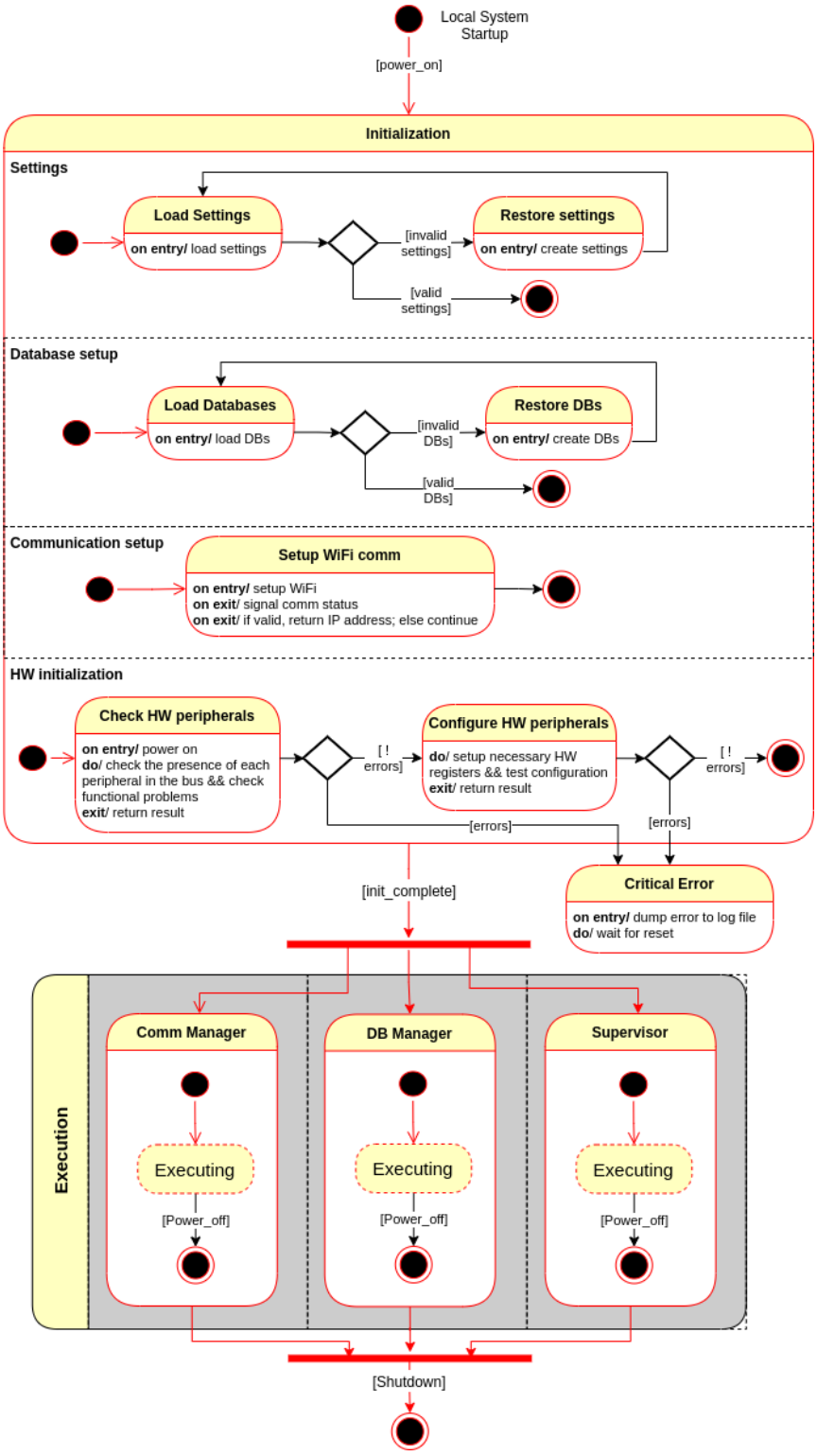


Figure 2.33.: State machine diagram: local system

that there is only one actual state for the device, although at the perceivable time scale they appear to happen simultaneously. These activities are communication management (**Comm Manager**), DB management (**DB manager**), and application supervision (**Supervisor**), and are executed forever until system's power off. They are detailed next.

Communication Manager

Fig. 2.34 depicts the state machine diagram for the **Comm Manager** component. Upon successful initialization the **Comm Manager** goes to **Idle**, listening for incoming connections. When a remote node tries to connects, it makes a connection request which can be accepted or denied. If the connection is accepted and the node authenticates successfully the **Comm Manager** is ready for bidirectional communication. When a message is received from the remote node, it is written to **TX msg queue** and the **Supervisor** is notified. When a message must be sent to the remote, it is read from the **TX msg queue** and sent to the recipient. If the connection goes down, it is restarted, going into **Idle** state again.

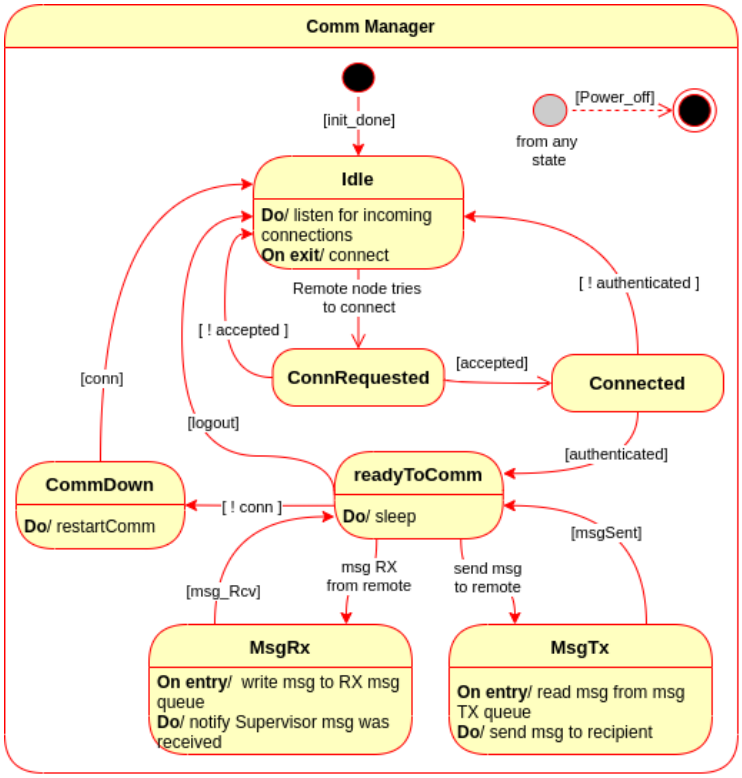


Figure 2.34.: State machine diagram: local system — Comm Manager

Database Manager

Fig. 2.35 depicts the state machine diagram for the **DB Manager** component. Upon successful initialization the **DB Manager** goes to **Idle**, waiting for incoming DB requests. When a request arrives, it is parsed, checking its validity. If the request is a DB query, a transaction is read from the respective DB to the **RX transaction queue** and the **Supervisor** is notified that there is a transaction to read. Otherwise, if the request is a DB update the transaction is written from the **TX transaction queue** to the DB and the **Supervisor** is notified that the DB was updated.

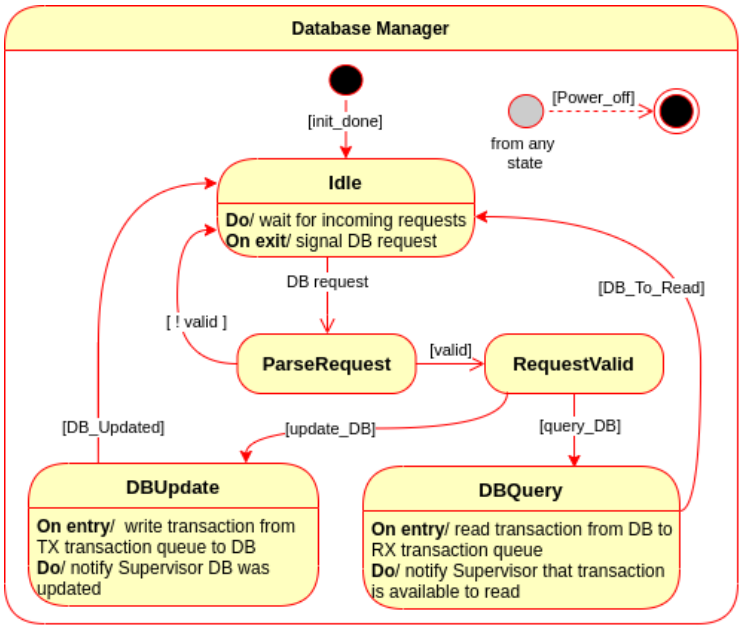


Figure 2.35.: State machine diagram: local system — DB Manager

Supervisor

Fig. 2.36 depicts the state machine diagram for the **Supervisor** component, comprising two tasks running in 'parallel' (Fig. 2.36a):

- Request Handler (Fig. 2.36b): handles incoming requests from the **Remote server**. When a request arrives, it is parsed, and, if valid, the appropriate callback is triggered, processing the request and returning its output.
- Mode manager (Fig. 2.36c): Upon successful initialization the **Mode Manager** goes to **Idle**, and it is 'awake' if it is time to play the advertisements or if a user is detected. If the former is verified— **Normal mode** — the device retrieves video and fragrance data from the DB and plays video and nebulizes fragrance. If the latter is verified — **Interaction mode** the device turns on the camera and mirrors the feed on the display, waiting for a recognizable gesture.

If the **User** choose to select an image filter, take a picture or create a GIF, the device goes into **Multimedia mode**, returning back to **Interaction mode** after its exit condition or after a timeout. Lastly, if the **User** chooses to share the image or GIF created, it must select the social media network, edit the post to enter some message and confirm the sharing, returning to **Interaction mode**. If no user interaction happens for a while, the device returns back to **Idle** mode.

Flow of events

The flow of events throughout the system is described using a sequence diagram, comprising the interactions between the most relevant system's entities. It is usually pictured as the visual representation of an use case. The main sequence diagrams are illustrated next.

Normal mode

Fig. 2.37 depicts the **Normal mode**'s sequence diagram. The blue area delimits the **MDO-L** system, comprising the **Local System Back-End**.

When its time to play the advertisements, the **Normal** mode is activated, retrieving video, audio and fragrance from the internal DB. Then two parallel activities are executed:

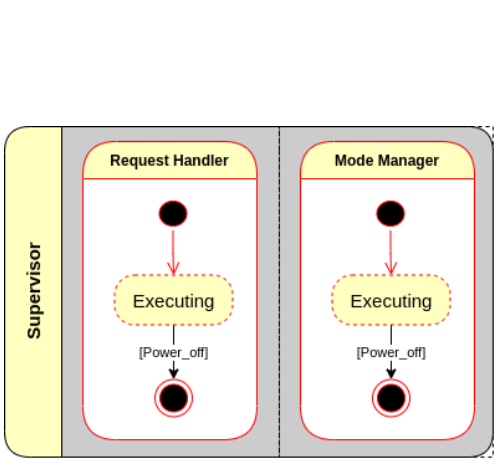
- Video playback: while its time to play the advertisements, a video is played from the video list. When it finishes, it moves the next video in the playback queue.
- Fragrance diffusion: while there are timestamps for fragrance diffusion, diffuse fragrance between start and stop times and sleep on the other occasions.

Interaction mode

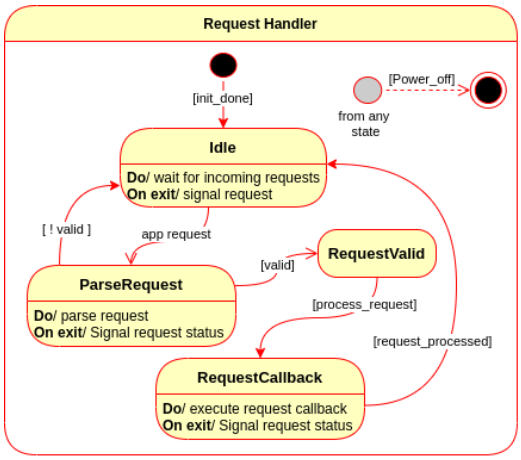
Fig. 2.38 depicts the **Interaction mode**'s sequence diagram. The blue area delimits the **MDO-L** system, comprising the **Gesture Recognition Engine**, the **UI engine** and the **Local System Back-End**.

When the **User** is in range (asynchronous event), the camera is activated, and two parallel activities are executed:

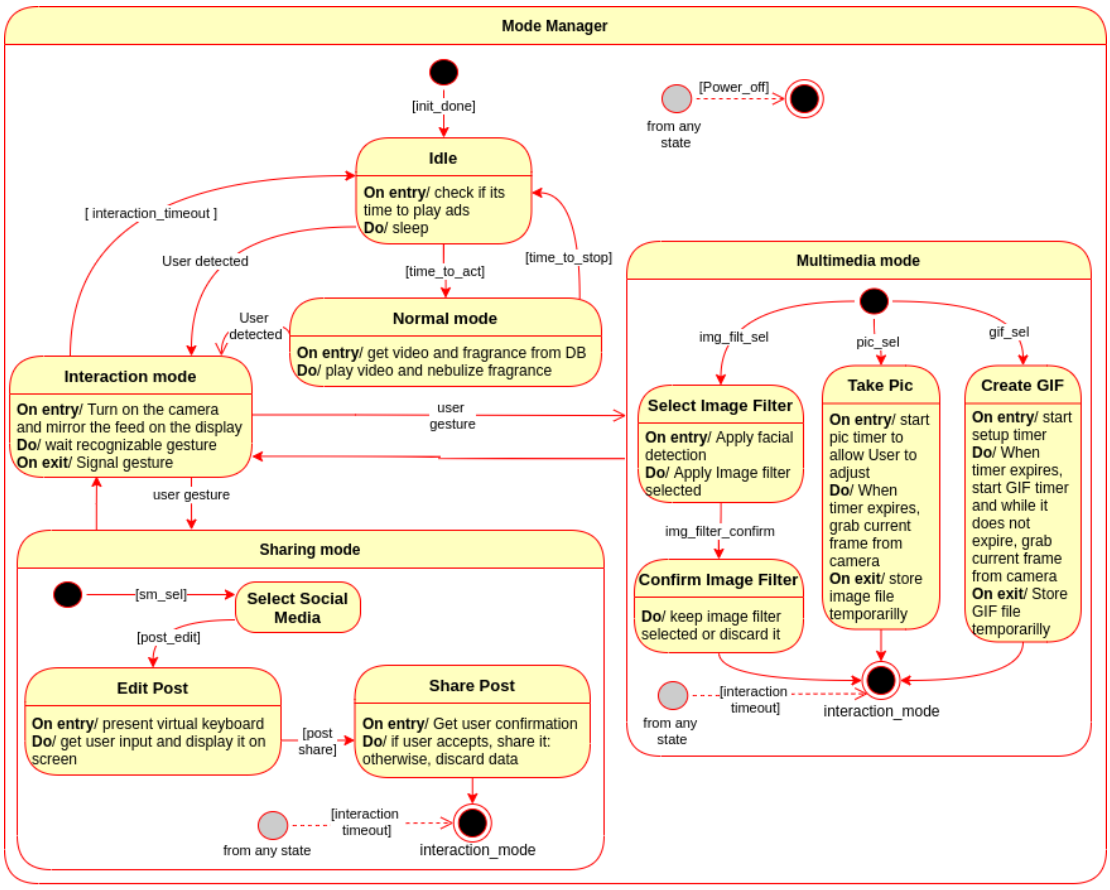
- mirror camera feed: while the **User** is in range and active, the **UI engine** will grab frame from the camera and display it on the window providing visual feedback to the **User**.
- gesture recognition and processing: if a gesture is recognized by the **Gesture Recognition Engine** it is dispatched to the **Local System Back-End** which will process it according to the following cases: **Select Image filter**, **Take Pic**, and **Create GIF**, showing the respective view in the UI and triggering the associate sequence diagram (indicated by the **ref** keyword).



(a) main



(b) Request Handler



(c) Mode Manager

Figure 2.36.: State machine diagram: local system — Supervisor

Multimedia mode

Fig. 2.39 through Fig. 2.41 depicts the **Multimedia mode**'s sequence diagrams, namely:

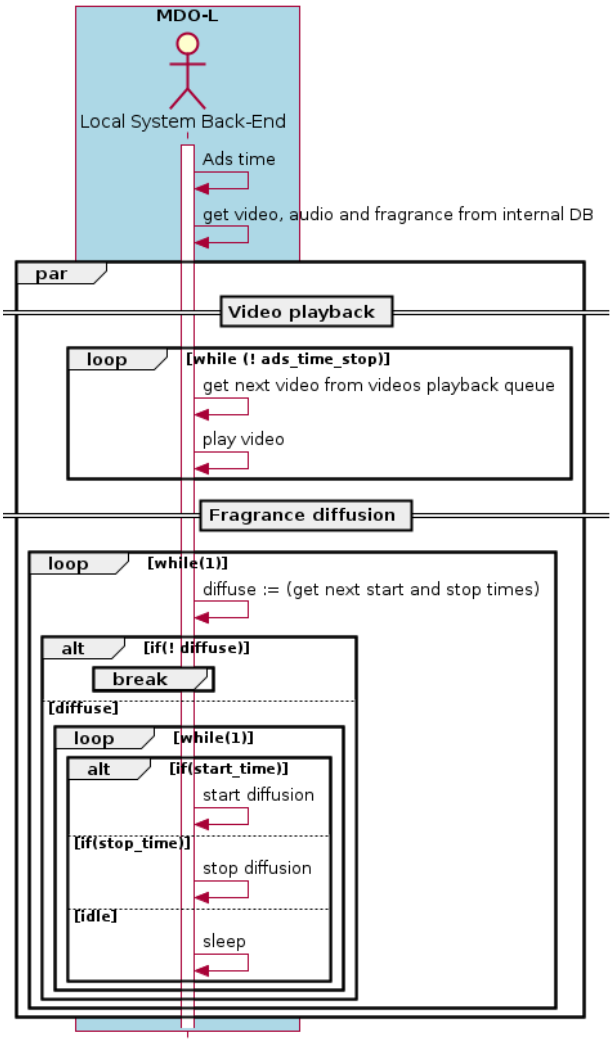


Figure 2.37.: Sequence diagram: local system — Normal mode

- Select image filter (Fig. 2.39): after the **Image filter view** is presented to the **User**, he/she can make a gesture to select the filter, which upon being recognized by the **Gesture Recognition Engine** it is dispatched by the **UI Engine** to the **Local System back-end**. Facial detection is then applied, and while the filter is active, a request is made to **Image Filtering APIs** to apply the designated filter, showing it to the **User** — **Apply filter** reference.
  If the **User** accepts the filter, it returns to **Interaction mode** with the filter simultaneously on (**Apply filter**). Otherwise, it the **User** cancels it, it simply returns to **Interaction mode**.
- Take picture: after **Picture mode** is initiated, the **Local System back-end** starts a timer to allow the **User** to get in position, and while the timer is running, the time remaining is presented to the **User**. When the timer elapses the picture is stored internally.

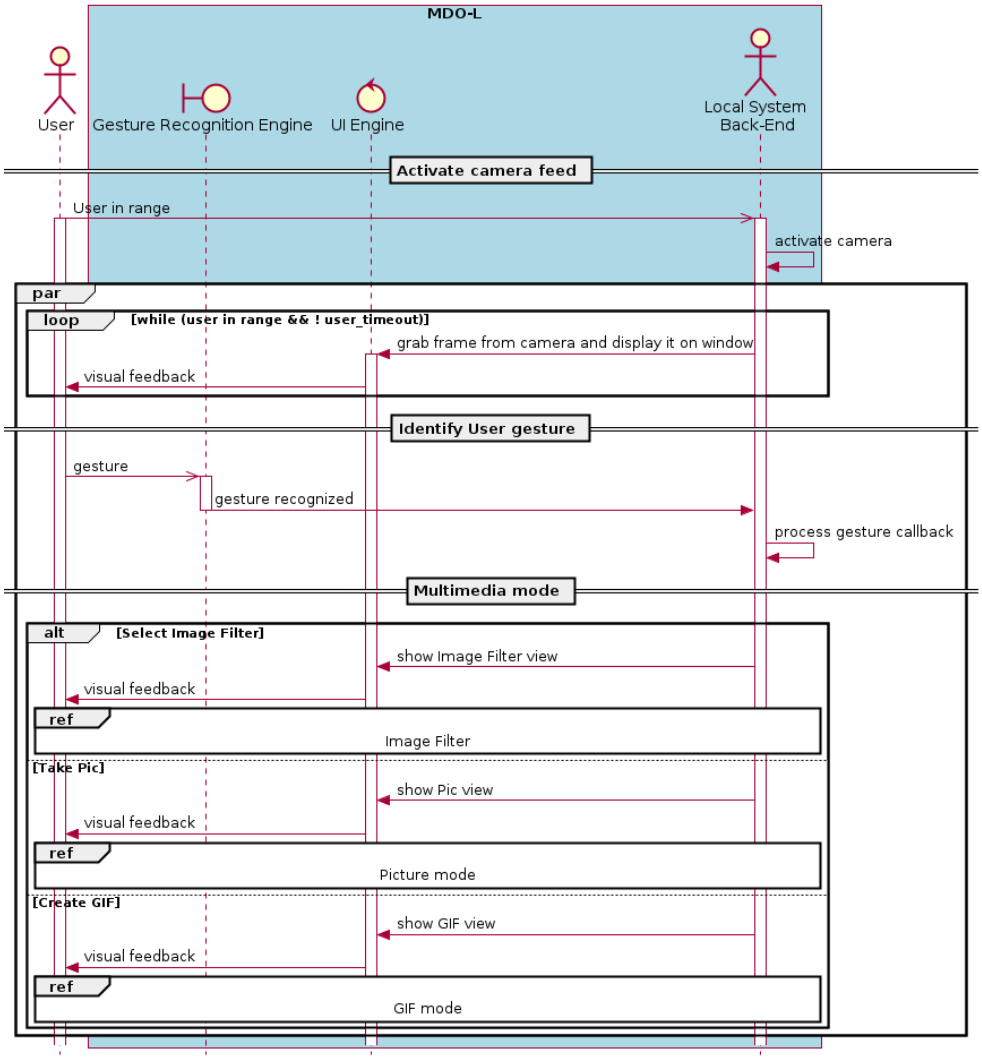


Figure 2.38.: Sequence diagram: local system — Interaction mode

- Create GIF: after **GIF mode** is initiated, the **Local System back-end** starts a timer to allow the **User** to get in position, and while the timer (**gif_setup_timer**) is running, the time remaining is presented to the **User**. When the timer elapses the GIF creation can start, with another timer (**gif_oper_timer**) being started, and while the timer is running the remaining time is shown to **User** but in a dial form. When this timer elapses the GIF is stored internally.

Sharing mode

Fig. 2.42 depicts the **Sharing mode**'s sequence diagram.

The **User** starts by selecting the social media platform (with a gesture), which upon being recognized is dispatched to the **Local System back-end** identifying the social media selected. Then, the social me-

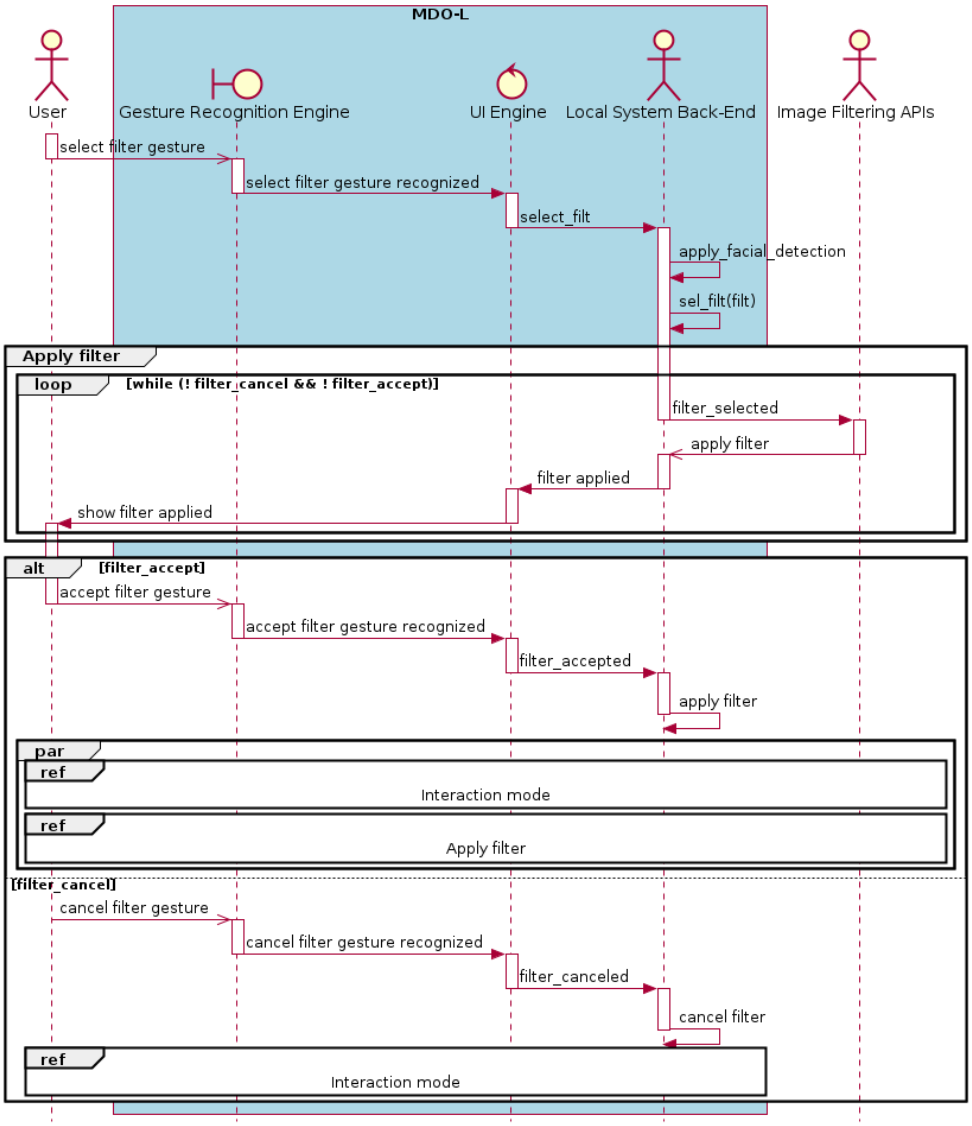


Figure 2.39.: Sequence diagram: local system — Multimedia mode (select image filter)

dia parameters are configured and the **attachment** is set to the last multimedia file. After social media configuration, the **Post Edit** view is shown to the **User**.

In the **Post editing** mode, while the **User** does not decide to share or cancel the post, the selected character from the virtual keyboard is visually feed back to the **User**.

When the **User** decides to share or cancel the post, it will trigger **share_post** or **cancel_post** callbacks, with the latter going into **Interaction mode**.

Upon triggering the **share_post** callback, **Local System back-end** tries to perform the login in the required social media platform, requesting it to one of its servers. If the login succeeds, the post is sent to **Social**

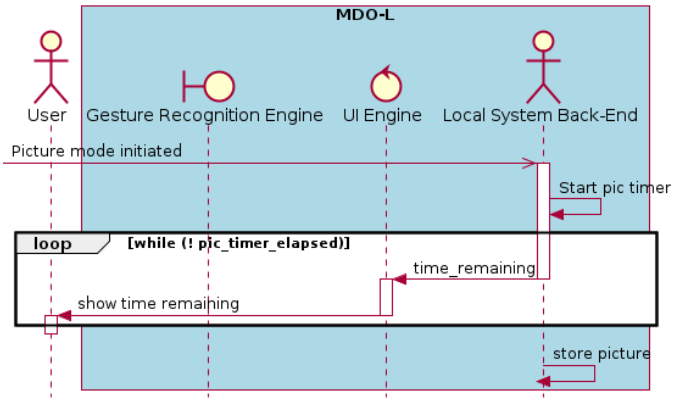


Figure 2.40.: Sequence diagram: local system — Multimedia mode (take picture)

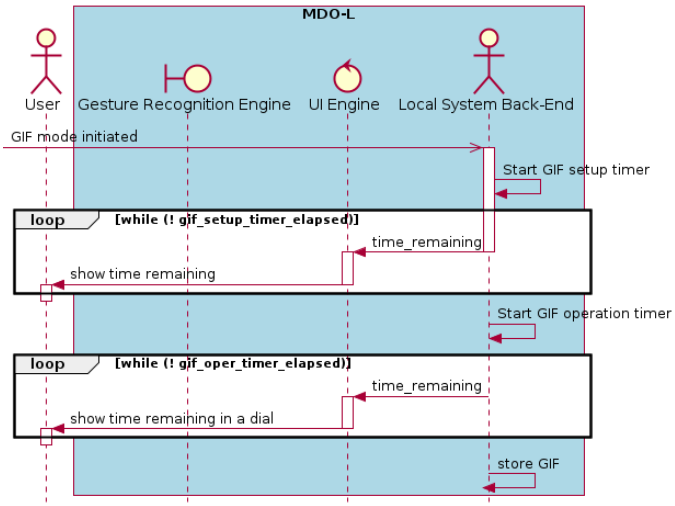


Figure 2.41.: Sequence diagram: local system — Multimedia mode (create GIF)

Media servers, which will return its status. Upon completion a dialog box will be presented to the **User**, informing it of share post status: success or failure (in case the login or the post transmission fails).

2.5. Budget estimation

Table 2.4 presents the budget estimation for the project.

Two classes of prototypes were considered, scale-model and real-scale, due to project feasibility concerns: the real-scale prototype envisions a large display (55 inch) whose cost is prohibitive; thus, a trade-off between functionality and cost needed to be performed.

Additionally, several types of costs were assessed, namely:

Table 2.4.: Budget estimation

	Scale-model Prototype		Real-scale Prototype	
	Item	**Cost (€) ***	**Item**	**Cost (€) ***
HW	Raspberry Pi 4B	50	Raspberry Pi 4B	50
	User Detection sensor	3	User Detection sensor mesh (4)	12
	LCD display 10" (non-touch)	55	LCD display 55" Full HD (non-touch)	1500
	Fragrance diffusion actuator	5	Fragrance diffusion actuator mesh (4)	20
	Camera 8 MP	32	Camera 8 MP	32
	Speakers	5	Speakers	30
	Power supply	10	Power supply	30
	PCB	8	PCB	16
Mechanical Structure	3D printed + screws	20	Built-in with display + HW packaging	100
			Full HW packaging	350
	Physical Prototype cost	**188**	**Physical Prototype cost ****	**2040**
SW development	Remote Client: 500 h ***	5000	Remote Client: 500 h ***	5000
	Remote Server: 300 h ***	3000	Remote Server: 300 h ***	3000
	Local system: 1000 h ***	10000	Local system: 1000 h ***	10000
	SW development cost	**18000**	**SW development cost**	**18000**
Operational costs	Local System power consumption ****	26,28	Local System power consumption ****	197,1
	Server operation cost *****	420	Server operation cost *****	420
	Yearly Operational cost	**446.28**	**Yearly Operational cost**	**617.1**
	Total cost	**18634.28**	**Total cost**	**20657.1**

* tax included
** considering the most expensive option
*** 10 €/h
**** 24h/7d for 1 year
***** yearly cost

- physical prototype cost — comprises HW and mechanical structure of the device: all estimation costs were made as a mean value between all trustworthy suppliers. The physical prototype cost for the scale-model prototype is about 188 EUR, while for the real-scale prototype is about 2,040 EUR, with the main difference being due to the 55 inch display cost (1,500 EUR) and the full HW packaging (350 EUR).
- SW development cost: the development cost for all software components, namely **Remote Client**, **Remote Server**, and **Local System**, yielding 18 000 hours of development, which represents about 3 months of work for a two people team. This cost is the same for both prototypes as the scale factor is only associated to HW.
- operational costs — comprises power consumption and server operation costs on a yearly basis. The server operation is the same for both prototypes, but the power consumption is more than 7 times more.
- total cost: sum of physical prototype, SW development and operational costs. The total cost for the

scale-model prototype is about 18,635 EUR, while for the real-scale one it is about 20,657 EUR, with the main difference being due to HW and power consumption costs.

Retail price and break-even analysis were not assessed at this point, as several business models may be used for that purpose.

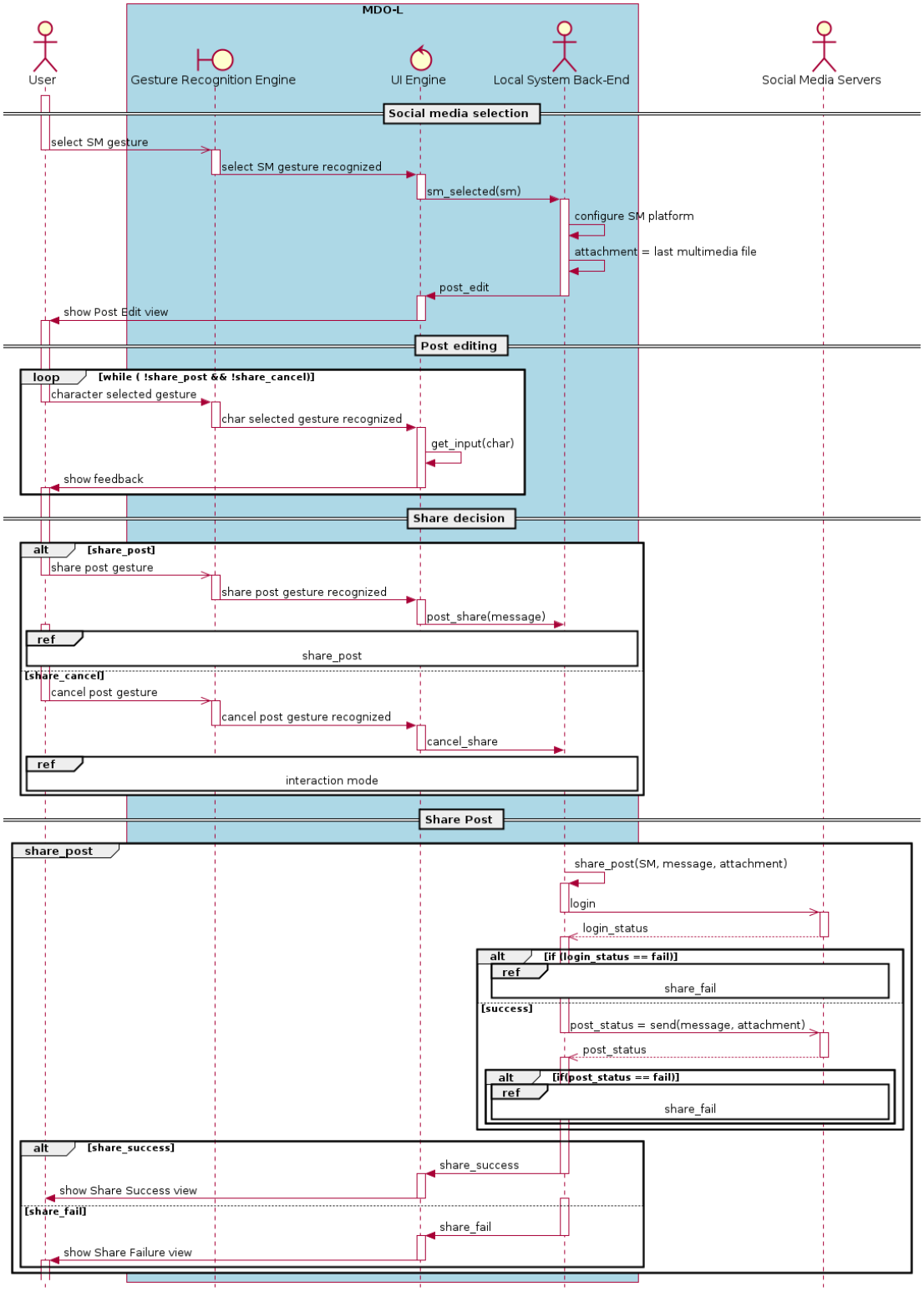


Figure 2.42.: Sequence diagram: local system — Sharing mode

3. Theoretical foundations

In this chapter the theoretical foundations are outlined, providing the basic technical knowledge to undertake the project.

3.1. Project methodology

In this section the project methodologies tools are outlined, easing the development process.

3.1.1. Waterfall model

For the domain-specific design of software the waterfall methodology is used. The waterfall model (fig. 3.1) represents the first effort to conveniently tackle the increasing complexity in the software development process, being credited to Royce, in 1970, the first formal description of the model, even though he did not coin the term [9]. It envisions the optimal method as a linear sequence of phases, starting from requirement elicitation to system testing and product shipment [10] with the process flowing from the top to the bottom, like a cascading waterfall.

In general, the phase sequence is as follows: analysis, design, implementation, verification and maintenance.

1. Firstly, the project requirements are elicited, identifying the key requirements and constraints the system being developed must meet from the end-user perspective, captured in natural language in a product requirements document.

2. In the analysis phase, the developer should convert the application level knowledge, enlisted as requirements, to the solution domain knowledge resulting in analysis models, schema and business rules.

3. In the design phase, a thorough specification is written allowing the transition to the implementation phase, yielding the decomposition in subsystems and the software architecture of the system.

4. In the implementation stage, the system is developed, following the specification, resulting in the source code.

5. Next, after system assembly and integration, a verification phase occurs and system tests are performed, with the systematic discovery and debugging of defects.

6. Lastly, the system becomes a product and, after deployment, the maintenance phase start, during the product life time.

While this cycle occurs, several transitions between multiple phases might happen, since an incomplete specification or new knowledge about the system, might result in the need to rethink the document.

The advantages of the waterfall model are: it is simple and easy to understand and use and the phases do not overlap; they are completed sequentially. However, it presents some drawbacks namely: difficulty to tackle change and high complexity and the high amounts of risk and uncertainty. However, in the present work, due to its simplicity, the waterfall model proves its usefulness and will be used along the project.

As a reference in the sequence of phases and the expected outcomes from each one, it will be used the chain of development activities and their products depicted in fig. 3.2 (withdrawn from [11]).

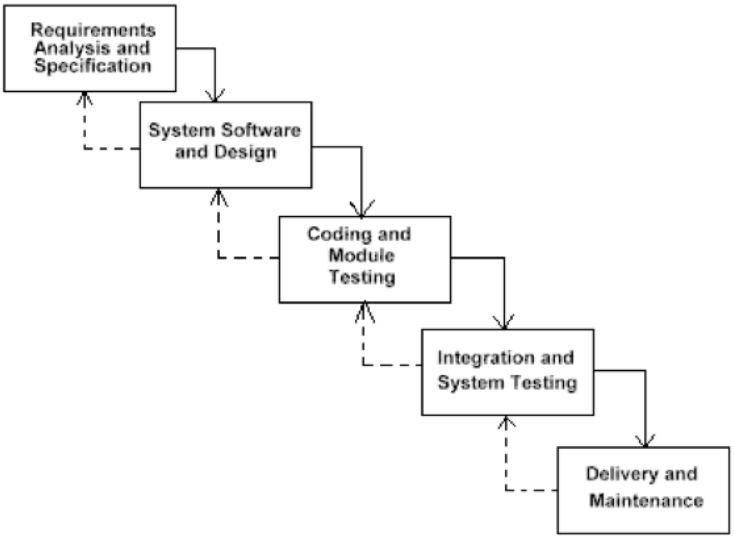


Figure 3.1.: Waterfall model diagram

3.1.2. Unified Modeling Language (UML)

To aid the software development process, a notation is required, to articulate complex ideas succinctly and precisely. The notation chosen was the Unified Modeling Language (UML), as it provides a spectrum of

notations for representing different aspects of a system and has been accepted as a standard notation in the software industry [11].

The goal of UML is to provide a standard notation that can be used by all object- oriented methods and to select and integrate the best elements of precursor software notations, namely Object-Modeling Technique (OMT), Booch, and Object Oriented Software Engineering (OOSE) [11]. It provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design, deployment). System development focuses on three different models of the system (fig. 3.2) [11]:

1. **The functional model**: represented in UML with use case diagrams, describes the functionality of the system from the user's point of view.

2. **The object model**: represented in UML with class diagrams, describes the structure of the system in terms of objects, attributes, associations, and operations.

3. **The dynamic model**: represented in UML with interaction diagrams, state-machine diagrams, and activity diagrams, describes the internal behaviour of the system.

3.2. Concurrency

Concurrency is used to refer to things that appear to happen at the same time, but which may occur serially [12], like the case of a multithreaded execution in a single processor system. Two concurrent tasks may start, execute and finish in overlapping instants of time, without the two being executed at the same time. As defined by the Portable Operating System Interface (POSIX) specification, a concurrent execution requires that a function that suspends the calling thread shall not suspend other threads, indefinitely.

This concept is different from parallelism. Parallelism refers to the simultaneous execution of tasks, like the one of a multithreaded program in a multiprocessor system. Two parallel tasks are executed at the same time and, as such, they require the execution in exclusivity in independent processors.

Every concurrent system provides three important facilities [12]:

- **Execution Context**: refers to the concurrent entity state. It allows the context switch and it must maintain the entities states, independently.
- **Scheduling**: in a concurrent system, the scheduling decides what context should execute at any given time.
- **Synchronization**: this allows the management of shared resources between the concurrent execution contexts.

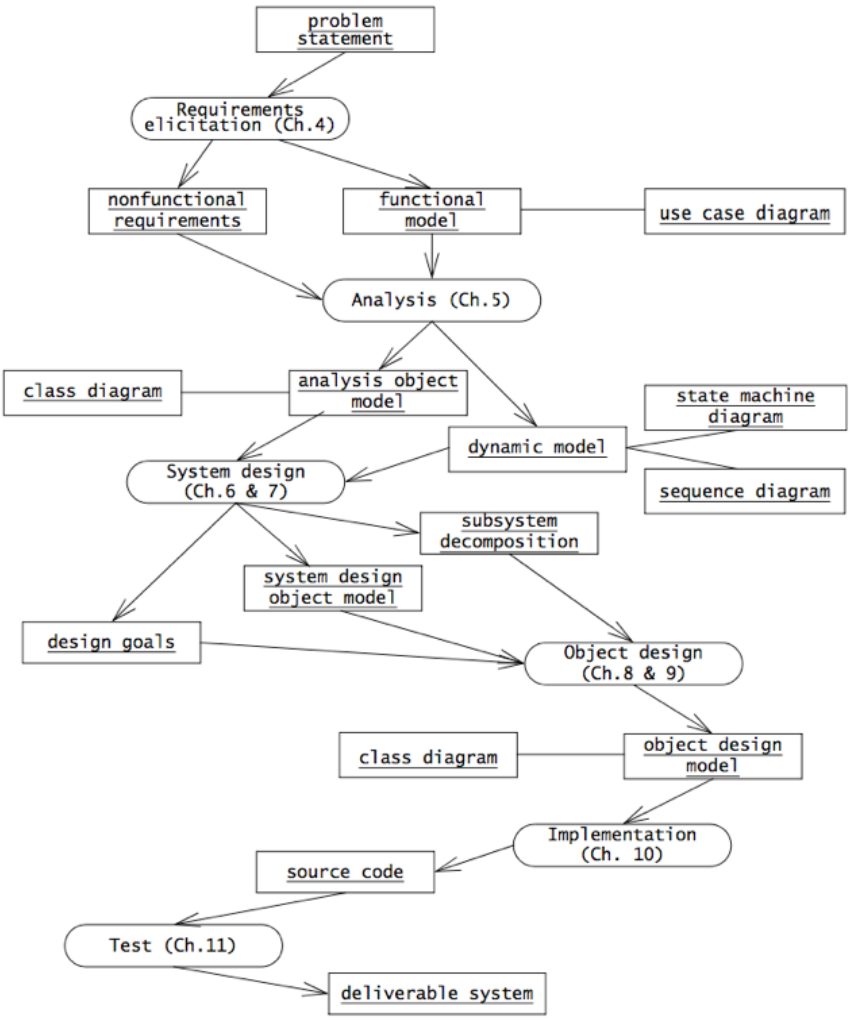


Figure 3.2.: An overview of the object-oriented software engineering development and their products. This diagram depicts only logical dependencies among work products (withdrawn from [11])

3.3. Threads versus Processes

Threads and processes are two mechanisms to design an application to perform multiple tasks concurrently. A single process can contain multiple threads. In this section, it is briefly presented some of the factors that might influence the choice of whether to implement an application as a group of threads or as a group of processes.

The advantages of a multithreaded approach are [13]:

- Data sharing: Sharing data between threads is easy, as all of them share the same data and heap address spaces. By contrast, sharing data between processes requires the usage of an Inter-Process Communication (IPC) mechanism.

- Context switching: Thread creation is faster than process creation; context-switch time may be lower for threads than for processes.

Using threads can have some disadvantages compared to using processes [13]:

- Thread safety: When programming with threads, one needs to ensure that the functions we call are thread-safe, i.e., can be invoked by multiple threads at the same time. Multiprocess applications don't need to be concerned with this.
- Isolation: A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. By contrast, processes are more isolated from one another.
- Memory usage: Each thread is competing for use of the finite virtual address space of the host process. In particular, each thread's stack and thread-specific data (or thread-local storage) consumes a part of the process virtual address space, which is consequently unavailable for other threads. Although the available virtual address space is large, this factor may be a significant limitation for processes employing large numbers of threads or threads that require large amounts of memory. By contrast, separate processes can each employ the full range of available virtual memory (subject to the limitations of RAM and swap space).

Summarizing, the key factors to consider when designing a concurrent application (multithread or multiprocess) are [13]:

- In a multithreaded process, multiple threads are concurrently executing the same program. All of the threads share the same global and heap variables, but each thread has a private stack for local variables. The threads in a process also share a number of other attributes, including process ID, open file descriptors, signal dispositions, current working directory, and resource limits.
- The key difference between threads and processes is the easier sharing of information that threads provide, and this is the main reason that some application designs map better onto a multithread design than onto a multiprocess design.
- Threads can also provide better performance for some operations (e.g., thread creation is faster than process creation), but this factor is usually secondary in influencing the choice of threads versus processes.

3.3.1. Pthreads API

In the late 1980s and early 1990s, several different threading APIs existed. Thus, a standard and portable implementation was required, leading to standardization of the POSIX threads API — Pthreads — by the POSIX.1c in 1995 [13].

Pthreads is a standardized model for dividing a program into subtasks whose execution can be interleaved or run in parallel [12].

Thread creation

When a program is started, the resulting process consists of a single thread, called the initial or main thread. Additional threads can be created using the function **pthread_create()** [13]:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start)(void*), void *arg);
```

This function takes as arguments a pointer to a buffer of type **pthread_t** — into which the unique identifier for this thread is copied (thread ID) before **pthread_create()** returns, the thread attributes, a function pointer containing the so called **worker** function, and the **worker** function arguments.

The new thread then starts execution by calling the function identified by **start** with the argument **arg** (i.e, **start(args)**). The thread that calls **pthread_create()** continues execution with the next statement that follows the call. The **arg** argument is declared as **void *****,** allowing generic data to be passed to the **worker** function. This function returns **0** on success, or a positive number indicating the error occurred.

Thread termination

The execution of a thread terminates in one of the following ways [13]:

- the thread's worker function performs a **return** specifying a return value for the thread;
- the thread calls **pthread_exit()**;
- the thread is canceled using **pthread_cancel()**;
- any of the thread calls **exit()**, or the main thread performs a return, causing all threads in the process to terminate immediately.

The **pthread_exit()** function terminates the calling thread and returns a value via **retval** that is available to another thread in the same process that calls **pthread_join()**.

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Joining with a terminated thread

The **pthread_join()** function waits for the thread identified by **thread** to terminate [13].

```
#include <pthread.h>
int pthread_join(pthread_t thread , void ** retval);
```

It is important to note that:

- if the thread has already terminated, **pthread_join()** returns immediately;
- calling **pthread_join()** for ta thread ID that has been previously joined can lead to unpredictable behavior;
- if a thread is not detached, it must be joined with **pthread_join()**, otherwise it produces a 'zombie' thread, wasting system resources.

Detaching a thread

By default, a thread is joinable, meaning that when it terminates, another thread can obtain its return status using **pthread_join()**. Sometimes, the thread's return status is irrelevant: one simply wants the system to automatically clean up and remove the thread when it terminates. In this case, the thread can be detached, by making a call to **pthread_detach()** specifying the thread's identifier in thread [13].

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

Once a thread has been detached, it is no longer possible to use **pthread_join()** to obtain its return status, and the thread can't be made joinable again. Another important note is that **pthread_detach()** only controls what happens after a thread terminates, not how or when it terminates. If another thread calls **exit()** or the main thread returns, all threads in the process are immediately terminated, regardless of whether they are joinable or detached.

3.4. Communications

The communications technologies and the associated tools used for the project development are briefly described next.

3.4.1. IEEE 802.11 — Wi-Fi

IEEE 802.11, commonly known as Wi-Fi, is part of the IEEE 802 set of Local Area Network (LAN) protocols, and specifies the set of Media Access Control (MAC) and physical layer protocols for implementing Wireless local Area Network (WLAN) communication in a wide sprectrum of frequencies, ranging from 2.4–60 GHz.

TCP/IP

The most commonly used protocols for Internet communications, including Wi-Fi, are Transmission Control Protocol (TCP) and IP, usually associated together, being part of the OSI model (Fig. 3.3), which characterises and standardises the communication functions of a telecommunication or computing system, being agnostic to their underlying internal structure and technology.

A computer protocol is a standardised procedure for the exchange and transmission of data between devices, as requested for the application processes. The TCP provides services at the Transport layer, handling the reliable, unduplicated and sequenced delivery of data [14], while the UDP provides data transportation without guaranteed data delivery or acknowledgments. The TCP can be thought of a reliable version of User Datagram Protocol (UDP), generalizing. The IP part of the TCP/IP suite, providing services at the Network layer, is used to make origin and destination addresses available to route data across networks.

These protocols are applied in sequence to the user's data to create a frame that can be transmitted from the sending application to the receiving application. The receiver reverses the procedure to obtain the original user's data and pass them to the receiving application [14].

Another interesting fact, due to the technology agnostic aspect of the OSI Model, is that IP and the higher-level protocols may be implemented on several kinds of physical nets.

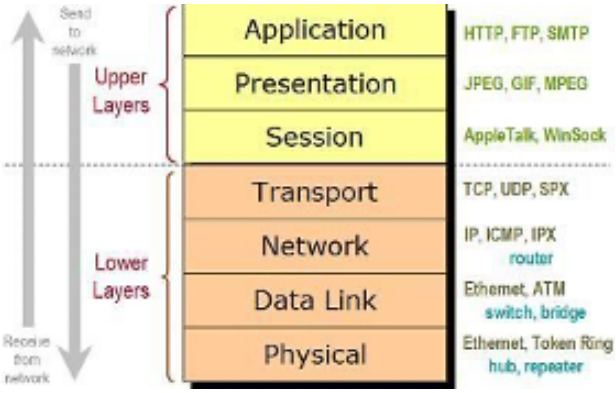


Figure 3.3.: OSI model

3.4.2. Network programming — sockets

Computer systems implement multiple processes which require an identifier. As such, the IP address is not enough to uniquely identify the origin/destination of data to be transmitted, and the port number is added. This combination of an IP address and port number is sometimes called a network socket [15], allowing data to be delivered to multiple processes in the same machine — same IP address. It is the socket pair (the

4-tuple consisting of the client IP address, client port number, server IP address, and server port number) that specifies the two end points that uniquely identifies each TCP connection in an internet [15].

In a broader sense, a socket can be described as a method of IPC that allows data to be exchanged between applications, either on the same host (computer) or on different hosts connected by a network [13], as a local interface to a system, created by the applications and controlled by the operating system, allowing an application process to simultaneously send and receive messages from other processes.

The Socket API was created in UNIX BSD 4.1 in 1981, with widespread implementation in UNIX BSD 4.2 [13]. It implements the Client-Server paradigm and implement several (standard) functions to access the operating system network resources, through system calls, in Linux [13].

There are two generic ways to use sockets: for outgoing connections — client socket — and for incoming connections — server socket. Fig. 3.4 illustrates the required steps to obtain a connected socket:

1. When a socket is initially created is mostly unuseful.
2. Binding the server socket associates it to an unique network tuple (address and port number), enabling it to be uniquely addressed.
3. When a socket server goes into listening mode, the remote devices can initiate the connection procedure, referring to its unique network tuple.
4. When the socket server accepts a connection, it spawns a new socket which is connected to the remote device, and the endpoints can effectively communicate. The server socket is ready to accept new incomming connections.

3.4.3. Client/server model

The client/server model is the most common form of network architecture used in data communications today [17]. A client is a system or application that request the activity of a service provider system or application, called servers, to accomplish specific tasks. The client/server concept functionally divides the execution of a unit of work between activities initiated by the end user (client) and resource responses (services) to the activity request as a cooperative environment [17]. The client, typically handling user interactions and data exchange/modification in the user's behalf, makes a request for a service, and a server, often requiring some resource management (synchronization and access to the resource), performs that service, responding to the client requests with either data or status information [18].

An example of a simple client-server model using the Socket API, through system calls, is presented in Fig. 3.5. The operation of sockets can be explained as follows [13]:

- The **socket()** system call creates a new socket, establishing the protocols under which they should communicate. For both client and server to communicate, each of them must create a socket.

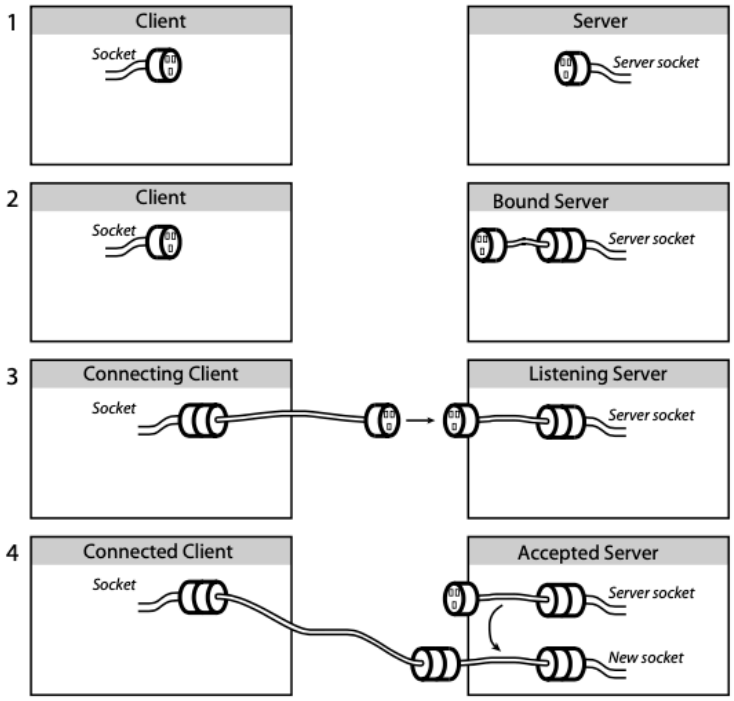


Figure 3.4.: Steps to obtain a connected socket (withdrawn from [16])

- Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place. Two sockets are connected as follows:
  1. One application, assuming the role of server, calls **bind()** to bind the socket to a well-known address, and then calls **listen()** to notify the kernel it is ready to accept incoming connections.
  2. The other application, assuming the role of client, establishes the connection by calling **connect()**, specifying the address of the socket to which the connection is to be made.
  3. The server then accepts the connection using **accept()**. If the **accept()** is performed before the client application calls **connect()**, then the **accept()** blocks.
- Once a connection has been established, data can be transmitted in both directions between the applications (analogous to a bidirectional telephone conversation) until one of them closes the connection using **close()**.
- Communication is performed using the conventional **read()** and **write()** system calls or via a number of socket-specific system calls (such as **send()** and **recv()**) that provide additional functionality. By default, these system calls block if the Input/Output (I/O) operation can't be completed immediately. However, nonblocking I/O is also possible.

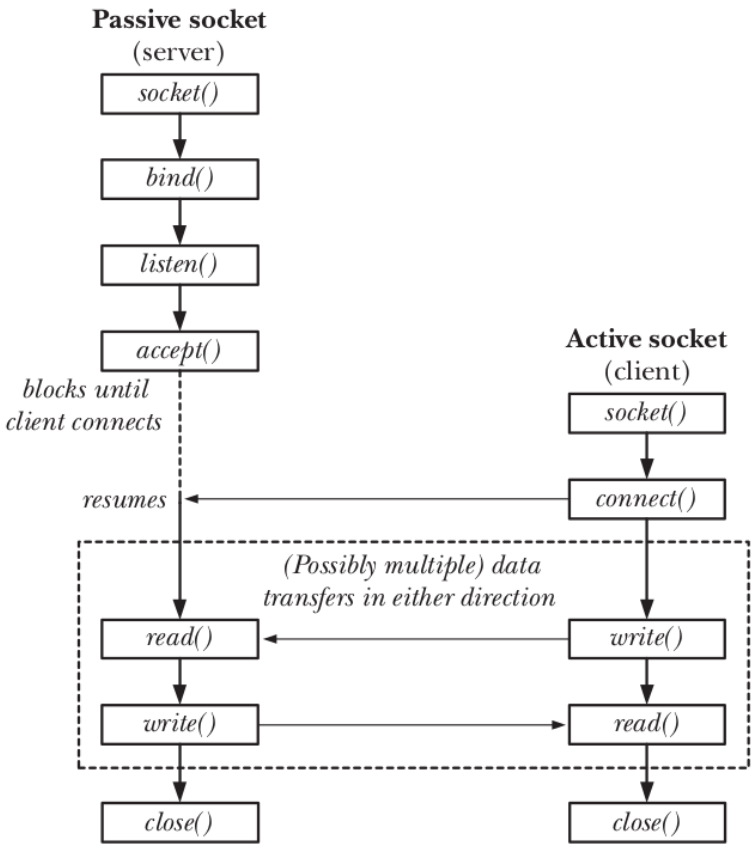


Figure 3.5.: Overview of UNIX system calls with sockets implementing a server/client paradigm (withdrawn from [13])

3.5. Daemons

In this section daemons are introduced, showing how to create them, handle possible errors and how to communicate with them.

3.5.1. What is a Daemon?

A daemon is a background process that runs without user input and usually provides some service, either for the system as a whole or for user programs. [19]

Normally, daemons are started when a system boots and, unless forcibly terminated, run until system shutdown. Because a daemon does not have a controlling terminal, any output, either to stderr or stdout, requires special handling. They often run with **superuser privilege** because they use privileges ports (1 - 1024) or because they have access to some sort of privileged resource. Generally, daemons are process

group leaders and session leaders, a daemon's parent is the **init** process, which has the PID of 1 (daemon is an orphan process inherited by init).

3.5.2. How to create a Daemon

The steps to create a daemon are:

1. **Fork** and exit in the parent process;
2. Create a new session in the child using the **setsid** call;
3. Make the root directory, "/", the child process's working directory;
4. Change the child process's umask to 0;
5. Close any unneeded file descriptor the child inherited.

Fork and exit in the parent process

A daemon is started from a shell script or the command line. Daemons are unlike application programs because they are not interactive, i.e., they run in the background and, as a result, do not have the controlling terminal.

The parent forks and exits as the first step toward getting rid of the controlling terminal (they only need a terminal interface long enough to get started).

```
pid_t pid;
// create a new process

pid = fork();
if (pid < 0) { // error trying to execute fork
  ERROR("fork failure");
  exit(EXIT_FAILURE);
}

if (pid > 0) // parent process (exit)
  exit(EXIT_SUCCESS);
// child process continues the execution
...
```

Create a new session in the child using the setsid call

Calling **setsid** accomplishes several things:

- It creates a new session if the calling process is not a process group leader, making the calling process the session leader of the new session;

- It makes the calling process the process group leader of the new process group;
- It sets the Process Group ID (PGID) and the Session ID (SID) to the Process ID (PID) of the calling process;
- It dissociates the new session from any controlling tty.

```
pid_t sid;
sid = setsid(); // create a new session
if (sid < 0) {
  ERROR("setsid failure");
  exit(EXIT_FAILURE);
}
```

- Each process is a member of a process group, which is a collection of one or more processes generally associated with each other for the purposes of job control (# cat ship-inventory.txt | grep booty | sort).
- When a new user first logs into a machine, the login process creates a new session that consists of a single process, the user's login shell. The login shell functions as the session leader.

Make the root directory, "/", the child process's working directory

This is necessary because any process whose current directory is on a mounted file system will prevent that file system from being unmounted.

Making "/" a daemon's working directory is a safe way to avoid this possibility.

```
// make '/' the root directory
if (chdir("/") < 0) {
  ERROR("chdir failure");
  exit(EXIT_FAILURE);
}

// continue executing the child process
...
```

Change the child process's umask to 0

This step is necessary to prevent the daemon's inherited umask from interfering with the creation of files and directories.

Consider the following scenario:

- A daemon inherits a umask of 055, which masks out read and execute permissions for group and other.
- Resetting the daemon's umask to 0 prevents such situation.

```
// resetting umask to 0
umask(0);

// continue executing the child process
...
```

Close any unneeded file descriptor the child inherited

This is simply a common sense step. There is no reason for a child to keep open descriptors inherited from parent. The list of potential file descriptors to close includes at least stdin, stdout and **stderr**.

```
// close unneeded file descriptors
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);

// continue executing the child process
...
```

3.5.3. How to handle errors

There's the **problem** that once a daemon calls **setsid**, it no longer has the controlling terminal an so it has nowhere to send output that would normally go to **stdout** or **stderr** (such as error messages).

The **solution** is that, fortunately, the standard utility for this purpose is the **syslog** service, provided by the system logging daemon, **syslogd**.

Handling Errors with syslog

syslogd is a daemon that allow to save log messages from other daemons or applications. The relevant interface is defined in **<syslog.h>** header file. The API is simple, **openlog** opens the log, **syslog** writes a message to it, and **closelog** close the log.

The function prototypes are listed here:

```
#include <syslog.h>

void openlog(char *ident, int option, int facility);
void closelog(void);
void syslog(int priority, chat *format, ...);
```

3.5.4. Communicating with a Daemon

To communicate with a daemon, you send a it signals that cause it to respond in a given way. For example, it is typically necessary to force a daemon to reread its configuration file. The most common way to do this is to send a **SIGHUP** signal to the daemon.

When you execute the command "kill PID" on command line, the signal **SIGINT** is sent to daemon to terminate the daemon execution.

3.6. Device drivers

A device driver is a small piece of software that tells the operating system and other software how to communicate with a piece of hardware [20].

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.

The device driver provides the following characteristics:

i. A set of functions to communicate with the hardware device, and a standardized kernel interface;
ii. It shoud be an autonomous component, able to be dynamically added to and removed from the OS;
iii. Data flux control between the user space program and device;
iv. A user defined section, so the device driver can be visible as a node in the /dev, for the OS.

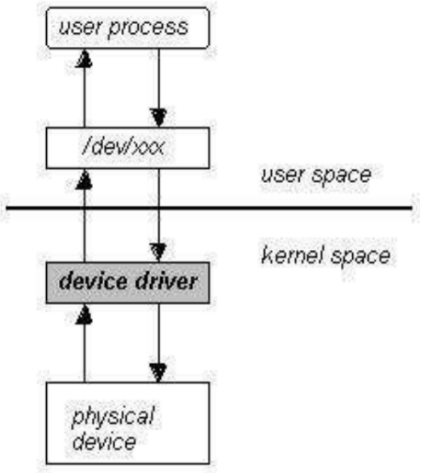


Figure 3.6.: Example of the usage of a device driver (withdrawn from [21])

Character device drivers and Block device drivers are the two of the most important device drivers. Block device drivers are accessed by the user space program by a system buffer that works like a data cache.

There's no need for management or allocation routines as the system transfers the data from/to the device. Character device drivers communicate directly with the user space program, so no buffer is required [21].

3.6.1. Kernel mode vs Application

How can an application access module services?

i. When one of the kernel functions associate to the driver is called, the function **module_init()** calls the function **register_capability()** to register the driver services on kernel - on character drivers this is done by **chrdev_region()** and **alloc_chrdev_region()**.

ii. The function **register_capability()** saves a pointer to an argument in the internal data structure **capabilities[]**.

iii. The system defines a protocol of how the application will have access to **capabilities[]** by system calls.

iv. When the driver is no more necessary, the module can be unregistered from kernel, freeing the entry on **capabilities[]** using the function **module_exit()**.

The modules run on kernel space, while the applications run on user space.

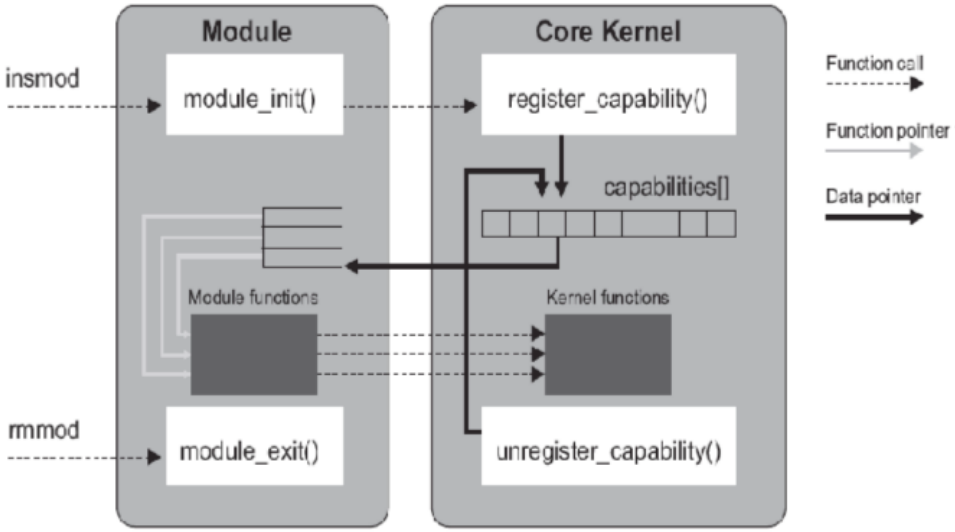


Figure 3.7.: Application accessing to module services (withdrawn from [21])

Linux kernel offers a set of functions to the user space for the interaction with hardware. Linux represents and manages a byte sequence using files. Almost all devices are represented by a sequence of bytes, so linux also represents the I/O devices as special files called nodes with entries in /**dev** directory.

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.

Normally, for each function in user space there is a matching function in the kernel space allowing data transfer between kernel and user space.

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

Figure 3.8.: Examples of drivers event and corresponding interface functions between kernel space and user space (withdrawn from [21])

To associate the normal files to the kernel mode, the linux system uses the major number and the minor number:

- **Major number** is normally used by linux system to map I/O requests to driver code. By other words, identifies the driver associated to the device.
- **Minor number** is dedicated to internal use and it is used by kernel to determine exactly which device was referenced. Depending on the way the driver was coded, this number can have a direct meaning, as a device counter, or it can be a simple driver iterator on the local devices array on kernel.

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.

3.7. Build system and Makefiles

Make is essentially a dependency-tracking build utility, which is most commonly used as a build automation tool to generate executable programs and libraries from source code by reading files called **Makefiles**. These files specify how to derive the target program, defining the prerequisites (dependencies) and the rule for generating it [22]. Nonetheless, the **Make** utility is also used as automation build tool for compiling **LaTeX** documents or to manage the workflow of data analysis projects [23].

Make was originally created by Stuart Feldman in April 1976 at Bell Labs, after the experience of a coworker in futilely debugging a program of his where the executable was accidentally not being updated with changes [24]:

> 'Make originated with a visit from Steve Johnson (author of yacc, etc.), storming into my office, cursing the Fates that had caused him to waste a morning debugging a correct program (bug had been fixed, file hadn't been compiled, cc *.o was therefore unaffected). As I had spent a part of the previous evening coping with the same disaster on a project I was working on, the idea of a tool to solve it came up. It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend. Use of tools that were still wet was part of the culture. Makefiles were text files, not magically encoded binaries, because that was the Unix ethos: printable, debuggable, understandable stuff.'
>
> — Stuart Feldman, The Art of Unix Programming, Eric S. Raymond 2003

Before **Make**'s introduction, the Unix build system most commonly consisted of OS dependent **make** and **install** shell scripts accompanying their program's source code. **Make** enabled the combination of commands for the different targets into a single file and to abstract out dependency tracking and archive handling, making it a cornerstone to pave the way for modern build environments.

Several versions of the **Make** utility were implemented, by porting them into different OSes or by rewriting from scratch, like the **BSD Make** (Berkeley Software Distribution (BSD) systems), **GNU make** (Linux and MacOS) or **nmake** (Windows). The **GNU make** is the standard implementation for Linux– and MacOS–based systems [25], and, thus, it will the one being addressed here.

The GNU implementation of **make** — written by Richard Stallman and Roland McGrath [22]— tracks down the targets that need to be recompiled and executes the defined commands in the **makefile** containing the rules for those targets. If the flag **-f** is not provided to **make**, it will look for makefiles **GNUmakefile** (not recommended — only if the file is specific to **GNU make**), **makefile**, and **Makefile** in that order [22]. make will then update the target if it depends on prerequisite files that have been modified since the target was last modified, or if the target does not exit.

3.7.1. Makefile syntax and example

A simple **makefile** consists of rules with the following shape [22]:

```
target ... : prerequisites ...
<TAB>recipe
<TAB>...
<TAB>...
```

The basic syntax can be explained as follows [22]:

- target: it is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as **clean**, being named as phony targets.
- prerequisite: file that is used as input to create the target. A target often depends on several files.
- recipe: it is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. A **tab** character is required at the beginning of every recipe line.

  Usually a recipe is in a rule with prerequisites and serves to create a target file if any of the prerequisites change. However, the rule that specifies a recipe for the target need not have prerequisites. For example, the rule containing the delete command associated with the target **clean** does not have prerequisites.
- rule: explains how and when to remake certain files which are the targets of the particular rule. **make** carries out the recipe on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action.

Listing 3.1 presents an **Makefile** example, with a preamble documenting the syntax for easier comprehension (lines 1 through 51). Line 54 defines the project name by assigning it to a variable **PROJ**. Line 57 and 58 define the file paths for source code and binaries using macro expansion. Line 61 defines the compilation flags — **CFLAGS** — which is known makefile variable. Line 62 defines the command **RM** to remove files and folders recursively. Lines 65 and 66 define the **Doxygen** documentation paths. Lines 69 and 71 retrieve database and C-source files by using the wildcard expansion which matches the patterns (globs) defined after. Lines 74 through 79 exemplify the usage of conditionals to add different compilation flags. Line 81 comment shows a bad practice, which is to define the **CC** variable — **C Compiler** — to a specific compiler, thus, making it not portable. The alternative is to set the **CC** outside of the **makefile**. Line 83 finalizes the initialization part of the makefile by performing pattern substitution to obtain all object files from **C**-source files.

Line 86 starts the rules part of the makefile, defining the default rule and stating that to build **$(PROJ)** executable it is required the object files (**$(OBJ)**) and one can compile it using the **C compiler**, the dependencies and the required libraries. When **make** tries to evaluate the dependency list, if the object files haven't been produced yet, it will generate them using the rule from line 95 and then it can execute rule **$(PROJ)**.

Line 101 states that calling **make all** will invoke the default rule and echo '**Compiling project**'.

Line 105 states that calling **make install** will invoke the **all** and **clean** rules to compile the application and install the binaries at the appropriate location.

Line 111 contains the phony targets, i.e., targets whose names represent actions like **clean**, **mrproper**,

clean-all, and doc: clean removes all objects files, mrproper and clean-all will remove also the executable file generated, and doc will generate the documentation using Doxygen.

```
1  ################################### Makefile ###############################
   # a Makefile works accordingly to the following syntax:
   # target: dependencies
   #    <TAB> command1
   #    <TAB> command2
6  #
   #   - all: the name of the rules to be executed
   #   - teste.o: teste.c (can be interpreted as destination_file: origin_file)
   #   - clean: deletes the intermediary files.
   #   - mrproper: deletes everything that must be modified
11 #
   # ----- Defining variables ----------------------------------
   #  VAR_NAME=VAL
   #  and to use the variable we placed between $(): $(VAR_NAME)
   # #Using wildcards to help define variables
16 # SRC=$(wildcard *.c)
   # OBJ=$(SRC:.c=.o)
   # ----------------------------------------------------------
   # #Silent Mode (no echo in the command line)
   # add @ to the beginning of the command. Ex:
21 #         @$(CC) -o $@ $^
   # ------------------- Internal Variables ---------------------
   # $@  Name of the rule
   # $<  Name of the 1st dependency
   # $^   List of dependencies
26 # $?   List of dependencies more recent than the rule
   # $*   Name of the file with suffix
   # ----------------------------------------------------------
   #  Rules without dependencies, e.g., actions like clean:, must be called
   #  explicitly, otherwise they wont be executed.
31 #    - make clean will remove all objects files from folder, as described below
   # -------------------- Interference rules -------------------
   #  Generic rules called by default
   #  - .c.o: make a .o file from a .c file
   #  - %o: %.c: the same thing. The line teste.o:teste.c can be modified with this
36 #  rule
   #  - .PHONY: This rule avoids conflicts.
   #    - If one has a clean file in the directory, nothing will happen when one
   #    executes make clean
   #    - So .PHONY says that clean and mrproper should be executed even if
41 #    files with that name exist current path
```

```
# ------------------------------------------------------------
# ------------------------ make install ----------------------
#  Automating the program's installation with the install: rule
#   - install: Places the binary or executable inside a given folder, e.g., /bin
#   or /usr/bin in Linux. It can be any other, using the command mv or cp to
#   move or copy
#   - Lets create 2 vars:
#     - prefix=path/to/proj
#     - bindir=$(prefix)/bin
#     - and we add the rule install:all

# PROJ name
PROJ=Electric-gym

# File Paths
SRC_DIR=.
BIN_DIR=${SRC_DIR}/bin

#LIBS=-lform -lncurses
CFLAGS=-Wall #-W -ansi -pedantic # options passed to the compiler
RM=rm -rf

# documentation
DOXYGEN=/Applications/Doxygen.app/Contents/Resources/doxygen
DOXYFILE=../Doxyfile

# Databases
DB = $(wildcard *.db)

SRC := $(wildcard *.c)

# Debub settings
DEBUG ?= 1
ifeq ($(DEBUG), 1)
    CFLAGS+=-g
else
    CFLAGS+=-DNDEBUG
endif

# CC=gcc #defining the compiler name for C or C++ (here its gcc) // bad practice
# to define it (not portable)
OBJ := $(SRC:.c=.o)
```

```
# Building executable (w/ linking)
$(PROJ): $(OBJ)
# the compiler does the linking between the 2 objects
# gcc -o teste teste.o main.o
# $@ = teste:     #Rule name
# $^ = teste.o main.o   #dependency list
  @echo "Creating executable"
  $(CC) -o $@ $^ ${LIBS}

# Building object files from sources
%.o: ${SRC_DIR}/%.c # make a .o file from a .c file
# gcc -o teste.o -c teste.c -W -Wall -ansi -pedantic
# $< Name of the 1st dependency
  @echo "Building object files"
  $(CC) -o $@ -c $^ $(CFLAGS)

all: $(PROJ)
  @echo "Compiling project"

# Install: run make and then make install
install: all clean
  @echo "Installing binaries"
  @mkdir -p $(BIN_DIR)
  @mv $(PROJ) $(BIN_DIR)/
  @mv $(BIN_DIR) ../

.PHONY: clean mrproper clean-all doc
clean-all: clean mrproper
clean:
   @- $(RM) $(OBJ)
#  @- $(RM) $(DB)
mrproper: clean
  @$(RM) $(PROJ)
# Documentation
doc:
  @echo "Generating documentation"
  @$(DOXYGEN) $(DOXYFILE)
```

Listing 3.1: Makefile example

3.7.2. Other build systems for C/C++

Make is just part of the simplest building system for C/C++. But there are other ones which are noteworthy[26]:

- Autotools – **Automake** and **Autoconf**: In 1991, David J. MacKenzie got tired of customizing Makefile for the 20 platforms he had to deal with. Instead, he handcrafted a little shell script called **configure** to automatically adjust the Makefile. Compiling his package was now as simple as running **./configure && make** [27]. The Autotools are tools that will create a GNU Build System for your package. Autoconf mostly focuses on **configure** and **automake** on Makefiles.
  Simply put, **automake** aims to allow the programmer to write a makefile in a higher-level language, rather than having to write the whole makefile manually. However, the system is very complex and has a steep learning curve.
- CMake: CMake reads the projects to build from **CMakeLists.txt** files written in a language of its own. From there it generates some Makefiles (or equivalent project files for Xcode or Visual Studio for example) that can be used to build the project.
  The main criticism against CMake is its language, requiring another syntax to be learned and that it generates of lot of intermediate files.

3.8. Source code documentation

Source code documentation is critical for software maintenance, specially on large developments teams. As software applications grow in size and evolve several modifications from base source code arise, possible with several branches to implement different features or correct bugs. A good practice that helps to reduce the complexity burden when interfacing and maintaining such systems is to document the source code, making it readable and easily understood by other people.

Source code documentation aims to describe how the code works instead of what it does, and it is useful for the following reasons [**?**]:

- Knowledge transfer: not all code is equally obvious. There might be some complex algorithms or custom workarounds that are not clear enough for other developers.
- Troubleshooting: if there are any problems with the product after it's released, having proper documentation can speed up the resolution time. Finding out product details and architecture specifics is a time-consuming task, which results in the additional costs.
- Integration: product documentation describes dependencies between system modules and third-party tools. Thus, it may be needed for integration purposes.
- Code style enforcement: the source code documentation tools requires specific comment syntax,

which forces the developer to follow a strict code style. Having a code style standard is specially useful on large project teams.

There are some key ideas to write good documentation [**?**]:

- Simple and concise. Follow the DRY (Don't Repeat Yourself) principle. Use comments to explain something that requires detailed information.
- Up to date at all times: the code should be documented when it's being written or modified.
- Document any changes to the code. Documenting new features or add-ons is pretty obvious. However, you should also document deprecated features, capturing any change in the product.
- Simple language and proper formatting: Code documents are typically written in English so that any developer could read the comments, regardless of their native language. The best practices for documentation writing require using the Imperative mood, Present tenses, preferably active voice, and second person.

There are several automatic source code documentation tools, typically language-specific, namely [**?**]:

- Doxygen: C, C++, C#, Java, Objective-C, PHP, Python
- GhostDoc: C#, Visual Basic, C++, JavaScript
- Javadoc: Java only
- Docurium or YARD: Ruby
- jsdoc: Javascript
- Sphinx: Python, C/C++, Ruby, etc.

3.8.1. Doxygen

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL and to some extent D. Doxygen is highly portable, running on MacOS, Linux and Windows platforms [**?**].

Doxygen supports the user in two ways [**?**]:

1. Flexibility in documentation generation: It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in **LaTeX**) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
2. Assistance in reverse engineering SW: You can configure doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distribu-

tions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.

Documenting the code

Doxygen uses special comment blocks, i.e., comment blocks with some additional markings to identify source code annotations for the documentation [**?**]. As aforementioned, Doxygen supports several programming languages, but here one focus only on C-like languages, specifically **C** and **C++**.

For each entity in the code there are two (or in some cases three) types of descriptions, which together form the documentation for that entity, namely [**?**]:

1. Brief description: short one-liner, briefly documenting the following block. It is optional.
2. Detailed description: provides more detailed documentation about the following block. It is optional.
3. Body description: for methods and functions there is also a third type of description, consists of the concatenation of all comment blocks found within the body of the method or function.

There are several ways to mark a comment block as a detailed description:

1. Javadoc style: consists of a C-style comment block starting with two *'s, as follows:

```
/**
 * Javadoc style ... text ...
 */
```

2. Qt style: add an exclamation mark (!) after the opening of a C-style comment block, as follows:

```
/*!
 * Qt style ... text ...
 */
```

3. slash style: use a block of at least two C++ comment lines, where each line starts with an additional slash or an exclamation mark, as follows:

```
///
/// slash style ... text ...
///
```

The style adopted by the authors is the **Javadoc**. Besides comment blocks, annotations can be used at other levels:

- inline: used to describe parameters, class members, and structure and enumeration fields.

```
    List_T menus; /**< Menus list */
```

- file: used to describe modules or classes. The tags **@file**, **@author** and **@date** are used to identify the file name, the author and the creation date; the tag **@brief** provides a brief description about the file and in the last lines is the detailed description.

```
/**
 * @file App.h
 * @author Jose Pires
 * @date 12 Jan 2019
 *
 * @brief App module containing the application logic
 *
 * It contains only two public functions:
 * 1. Init - to initialize the app's memory
 * 2. Exec - contains all application logic
 */
```

Listing 3.2 illustrates an example of a documented C header file using Doxygen in the Javadoc style. Lines 1 through 11 document the header file, explaining its purpose and the available public interface. Lines 16-19 document the opaque pointer to a **struct App_T**. Lines 22-39 document the public interface of the module — **App_init** and **App_exec**. Specifically, considering the latter, it showcases how to document functions:

- **@brief**: briefly describing the function's purpose.
- **@param**: describing all parameters and its prerequisites, in this case, **app**.
- **@return**: identifying the return value of the function.
- Detailed description: providing further details about the function's operation.

```
/**
 * @file App.h
 * @author Jose Pires
 * @date 12 Jan 2019
 *
 * @brief App module containing the application logic
 *
 * It contains only two public functions:
 * 1. Init - to initialize the app's memory
 * 2. Exec - contains all application logic
 */

#ifndef App_H
#define App_H

/**
```

```
17  * @brief opaque pointer to struct App_T.
    * It hides the implementation details (allows modularity)
19  */
    typedef struct App_T *App_T;

    /* ========== External accessable functions ============== */
23  /**
    * allocate dynamic memory and initialze App
25  * @return initialized App_T
    */
27  App_T App_init();

29  /**
    * @brief App controller: handles all application's logic
31  * @param app - an initialized application instance
    * @return an integer signaling the execution state
33  *
    * Its the execution loop controlling the FSM machine behind the application's
35  * logic. It starts in S_Login state. When end user quits the application,
    * the databases are saved, for future restoration at subsequent program
37  * executions.
    */
39  int App_exec(App_T app);
    /* ============================================================ */
41  #endif // App_H
```

Listing 3.2: Example of a documented C header file using Doxygen — Javadoc style

Listing 3.3 illustrates an example of a documented C implementation file using Doxygen in the Javadoc style. As a recommendation, public interfaces should be documented in the header file and the private implementation details in the implementation file. Once again, the file is documented; inline comments are used for **#define**'s, enumeration and structure fields; brief descriptions for enumerations and structures; and function's comment blocks to document private interfaces.

```
1   /**
    * @file App.c
3   * @author Jose Pires
    * @date 12 Jan 2019
5   *
    * @brief App's module implementation
7   */
```

```
9  #include <stdio.h>
   #include <stdlib.h>
11 #include <assert.h>
   #include <strings.h>
13 #include "App.h"
   #include "list.h"
15 #include "User.h"
   #include "Activity.h"
17 #include "Menu.h"
   #include "Pack.h"
19 #include "Database.h"
   #include "m-utils.h"

   /* Menus titles */
23 #define MENU_LOGIN "LOGIN" /**< Menu title for Login state */
   #define MENU_GERENTE "GERENTE" /**< Menu title for Gerente state */
25 #define MENU_FUNC "FUNCIONARIO" /**< Menu title for Func state */
   #define MENU_CLIENTE "CLIENTE" /**< Menu title for Client state */
27 #define MENU_MANAGE_CLI "GERIR CLIENTE" /**< Menu title for ManageClient state */
   #define MENU_MANAGE_ACT "GERIR ACTIVIDADE" /**< Menu title for ManageActivity state */
29 #define MENU_MANAGE_PACK "GERIR PACK" /**< Menu title for Manage Pack state */
   #define MENU_EDIT_USER "EDITAR UTILIZADOR" /**< Menu title for Edit User state */
31 #define MENU_EDIT_ACT "EDITAR ACTIVIDADE" /**< Menu title for Edit Activity state */
   #define MENU_SEARCH_ACT "PROCURAR ACTIVIDADE" /**< Menu title for search Activity state */
33 #define MENU_EDIT_PACK "EDITAR PACK" /**< Menu title for Edit Pack state */
   #define MENU_ACTIV "ACTIVIDADES" /**< Menu title for Client Activities state */

   /* Database files */
37 #define DATABASE_USERS "user.db" /**< Database file for users */
   #define DATABASE_ACTIVITIES "act.db" /**< Database file for activities */
39 #define DATABASE_PACKS "pack.db" /**< Database file for packs */

41 /**
    * @brief Constants for the App's states. Used in FSM management.
43  */
   enum App_state{
45     S_Login, /**< Login state */
       S_Gerente, /**< Manager state */
47     S_Func,/**< Employee state */
       S_Cliente,/**< Client state */
49     S_Manage_Cli,/**< Manage Client state */
       S_Manage_Act,/**< Manage Activity state */
51     S_Manage_Pack,/**< Manage Pack state */
```

```
    S_Edit_User,/**< Edit User state */
    S_Edit_Act,/**< Edit Activity state */
    S_Edit_Pack,/**< Edit Pack state */
    S_Activities,/**< Client Activities state */
    S_Logout,/**< Logout state */
    S_Quit/**< Quit state */
};

/**
 * @brief App's struct: contains the relevant data members
 */
struct App_T
{
    List_T menus; /**< Menus list */
    List_T users;  /**< Users list */
    List_T activities; /**< Activities list */
    List_T packs; /**< List of available packs */
    User_T cur_user; /**< Current user */
    enum App_state state; /**< App's current state */
    enum App_state prev_state; /**< App's previous state */
    void * userdata; /**< ptr to generic data (used in App_state_functions) */
    Database_T db_user; /**< Users database */
    Database_T db_act; /**< Activities database */
    Database_T db_pack; /**< Packs database */
};

/**
 * @brief Allocates memory for an App's instance
 * @return initialized memory for App
 *
 * It is checked by assert to determine if memory was allocated.
 * If assertion is valid, returns a valid memory address
 */
static App_T App_new()
{
    App_T app = malloc(sizeof(*app));
    assert(app);
    return app;
}
```

Listing 3.3: Example of a documented C implementation file using Doxygen — Javadoc style

Generating the documentation

To generate the documentation, **Doxygen** must be installed in the system and a **Doxyfile** must be provided to assist in this process. Listing **??** presents an excerpt of a **Doxyfile** comprising the following sections:

- Project related options — lines 1–92: contains the encoding, the project's name, version and brief description, the output's directory and language.
- Build related configuration options — lines 93—132: describes how to extract tags from the annotated source files for private and static members, packages and classes.
- Configuration options related to warning and progress messages — lines 133–167: describes how and when to generate these messages.
- Configuration options related to input files — lines 168–204: contains the input directory and encoding, the exclude patterns and the option to use a **readme** file.
- Configuration options related to source browsing — lines 205–217: defines if source file browsing is possible.
- Configuration options related to the alphabetical class index — lines 218–228: defines if an alphabetical index of compounds is required (classes, structures, enumerations, unions or interfaces)
- Configuration options related to the Hypertext Markup Language (HTML) output — lines 229–252: enables **HTML** output and defines its output format and file extension.
- Configuration options related to the **LaTeX** output — lines 253–282: enables **LaTeX** output and defines its output format and processing command.
- Configuration options related to the preprocessor — lines 283–292: enables C-processor directives found in the source and include files.
- Configuration options related to the dot tool — lines 293–424: **dot** is a tool for drawing diagrams. This enables the generation of class diagrams and graphs, collaboration and dependency graphs, the template relations, a hierarchical view of the all classes and a directory graph.

```
# Doxyfile 1.8.15
# This file describes the settings to be used by the documentation system
# doxygen (www.doxygen.org) for a project.
#
# All text after a double hash (##) is considered a comment and is placed in
# front of the TAG it is preceding.
#
# All text after a single hash (#) is considered a comment and will be ignored.
# The format is:
# TAG = value [value, ...]
# For lists, items can also be appended using:
# TAG += value [value, ...]
```

```
# Values that contain spaces should be placed between quotes (\" \").

#---------------------------------------------------------------------------
# Project related configuration options
#---------------------------------------------------------------------------

# This tag specifies the encoding used for all characters in the configuration
# file that follow. The default is UTF-8 which is also the encoding used for all
# text before the first occurrence of this tag. Doxygen uses libiconv (or the
# iconv built into libc) for the transcoding. See
# https://www.gnu.org/software/libiconv/ for the list of possible encodings.
# The default value is: UTF-8.

DOXYFILE_ENCODING      = UTF-8

# The PROJECT_NAME tag is a single word (or a sequence of words surrounded by
# double-quotes, unless you are using Doxywizard) that should identify the
# project for which the documentation is generated. This name is used in the
# title of most generated pages and in a few other places.
# The default value is: My Project.

PROJECT_NAME           = ElectricGym

# The PROJECT_NUMBER tag can be used to enter a project or revision number. This
# could be handy for archiving the generated documentation or if some version
# control system is used.

PROJECT_NUMBER         = v2

# Using the PROJECT_BRIEF tag one can provide an optional one line description
# for a project that appears at the top of each page and should give viewer a
# quick idea about the purpose of the project. Keep the description short.

PROJECT_BRIEF          = "Management application for a Gym"

# With the PROJECT_LOGO tag one can specify a logo or an icon that is included
# in the documentation. The maximum height of the logo should not exceed 55
# pixels and the maximum width should not exceed 200 pixels. Doxygen will copy
# the logo to the output directory.

PROJECT_LOGO           =

# The OUTPUT_DIRECTORY tag is used to specify the (relative or absolute) path
```

```
# into which the generated documentation will be written. If a relative path is
# entered, it will be relative to the location where doxygen was started. If
# left blank the current directory will be used.

OUTPUT_DIRECTORY       = /Users/zemiguel/Trab/code/doc/doxygen

# The OUTPUT_LANGUAGE tag is used to specify the language in which all
# documentation generated by doxygen is written. Doxygen will use this
# information to generate all constant output in the proper language.
# Possible values are: Afrikaans, Arabic, Armenian, Brazilian, Catalan, Chinese,
# Chinese-Traditional, Croatian, Czech, Danish, Dutch, English (United States),
# Esperanto, Farsi (Persian), Finnish, French, German, Greek, Hungarian,
# Indonesian, Italian, Japanese, Japanese-en (Japanese with English messages),
# Korean, Korean-en (Korean with English messages), Latvian, Lithuanian,
# Macedonian, Norwegian, Persian (Farsi), Polish, Portuguese, Romanian, Russian,
# Serbian, Serbian-Cyrillic, Slovak, Slovene, Spanish, Swedish, Turkish,
# Ukrainian and Vietnamese.
# The default value is: English.

OUTPUT_LANGUAGE        = English

# The TAB_SIZE tag can be used to set the number of spaces in a tab. Doxygen
# uses this value to replace tabs by spaces in code fragments.
# Minimum value: 1, maximum value: 16, default value: 4.

TAB_SIZE               = 4

# If the MARKDOWN_SUPPORT tag is enabled then doxygen pre-processes all comments
# according to the Markdown format, which allows for more readable
# documentation. See https://daringfireball.net/projects/markdown/ for details.
# The output of markdown processing is further processed by doxygen, so you can
# mix doxygen, HTML, and XML commands with Markdown formatting. Disable only in
# case of backward compatibilities issues.
# The default value is: YES.

MARKDOWN_SUPPORT       = YES

#---------------------------------------------------------------------------
# Build related configuration options
#---------------------------------------------------------------------------

# If the EXTRACT_ALL tag is set to YES, doxygen will assume all entities in
# documentation are documented, even if no documentation was available. Private
```

```
# class members and static file members will be hidden unless the
# EXTRACT_PRIVATE respectively EXTRACT_STATIC tags are set to YES.
# Note: This will also disable the warnings about undocumented members that are
# normally produced when WARNINGS is set to YES.
# The default value is: NO.

EXTRACT_ALL            = NO

# If the EXTRACT_PRIVATE tag is set to YES, all private members of a class will
# be included in the documentation.
# The default value is: NO.

EXTRACT_PRIVATE        = NO

# If the EXTRACT_PACKAGE tag is set to YES, all members with package or internal
# scope will be included in the documentation.
# The default value is: NO.

EXTRACT_PACKAGE        = NO

# If the EXTRACT_STATIC tag is set to YES, all static members of a file will be
# included in the documentation.
# The default value is: NO.

EXTRACT_STATIC         = YES

# If the EXTRACT_LOCAL_CLASSES tag is set to YES, classes (and structs) defined
# locally in source files will be included in the documentation. If set to NO,
# only classes defined in header files are included. Does not have any effect
# for Java sources.
# The default value is: YES.

EXTRACT_LOCAL_CLASSES  = YES

#---------------------------------------------------------------------------
# Configuration options related to warning and progress messages
#---------------------------------------------------------------------------

# The QUIET tag can be used to turn on/off the messages that are generated to
# standard output by doxygen. If QUIET is set to YES this implies that the
# messages are off.
# The default value is: NO.
```

```
QUIET                  = NO

# The WARNINGS tag can be used to turn on/off the warning messages that are
# generated to standard error (stderr) by doxygen. If WARNINGS is set to YES
# this implies that the warnings are on.
#
# Tip: Turn warnings on while writing the documentation.
# The default value is: YES.

WARNINGS               = YES

# If the WARN_IF_UNDOCUMENTED tag is set to YES then doxygen will generate
# warnings for undocumented members. If EXTRACT_ALL is set to YES then this flag
# will automatically be disabled.
# The default value is: YES.

WARN_IF_UNDOCUMENTED   = YES

# If the WARN_IF_DOC_ERROR tag is set to YES, doxygen will generate warnings for
# potential errors in the documentation, such as not documenting some parameters
# in a documented function, or documenting parameters that don't exist or using
# markup commands wrongly.
# The default value is: YES.

WARN_IF_DOC_ERROR      = YES

#---------------------------------------------------------------------------
# Configuration options related to the input files
#---------------------------------------------------------------------------

# The INPUT tag is used to specify the files and/or directories that contain
# documented source files. You may enter file names like myfile.cpp or
# directories like /usr/src/myproject. Separate the files or directories with
# spaces. See also FILE_PATTERNS and EXTENSION_MAPPING
# Note: If this tag is empty the current directory is searched.

INPUT                  = /Users/zemiguel/Trab/code/

# This tag can be used to specify the character encoding of the source files
# that doxygen parses. Internally doxygen uses the UTF-8 encoding. Doxygen uses
# libiconv (or the iconv built into libc) for the transcoding. See the libiconv
# documentation (see: https://www.gnu.org/software/libiconv/) for the list of
# possible encodings.
```

```
# The default value is: UTF-8.

INPUT_ENCODING         = UTF-8

# If the value of the INPUT tag contains directories, you can use the
# EXCLUDE_PATTERNS tag to specify one or more wildcard patterns to exclude
# certain files from those directories.
#
# Note that the wildcards are matched against the file with absolute path, so to
# exclude all test directories for example use the pattern */test/*

EXCLUDE_PATTERNS       = */tests/* */old/* */versions/*

# If the USE_MDFILE_AS_MAINPAGE tag refers to the name of a markdown file that
# is part of the input, its contents will be placed on the main page
# (index.html). This can be useful if you have a project on for instance GitHub
# and want to reuse the introduction page also for the doxygen output.

USE_MDFILE_AS_MAINPAGE = /Users/zemiguel/Trab/code/readme.md

#---------------------------------------------------------------------------
# Configuration options related to source browsing
#---------------------------------------------------------------------------

# If the SOURCE_BROWSER tag is set to YES then a list of source files will be
# generated. Documented entities will be cross-referenced with these sources.
#
# Note: To get rid of all source code in the generated output, make sure that
# also VERBATIM_HEADERS is set to NO.
# The default value is: NO.

SOURCE_BROWSER         = YES

#---------------------------------------------------------------------------
# Configuration options related to the alphabetical class index
#---------------------------------------------------------------------------

# If the ALPHABETICAL_INDEX tag is set to YES, an alphabetical index of all
# compounds will be generated. Enable this if the project contains a lot of
# classes, structs, unions or interfaces.
# The default value is: YES.

ALPHABETICAL_INDEX     = YES
```

```
#---------------------------------------------------------------------------
# Configuration options related to the HTML output
#---------------------------------------------------------------------------

# If the GENERATE_HTML tag is set to YES, doxygen will generate HTML output
# The default value is: YES.

GENERATE_HTML          = YES

# The HTML_OUTPUT tag is used to specify where the HTML docs will be put. If a
# relative path is entered the value of OUTPUT_DIRECTORY will be put in front of
# it.
# The default directory is: html.
# This tag requires that the tag GENERATE_HTML is set to YES.

HTML_OUTPUT            = html

# The HTML_FILE_EXTENSION tag can be used to specify the file extension for each
# generated HTML page (for example: .htm, .php, .asp).
# The default value is: .html.
# This tag requires that the tag GENERATE_HTML is set to YES.

HTML_FILE_EXTENSION    = .html

#---------------------------------------------------------------------------
# Configuration options related to the LaTeX output
#---------------------------------------------------------------------------

# If the GENERATE_LATEX tag is set to YES, doxygen will generate LaTeX output.
# The default value is: YES.

GENERATE_LATEX         = YES

# The LATEX_OUTPUT tag is used to specify where the LaTeX docs will be put. If a
# relative path is entered the value of OUTPUT_DIRECTORY will be put in front of
# it.
# The default directory is: latex.
# This tag requires that the tag GENERATE_LATEX is set to YES.

LATEX_OUTPUT           = latex

# The LATEX_CMD_NAME tag can be used to specify the LaTeX command name to be
```

```
# invoked.
#
# Note that when not enabling USE_PDFLATEX the default is latex when enabling
# USE_PDFLATEX the default is pdflatex and when in the later case latex is
# chosen this is overwritten by pdflatex. For specific output languages the
# default can have been set differently, this depends on the implementation of
# the output language.
# This tag requires that the tag GENERATE_LATEX is set to YES.

LATEX_CMD_NAME         =

#---------------------------------------------------------------------------
# Configuration options related to the preprocessor
#---------------------------------------------------------------------------

# If the ENABLE_PREPROCESSING tag is set to YES, doxygen will evaluate all
# C-preprocessor directives found in the sources and include files.
# The default value is: YES.

ENABLE_PREPROCESSING   = YES

#---------------------------------------------------------------------------
# Configuration options related to the dot tool
#---------------------------------------------------------------------------

# If the CLASS_DIAGRAMS tag is set to YES, doxygen will generate a class diagram
# (in HTML and LaTeX) for classes with base or super classes. Setting the tag to
# NO turns the diagrams off. Note that this option also works with HAVE_DOT
# disabled, but it is recommended to install and use dot, since it yields more
# powerful graphs.
# The default value is: YES.

CLASS_DIAGRAMS         = YES

# You can define message sequence charts within doxygen comments using the \msc
# command. Doxygen will then run the mscgen tool (see:
# http://www.mcternan.me.uk/mscgen/)) to produce the chart and insert it in the
# documentation. The MSCGEN_PATH tag allows you to specify the directory where
# the mscgen tool resides. If left empty the tool is assumed to be found in the
# default search path.

MSCGEN_PATH            =
```

```
# You can include diagrams made with dia in doxygen documentation. Doxygen will
# then run dia to produce the diagram and insert it in the documentation. The
# DIA_PATH tag allows you to specify the directory where the dia binary resides.
# If left empty dia is assumed to be found in the default search path.

DIA_PATH               =

# If set to YES the inheritance and collaboration graphs will hide inheritance
# and usage relations if the target is undocumented or is not a class.
# The default value is: YES.

HIDE_UNDOC_RELATIONS   = YES

# If you set the HAVE_DOT tag to YES then doxygen will assume the dot tool is
# available from the path. This tool is part of Graphviz (see:
# http://www.graphviz.org/), a graph visualization toolkit from AT&T and Lucent
# Bell Labs. The other options in this section have no effect if this option is
# set to NO
# The default value is: NO.

HAVE_DOT               = YES


# If the CLASS_GRAPH tag is set to YES then doxygen will generate a graph for
# each documented class showing the direct and indirect inheritance relations.
# Setting this tag to YES will force the CLASS_DIAGRAMS tag to NO.
# The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.

CLASS_GRAPH            = YES

# If the COLLABORATION_GRAPH tag is set to YES then doxygen will generate a
# graph for each documented class showing the direct and indirect implementation
# dependencies (inheritance, containment, and class references variables) of the
# class with other documented classes.
# The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.

COLLABORATION_GRAPH    = YES

# If the GROUP_GRAPHS tag is set to YES then doxygen will generate a graph for
# groups, showing the direct groups dependencies.
# The default value is: YES.
```

```
# This tag requires that the tag HAVE_DOT is set to YES.

GROUP_GRAPHS           = YES

# If the UML_LOOK tag is set to YES, doxygen will generate inheritance and
# collaboration diagrams in a style similar to the OMG's Unified Modeling
# Language.
# The default value is: NO.
# This tag requires that the tag HAVE_DOT is set to YES.

UML_LOOK               = YES

# If the TEMPLATE_RELATIONS tag is set to YES then the inheritance and
# collaboration graphs will show the relations between templates and their
# instances.
# The default value is: NO.
# This tag requires that the tag HAVE_DOT is set to YES.

TEMPLATE_RELATIONS     = YES

# If the INCLUDE_GRAPH, ENABLE_PREPROCESSING and SEARCH_INCLUDES tags are set to
# YES then doxygen will generate a graph for each documented file showing the
# direct and indirect include dependencies of the file with other documented
# files.
# The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.

INCLUDE_GRAPH          = YES

# If the INCLUDED_BY_GRAPH, ENABLE_PREPROCESSING and SEARCH_INCLUDES tags are
# set to YES then doxygen will generate a graph for each documented file showing
# the direct and indirect include dependencies of the file with other documented
# files.
# The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.

INCLUDED_BY_GRAPH      = YES

# If the GRAPHICAL_HIERARCHY tag is set to YES then doxygen will graphical
# hierarchy of all classes instead of a textual one.
# The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.
```

```
GRAPHICAL_HIERARCHY     = YES

# If the DIRECTORY_GRAPH tag is set to YES then doxygen will show the
# dependencies a directory has on other directories in a graphical way. The
# dependency relations are determined by the #include relations between the
# files in the directories.
# The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.

DIRECTORY_GRAPH         = YES

# The DOT_IMAGE_FORMAT tag can be used to set the image format of the images
# generated by dot. For an explanation of the image formats see the section
# output formats in the documentation of the dot tool (Graphviz (see:
# http://www.graphviz.org/)).
# Note: If you choose svg you need to set HTML_FILE_EXTENSION to xhtml in order
# to make the SVG files visible in IE 9+ (other browsers do not have this
# requirement).
# Possible values are: png, jpg, gif, svg, png:gd, png:gd:gd, png:cairo,
# png:cairo:gd, png:cairo:cairo, png:cairo:gdiplus, png:gdiplus and
# png:gdiplus:gdiplus.
# The default value is: png.
# This tag requires that the tag HAVE_DOT is set to YES.

DOT_IMAGE_FORMAT        = png
```

Listing 3.4: Example of a **Doxyfile** — excerpt

Documentation output

As aforementioned in Section 3.8.1, Doxygen extracts the annotations from source code files to build the documentation, in several formats, most notably **html** (online) and **LaTeX** (off-line), and to generate several diagrams that highlight the code structure.

3.9. Scenting technologies

A scenting technology transforms an aromatic liquid into a gaseous fluid that can be conveyed through the air and be captured by human olfactory sense, usually for therapeutic or marketing purposes. As

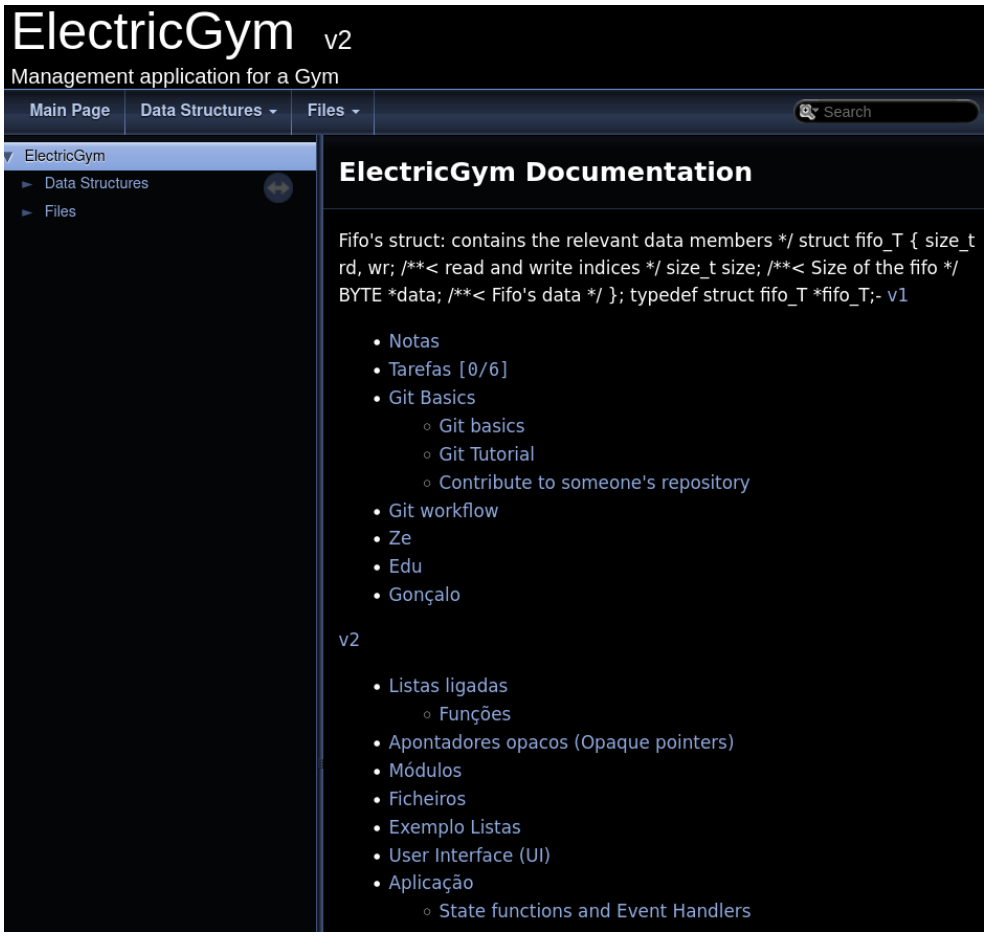


Figure 3.9.: Doxygen output: readme file

aforementioned in Section 1.1, olfactory sense is the fastest way to the brain, thus, providing an exceptional opportunity for marketing [2] — “75% of the emotions we generate on a daily basis are affected by smell. Next to sight, it is the most important sense we have” [3].

In this section a brief overview of the scenting technologies is provided, with special focus on ultrasonic diffusion as simple to control and cost-effective solution.

3.9.1. Overview

There are several scenting technologies, mainly divided into [**?**]:

- Atomization: it dispense odorants by transforming them into a gaseous fluid without requiring to heat. Its advantages are the dispensing process is fast and the dispensing quantity is controllable.
- Thermalization: it dispense odorants by vaporizing odor sources in the liquid state of the solid state using Pulse-Width Modulation (PWM) heaters. It requires a temperature controller to avoid scorching

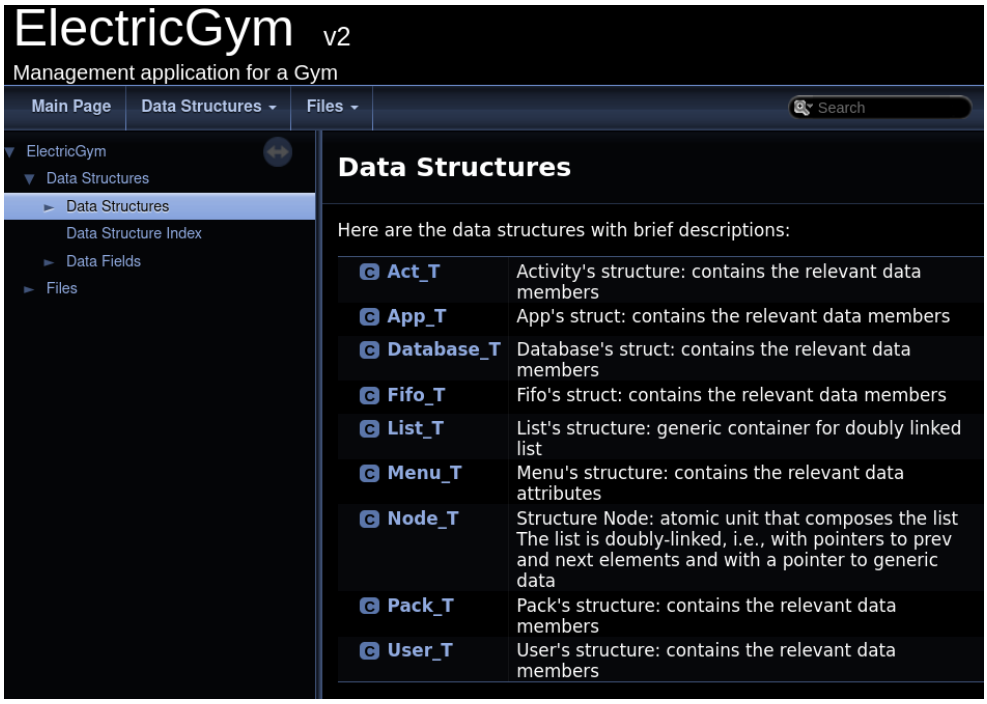


Figure 3.10.: Doxygen output: data structures' overview

odor sources.

- Evaporation: it dispense odorants by conveying the liquid through a porous material into the outer surface (capillary action) where it evaporates naturally. It is a passive method, thus, not controllable.

The thermalization process requires heat which can modify fragrances, besides requiring more power. Evaporation is a passive method, hence, not controllable. Thus, one will focus on the **atomization** processes.

There are several atomization processes, with the most commercially relevant being [**?**]:

- Ultrasonic diffusers: it contains reservoirs for water and essential aromatic oils. It uses mechanical ultrasonic vibrations to brake down water molecules into droplets, producing mist, diffusing the oils into the air. Its advantages are: low power consumption, easy to clean, silent operation, and they double as a humidifier (can be a disadvantage too). Furthermore, they are a very cost-effective solution: the units themselves tend to cost less than nebulizing diffusers on average, but more importantly, ultrasonic diffusers use much less oil than nebulizing diffusers. They also run for longer periods of time, in several cases up to 24 hours before needing to be refilled. As a disadvantage they change the fragrance composition by incorporating water into it (this is not critical).
- Nebulizers: Nebulizing diffusers don't use water. Instead the essential oil is diffused by an air compressor that blows air across the top of the reservoir tube, creating a vacuum which pulls fine particles

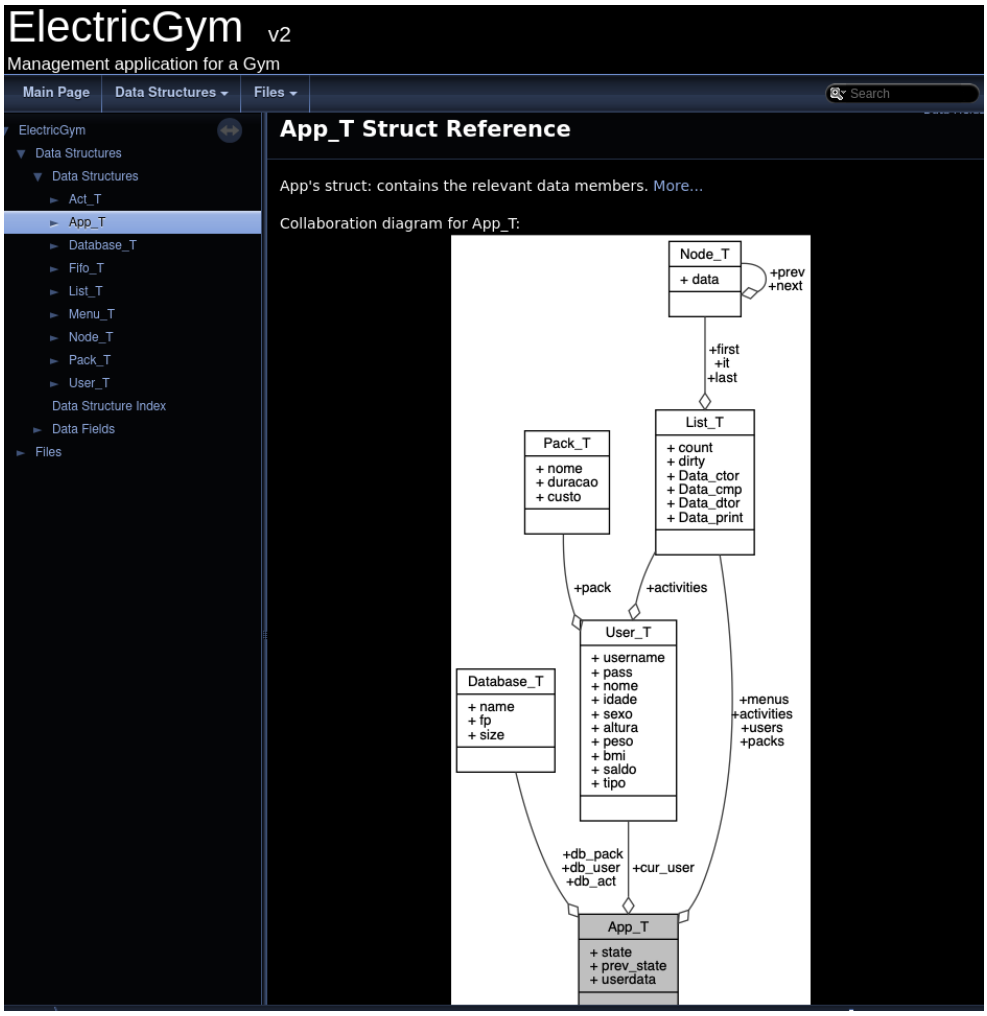


Figure 3.11.: Doxygen output: collaboration graph for a structure

of the essential oil up and sprays them into the air around the unit. Its advantages are: fragrance composition is unaltered, more compact (typically), faster and more concentrated fragrance diffusion. The drawbacks are: less cost-effective when compared to ultrasonic diffusers as the oil consumption rate is much higher and the units are more expensive, and they tend to be noisy due to the air compressor operation.

The ultrasonic diffuser was preferred due to its low power consumption, silent operation, cost-effectiveness, and easy control and assembly. The ultrasonic diffusion process will be detailed in the following section.

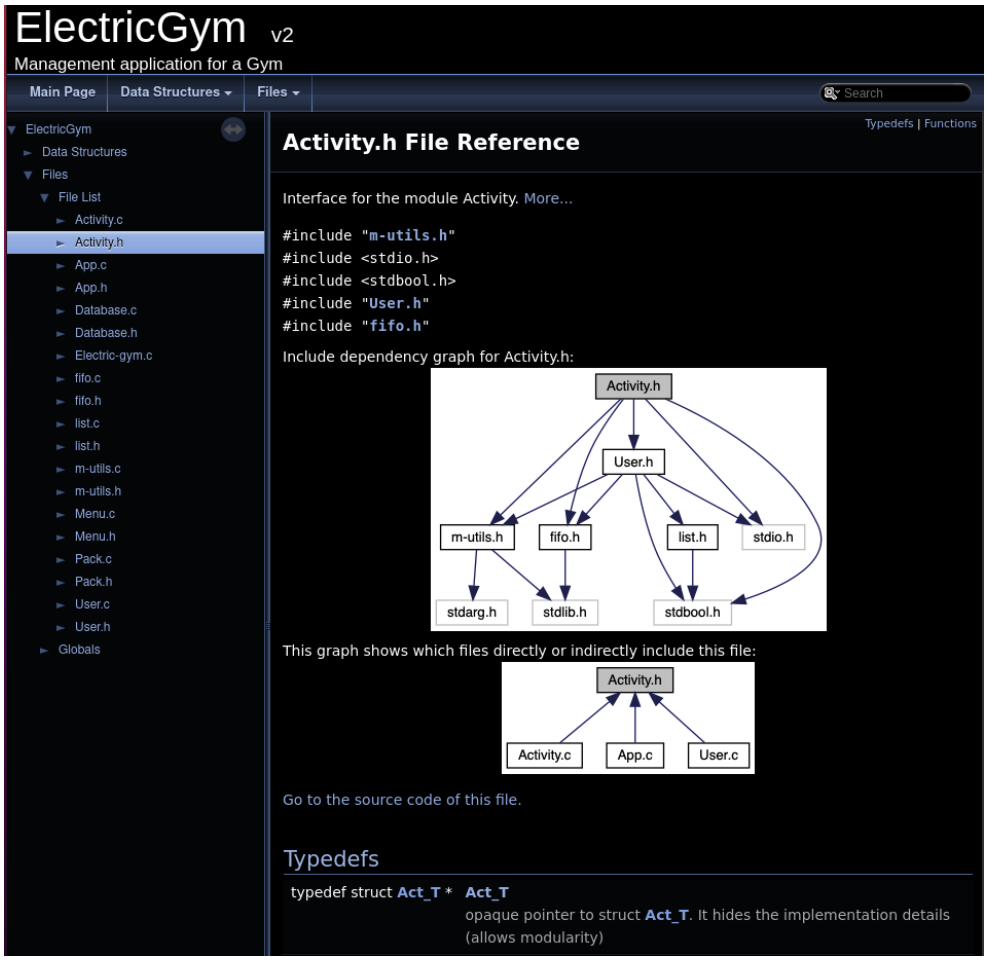


Figure 3.12.: Doxygen output: header file — dependency graph and **typedef**

3.9.2. Ultrasonic diffusion

3.10. Computer vision

Computer vision is a vast field, but can broadly be defined as the transformation of data from a still image or video camera into either a decision or a new representation to achieve some particular goal [28].

The input data may include some contextual information such as 'the camera is mounted in a car' or 'laser range finder indicates an object is 1 meter away.' The decision might be 'there is a person in this scene' or 'there are 14 tumor cells on this slide.' A new representation might mean turning a color image into a grayscale image or removing camera motion from an image sequence [28].

In this section, computer vision frameworks/libraries are listed, with special focus on OpenCV, and computer vision algorithms for face detection and hand gesture recognition are analyzed.

Figure 3.13.: Doxygen output: list of public prototypes for **Activity**'s module

3.10.1. Computer vision frameworks

There are several noteworthy computer vision frameworks, namely [29]:

1. Google Cloud's Vision API: it is an easy-to-use image recognition technology that lets developers understand the content of an image by applying powerful machine learning models. It enables key vision detection features within an application — face, and landmark detection, image labeling, Optical Character Recognition (OCR), and explicit content tagging — and image classification into millions of predefined categories.
2. YOLOv3: YOLO (You Only Look Once) is a state-of-the-art, real-time object detection system among the most widely used deep learning-based object detection methods. It considers object detection as a regression problem, directly predicting the class probabilities and bounding box offsets from full images with a single feed-forward Convolutional Neural Network (CNN). YOLOv3 eliminates region

Figure 3.14.: Doxygen output: implementation file — function details

proposal generation and feature resampling and encapsulates all stages in a single network to form a true end-to-end detection system.

3. TensorFlow: it is a free, open-source framework for creating algorithms to develop a user-friendly Graphical Framework called TensorFlow Graphical Framework (TF-GraF) for object detection API, which is widely applied to solve complex tasks efficiently in agriculture, engineering, and medicine. The TF-GraF provides independent virtual environments for amateurs and beginners to design, train, and deploy machine intelligence models without coding or CLI in the client-side.
4. libfacedetection: it is an open-source library for face detection in images. It uses a pre-trained CNN, enabling face detection on inputs with a size greater than 10×10 pixels. The source code is not dependant on any other libraries. A C++ compiler is required to compile the source under various platforms, such as Windows, Linux, ARM, etc..
5. Raster Vision: it is an open-source Python framework to build computer vision models on satellite, aerial, and other large sets of images (including oblique drone imagery), using deep learning or machine learning models. It has built-in support for chip classification, object detection, and semantic

segmentation with backbends using PyTorch and Tensorflow. The framework is also extensible to new data sources, tasks (e.g., object detection), backend (e.g., TF Object Detection API), and cloud providers.

6. SOD: it is an embedded, modern cross-platform computer vision and machine learning software library. It exposes a set of APIs for deep-learning, advanced media analysis, and processing, including real-time, multi-class object detection, and model training on embedded systems with limited computational resource and Internet of Things (IOT) devices. Designed for computational efficiency and with a strong focus on real-time applications, SOD includes a comprehensive set of both classic and state-of-the-art deep-neural networks with their pre-trained models. Although it is open source, the pre-trained models are charged (one time fee only – up to 30 United States Dollar (USD)).
7. Face_recognition: it is a facial recognition API for Python and the command line, built with deep learning using dlib60's state-of-the-art face recognition. The model has an accuracy of 99.38% on the Labeled Faces in the Wild benchmark.
8. JeelizFaceFilter: it is a lightweight and robust face tracking library, designed for augmented reality face filters. This JavaScript library can detect and track the face in real-time from the webcam video feed captured, enabling the developers to solve computer-vision problems directly from the browser. The key features include face detection, face tracking, face rotation detection, mouth opening detection, multiple face detection, and tracking, video acquisition with High-Definition (HD) video ability, etc.
9. OpenCV: it is an open-source computer vision and machine learning software library, built to provide a common infrastructure for computer vision applications and accelerate the use of machine perception in commercial products. OpenCV was designed for computational efficiency and with a strong focus on real-time applications. It is written in optimized C++ and can take advantage of multicore processors, with wrappers written in Python, Java, and Matlab, and supporting Windows, Linux, Android and Mac OS. The library has more than 2500 optimized algorithms, including a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects and produce 3D point clouds from stereo cameras. It can stitch images together to produce a high-resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize scenery and establish markers to overlay it with augmented reality.

3.10.2. Face detection

Face detection has been studied for decades in the computer vision literature. Modern face detection algorithms can be categorized into four categories [30]: cascade based methods, part based methods, channel feature based methods, and neural network based methods. A detailed survey can be found in [31, 32].

The seminal work by Viola and Jones [33] introduces integral image to compute Haar-like features in constant time. These features are then used to learn AdaBoost classifier with cascade structure for face detection. Various later studies follow a similar pipeline, with SURF cascade [34] achieving competitive performance.

Although the Viola-Jones are extremely fast, they require severe tunning to prevent false-positives or complete misses. In 2005, Dalal and Triggs [35] demonstrated that the Histogram of Oriented Gradients (HOG) image descriptor and a Linear Support Vector Machine (SVM) could be used to train highly accurate object classifiers — or in their particular study, human detectors. However, due to the nature of HOG that are only suited for frontal face detection as it is not invariant to changes in rotation and viewing angle.

One of the well-known part based methods is Deformable part models (DPM) [36]. Deformable part models define face as a collection of parts and model the connections of parts through Latent SVM. The part based methods are more robust to occlusion compared with cascade-based methods.

Aggregated channel feature (ACF) is first proposed by Dollar et al. [37] to solve pedestrian detection. Later on, Yang et al. [38] applied this idea on face detection. In particular, features such as gradient histogram, integral histogram, and color channels are combined and used to learn boosting classifier with cascade structure.

Recent studies [39, 40] show that face detection can be further improved by using deep learning, leveraging the high capacity of deep convolutional networks.

Considering a more pragmatic approach, the following tips are useful when selecting a face detection method [41]:

1. OpenCV's Haar cascades: Use OpenCV's Haar cascades when speed is your primary concern (e.g., when considering an embedded device like the Raspberry Pi). Haar cascades aren't as accurate as their HOG + Linear SVM and deep learning-based counterparts, but they compensate it in raw speed. False-positive detections is highly likely and parameter tuning is required when calling **detectMultiScale**.
2. HOG + linear SVM detector: Use **dlib's HOG + Linear SVM** detector when Haar cascades are not accurate enough, but you cannot commit to the computational requirements of a deep learning-based face detector. The HOG + Linear SVM object detector is a classic algorithm in the computer vision literature that is still relevant today. However, running HOG + Linear SVM on a Central Processing Unit (CPU) will likely be too slow for an embedded device.

3. CNN face detection: Use dlib's CNN face detection when requiring extremely accurate face detections. However, there is a tradeoff — with higher accuracy comes slower run-time. This method cannot run in real-time on a laptop/desktop CPU, and even with Graphics Processing Unit (GPU) acceleration, you'll struggle to hit real-time performance.
4. DNN face detector: Use OpenCV's DNN face detector as a good balance. As a deep learning-based face detector, this method is accurate — and since it's a shallow network with an single-shot detector backbone, it's easily capable of running in real-time on a CPU. Furthermore, since you can use the model with OpenCV's **cv2.dnn** module, that also implies that (1) you can increase speed further by using a GPU or (2) utilizing the Movidius NCS on your embedded device.

Haar cascade classifier

Object detection using Haar feature-based cascade classifiers is an effective method proposed by Viola and Jones [42]. It is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images [43].

The algorithm can be explained in four stages [44]:

1. Calculating Haar Features: the first step is to collect the Haar features. A Haar feature is essentially calculations that are performed on adjacent rectangular regions at a specific location in a detection window. The calculation involves summing the pixel intensities in each region and calculating the differences between the sums. Some examples of Haar features are illustrated in Fig. 3.15.
2. Creating Integral Images: instead of computing at every pixel, sub-rectangles and its array references are created, which are then used to compute the Haar features (see Fig. 3.16).
3. Using Adaboost: to determine the best features from the hundreds of thousands of Haar features, one requires a machine learning model. Adaboost essentially chooses the best features and trains the classifiers to use them. It uses a combination of 'weak classifiers' to create a 'strong classifier' that the algorithm can use to detect objects. Weak learners are created by moving a window over the input image, and computing Haar features for each subsection of the image. This difference is compared to a learned threshold that separates non-objects from objects. Because these are 'weak classifiers', a large number of Haar features is needed for accuracy to form a strong classifier (see Fig. 3.17).
4. Implementing Cascading Classifiers: The cascade classifier is made up of a series of stages, where each stage is a collection of weak learners (see Fig. 3.18). Weak learners are trained using boosting, which allows for a highly accurate classifier from the mean prediction of all weak learners.

   Based on this prediction, the classifier either decides to indicate an object was found (positive) or move on to the next region (negative). Stages are designed to reject negative samples as fast as

possible, because a majority of the windows do not contain anything of interest.

It's important to maximize a low false negative rate, because classifying an object as a non-object will severely impair your object detection algorithm, so it is critical to tune hyperparameters accordingly when training your model.

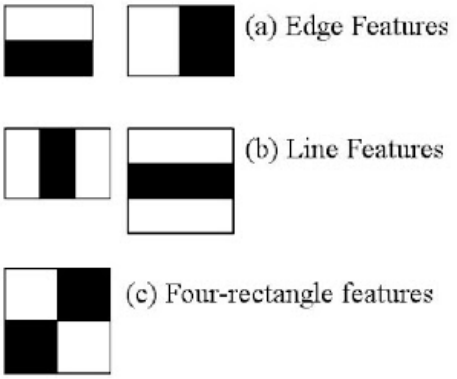


Figure 3.15.: Examples of Haar features (withdrawn from [43])

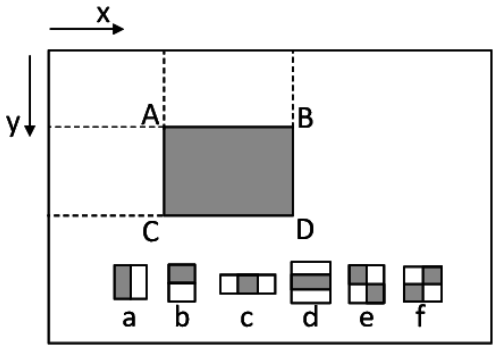


Figure 3.16.: Integral image creation illustration (withdrawn from [44])

3.10.3. Hand gesture recognition

Human communication naturally relies on hand gestures for clarification of intentions, thus making a viable bridge for interchanging information between humans and computers [45]. The most notable applications of hand gesture recognition are: sign language comprehension, robotics control, assisting disabled people, creation of contactless user interfaces, game industry (e.g. Kinect), home automation, clinical and health, virtual environments, etc. [46].

However, the computational performance of recognizing hand gestures is affected by the environmental surroundings (such as light, background, distance, skin color) and the position and direction of hand [47].

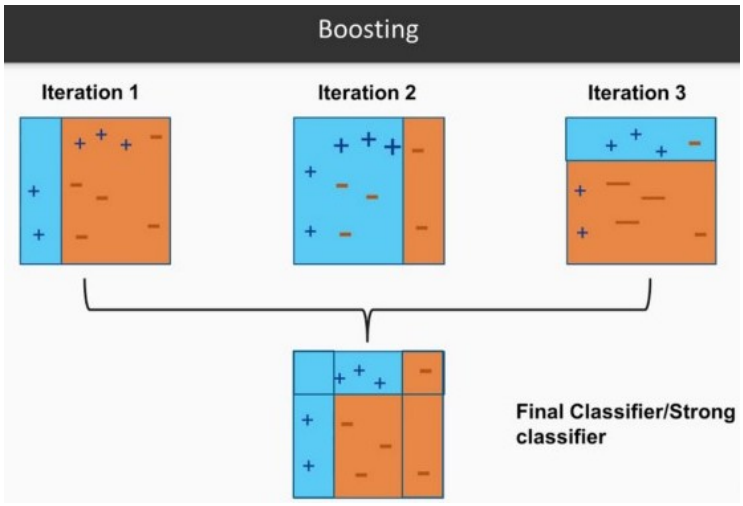


Figure 3.17.: Illustration of a boosting algorithm (withdrawn from [44])

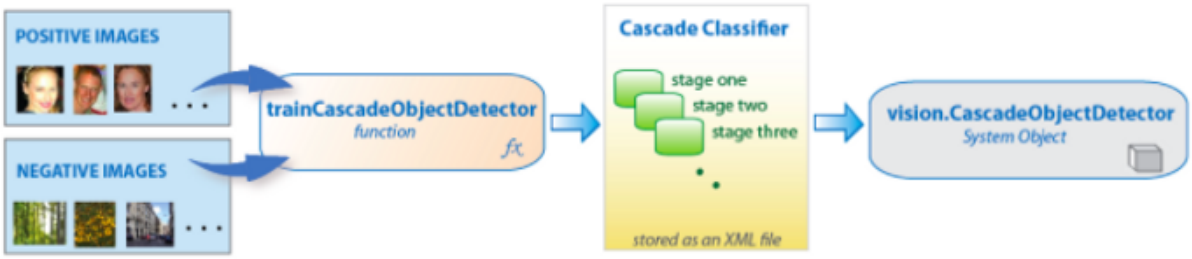


Figure 3.18.: Flowchart of a cascade classifiers (withdrawn from [44])

Additionally, hand gestures applications require users to be well trained at employing and understanding the meaning of different gestures, as for each application, a different group of gestures is used to perform its operations [46].

In this section the hand gesture recognition process is explored, concerning its workflow and most relevant methods.

The basic hand gesture recognition procedure is illustrated in Fig. 3.19, comprising the following stages [46]:

- image frame acquisition: capture the human gesture image by computer, usually performed using a web camera or depth camera [48]. Special tools can be used such as wired or wireless gloves to detect hand movements, and motion sensing input devices (e.g., Kinect, Leap Motion, etc.) to capture the hand gestures and motions.
- Hand tracking: is the ability of the computer to trace the user hand and separate it from the background and from the surroundings objects [48]. This can be done using multi-scale color feature hierarchies that gives the users hand and the background different shades of colors to be able to

identify and remove the background — background subtraction and thresholding, or by using clustering algorithms that are capable of treating each finger as a cluster and removing the empty spaces in-between them [46].

- Feature extraction: The features vary depending on the application, with the most common being: fingers status, thumb status, skin color, alignments of fingers, and the palm position [48]. Several feature extraction methods can be used, such as Fourier descriptor method for capturing the palm, the fingers and the finger tips, or the centroid method which captures the essential structure of the hand [46].
- Classification: The features extracted are then sent to training and testing the classification algorithm (such as Artificial Neural Networks (ANN), K-nearest neighbor (KNN), Naive Bayes (NB), SVM, etc.) to reach the output gesture. A simple case of an output gesture can contain two classes to detect only two gestures such as open and closed hand gestures [46].

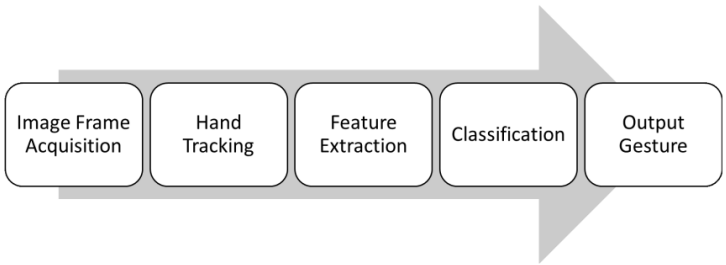


Figure 3.19.: Basic steps of hand gesture recognition (withdrawn from [46])

Fig. 3.20 illustrates an hand gesture recognition workflow, materializing the aforementioned procedure, namely: building a gesture dataset, tracking the hand and extracting the gesture by creating a Region Of Interest (ROI) and applying background subtraction and binary thresholding, training the model with the new dataset, and perform the hand gesture recognition using the predictive model.

3.11. RDBMS

A database is a collection of data, typically describing the activities of one or more related organizations. For example, a university database might contain information about the following [51]:

- Entities: such as students, faculty, courses and classrooms
- Relationships between entities: such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses.

An Database Management System (DBMS) is a software designed to assist in maintaining and utilizing large collections of data. A Relational Database Management System (RDBMS) is a subset of Database

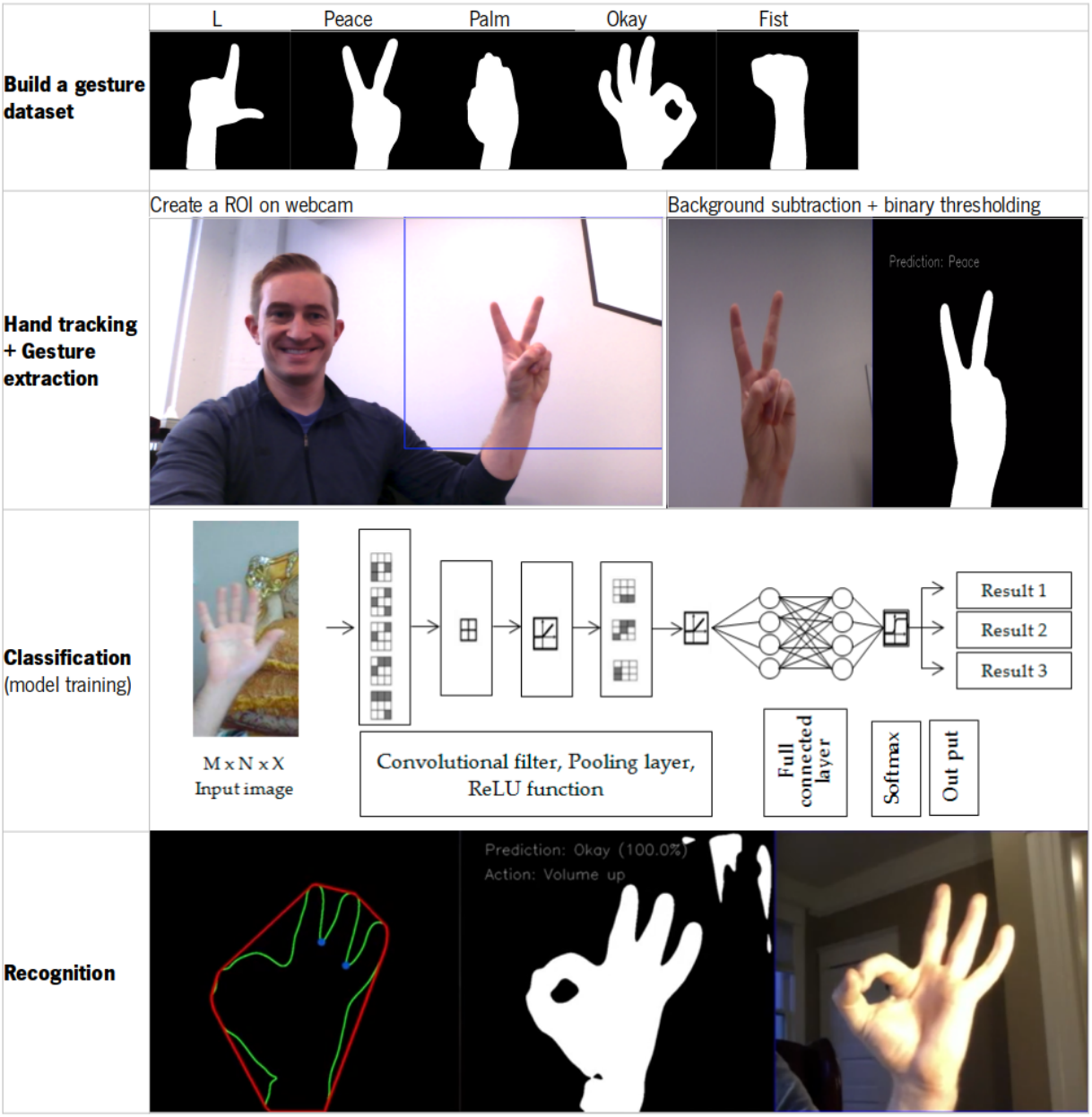


Figure 3.20.: Hand gesture recognition workflow illustrated: example — adapted from [49] and [50]

Management System (DBMS) with relationship between tables (entities) and rows (entities' attributes). It follows the relational model, introduced by E.F. Codd in 1970 [51], instead of navigational model, where in the data is stored in multiple tables. The tables are related to each other using primary and foreign keys. It is the most used database model widely used by enterprises and developers for storing complex and huge amounts of data [51]. Some examples of RDBMS are Oracle Database, MySQL, IBM DB2, SQLite, PostgreSQL, and MariaDB.

From the users' application standpoint, a RDBMS is a management system for databases, but is useless unless it provides an efficient and easy method to pose questions involving the data stored in the databases. These questions are called queries [51]. A DBMS provides a specialized language — query language — in which queries can be performed. The Structured Query Language (SQL) for relational databases, is now the standard. Arguably, the most widely used form of concurrent programming is the concurrent execution of database programs (called transactions). Users write programs as if they are to be run by themselves, and the responsibility for running them concurrently is given to the DBMS [51].

In this section an overview is presented about RDBMS foundations: description and storage of data in a DBMS, relational model, levels of abstraction in a DBMS, transaction management, and the structure of a DBMS. Additionally, a brief overview over SQL and a C++ interface is presented.

3.11.1. Description and storage of data in a DBMS

The user of a DBMS is ultimately concerned about the description of various aspects of some real-world enterprise in the form of data. There are two important data models used [51]:

- Data model: collection of high-level data description constructs that hide many low-level storage details. A DBMS allows a user to define the data to be stored in terms of a data model, such as the **relational data model**. It is closer to how the DBMS stores the data.
- Semantic data model: more abstract, high-level data model, closer to human thinking, serving as an useful starting point for the database design. The semantic data model is subsequently translated into a database design in terms of the data model the DBMS actually supports. An example is the Entity-Relationship (ER) model which allows the user to pictorially denote entities and the relationship among them. The semantic data

3.11.2. Relational model

The central data description construct in the relational model is a relation, which can be thought as a set of records [51]. A schema is a description of data in terms of the data model. In the relation model, the schema for a relation specifies its name, the name of each field (or attribute or column), and the type of each field. As an example, student information in a university database may be stored in a relation with the following schema [51]:

Students(sid: string, name: string, login: string, age: integer, gpa: real)

The preceding schema states that each record in the Students relation has five fields, with field names and types as indicated. As example instance of the student relation appears in Table 3.1 [51].

Table 3.1.: An instance of the students relation — withdrawn from [51]

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53932	Guldu	guldu@music	12	2.0

Each row in the Students relation is a record that describes a student, following the schema of the Students relation. Thus, the schema can be thought as a template for describing a student. This description can be made more precise by specifying integrity constraints, i.e., the conditions that the records in a relation must satisfy [51]. For example, one could specify that every student had a unique **sid** value, thus making a potential candidate for a primary key, i.e., an unique identifier that univocally identifies each record in a relation. This information cannot be captured by simply adding another field to the Students schema, thus requiring integrity constraints to increase the expressiveness of the constructs of a data model [51].

3.11.3. Levels of abstraction in a DBMS

The data in a DBMS is described at three levels of abstraction, as illustrated in Fig. 3.22, namely [51]:

- External schema: allow data access to be customized (and authorized) at the level of individual users or group of users. Any given database has exactly one conceptual schema and one physical schema because it has just one set of stored relations, but it may have several external schemas, each tailored to a particular group of users. Each external schema consists of a collection of one or more views and relations from the conceptual schema. A view is conceptually a relation, but the records in a view are not stored in the DBMS. Rather, they are computed using a definition for the view, in terms of relations stored in the DBMS.
- Conceptual schema: also known as the logical schema, describes the stored data in terms of the data model of the DBMS. In a RDBMS, the conceptual schema describes all relations that are stored in the database. The conceptual schema may be design using the Entity-Relationship (ER) model.
- Physical schema: translates how the relations described in the conceptual schema are actually stored on secondary storage devices such as disks and tapes. Decisions about the physical schema are based on the understanding of how the data is typically accessed, typically requiring the design to decide about the file organizations used to store the relations and to create auxiliary data structures called indexes to speed up data retrieval operations.

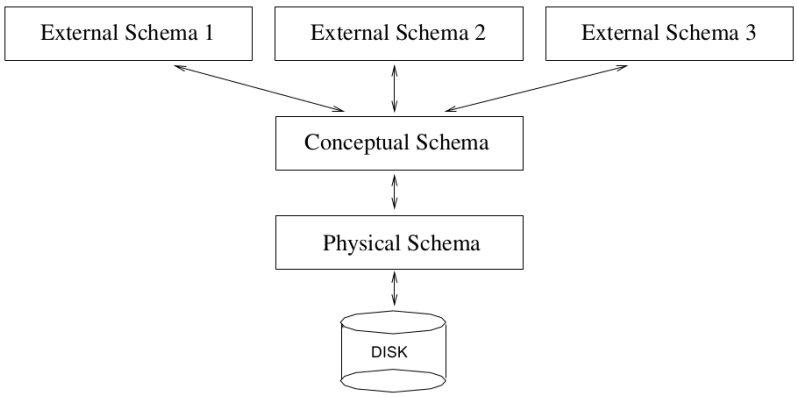


Figure 3.21.: Levels of abstraction in a DBMS (withdrawn from [51])

3.11.4. Transaction management

An important task of a DBMS is to schedule concurrent accesses to data in a safe and seamless way to the user. Such accesses are named transactions, i.e., any one execution of a user program in a DBMS, corresponding to the basic unit of change as seen by the DBMS. Partial transactions are not allowed, and the effect of a group of transactions is equivalent to some serial execution of all transactions [51].

For the concurrent execution of transactions to take place, a locking protocol is enforced by the DBMS, establishing a set of rules to be followed by each transaction, using a lock — a mechanism to control access to database objects. Two kinds of locks are commonly supported by a DBMS: shared locks on an object can be held by two different transactions at the same time, but an exclusive lock on an object ensures that no other transactions hold any lock on this object [51].

Transactions can be interrupted before running to completion for a variety of reasons, e.g., a system crash. A DBMS must ensure that the changes made by such incomplete transactions are removed from the database. To do so, the DBMS maintains a log of all writes to the database, even before the corresponding change is reflected in the database itself, enabling the DBMS to detect and undo the changes if a system crash occurs. This property is called Write-Ahead Log (WAL). To ensure this property, the DBMS must be able to selectively force a page in memory to disk [51].

The log is also used to ensure that the changes made by a successfully completed transaction are not lost due to a system crash, as explained. Bringing the database to a consistent state after a system crash can be a slow process, since the DBMS must ensure that the effects of all transactions that completed prior to the crash are restored, and that the effects of incomplete transactions are undone. The time required to recover from a crash can be reduced by periodically forcing some information to disk; this periodic operation is called a checkpoint [51].

Summarizing, the main takeaways for DBMS support for concurrency control and recovery are [51]:

- Locking: every object that is read or written by a transaction is first locked in shared or exclusive mode, respectively. Placing a lock on an object restricts its availability to other transactions and thereby affects performance.
- WAL: for efficient log maintenance, the DBMS must be able to selectively force a collection of pages in main memory to disk. OS support for this operation is not always satisfactory.
- Periodic checkpoint: can reduce the time needed to recover from a crash. There is a trade-off between speed and system integrity, as checkpointing too often slows down normal execution.

3.11.5. Structure of a RDBMS

Fig. 3.22 shows the structure (with some simplification) of a typical DBMS based on the relational data model.

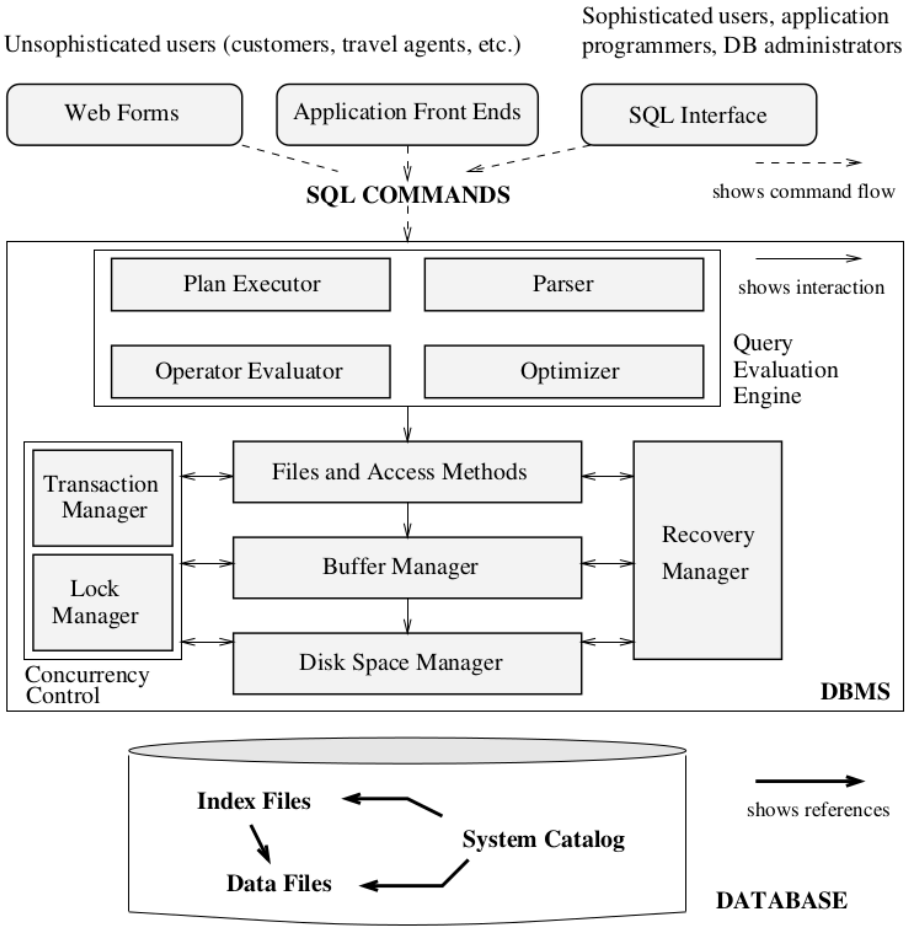


Figure 3.22.: Architecure of a DBMS (withdrawn from [51])

It is comprised of [51]:

- Interfaces: The DBMS accepts SQL commands generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. SQL commands can also be embedded in host-language application programs, e.g., Java or C++ programs, but here one concentrates only on the core DBMS functionality.
- DBMS: contains:
    - Query Evaluation Engine: When a user issues a query, the parsed query is presented to a query optimizer, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An execution plan is a blueprint for evaluating a query, and is usually represented as a tree of relational operators (with annotations that contain additional detailed information about which access methods to use, etc.).
    - Concurrency control: The Transaction Manager ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions. The Lock Manager keeps tracks of request for locks and grants locks on database objects when they become available.
    - Recovery manager: responsible for maintaining a log, and restoring the system to a consistent state after a crash.
    - Disk access manager: The file and access methods layer includes a variety of software for supporting the concept of a file, which, in a DBMS, is a collection of pages or a collection of records. This layer typically supports a heap file, or file of unordered pages, as well as indexes. In addition to keeping track of the pages in a file, this layer organizes the information within a page. The Buffer manager handles pages as a response to read requests. The disk space manager deals with management of space on disk, where the data is stored. Higher layers allocate, deallocate, read, and write pages through (routines provided by) this layer.
- Database: contains the system catalog information consisting of the index files referencing the data files storing the actual data on physical memory.

3.11.6. Database design overview

The database design process can be divided into six steps, namely [51]:

1. Requirements Analysis: the requirements for the database application are elicited and analyzed assessing what data is to be stored in the database, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements. Several methodologies

have been proposed for organizing and presenting the information gathered in this step, and some automated tools have been developed to support this process.

2. Conceptual Database Design: the information gathered in the previous step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. This step is often carried out using the ER model, or a similar high-level data model.
3. Logical Database Design: A DBMS must be chosen to implement the database design, and convert the conceptual database design – ER schema — into a database schema in the data model of the chosen DBMS — relational schema.
4. Schema Refinement: Next, the collection of relations in the relation database schema are analyzed to identify potential problems, and to refine it. This is performed by normalizing relations, restructuring them to ensure some desirable properties.
5. Physical Database Design: In this step, the typical expected workloads that the database must support are analyzed and further refine the database design to ensure that it meets desired performance criteria. This may simply involve building indexes on some tables and clustering some tables, or it may involve a substantial redesign of parts of the database schema obtained from the earlier design steps.
6. Security Design: Lastly, the different user groups and different roles played by various users are identified (e.g., the development team for a product, the customer support representatives, the product manager). For each role and user group, the permitted and forbidden parts of the database are identified and the policies are enforced to ensure this. A DBMS provides several mechanisms to assist in this step.

3.11.7. Entity-Relationship model

The Entity-Relationship (ER) data model enables the description of the data involved in a real-world enterprise in terms of entities and their relationships and is widely used to develop an initial database design. The ER model is most relevant to the first three steps of the database design [51] (see Section 3.11.6). In this section are presented the key concepts for the ER model as a database design modeling tool.

Key concepts

It is important to understand some key concepts for the ER model, namely [51]:

- Entity: is an object in the real world that is distinguishable from other objects, e.g., the Pokemon toy, the toy department, the manager of the toy department, etc.

- Entity set: collection of entities. They do not need to be disjoint, i.e., entities can simultaneously belong to different entity sets. For example, one can define an entity set called **Employees** that contain both the toy and appliance department employee sets. An entity set is represented by a rectangle in the ER model.
- Attributes: an entity is described using a set of attributes. All entities in a given entity set have the same attributes. For example, the **Employees** entity set could use **name**, social security number (**ssn**) and parking lot (**lot**) as attributes. An attribute is represented by an oval in the ER model.
- Domain: for each attribute associated with a set, one must identify a domain of possible values. For example, the domain associated with the attribute **name** of **Employees** might be the set of 20-character strings. The domain information can be listed along the attribute name.
- Key: minimal set of attributes whose values uniquely identify an entity in the set. There could be more than one candidate key: if so, one designate one of them as the primary key. Each attribute in the primary key is underlined in the ER model. The foreign key is(are) the attribute(s) which in a relationship one-to-many is(are) primary key(s) in the entity of the side **one** and integrates the set of attributes of the entity in the side **many** of that relationship
- Relationship: association between two or more entities. For example, one may have the relationship that Joe works in the pharmacy department. A relationship is represented by a straight line in the ER model and may also have descriptive attributes.
- Relationship set: collection of similar relationships.
- Key constraints: restrictions over entities. For example, the restriction that each department has at most one manager, and is denoted by using an arrow from the entity to the relationship.
- Participation constraints: states the participation of an entity set in the relationship. It can be partial or total. For example, if every department is required to have a manager, the participation of the entity set **Departments** in the relationship set **Manages** is said to be total.
- Class hierarchies: classification of entities in an entity set into subclasses, using the relationship is a. For example, a **Car is a Vehicle** and a **Truck is a Vehicle** too.
- Aggregation: indicates that a relationship set (identified through a dashed box) participates in another relationship set.

The ERDs use a graphical conventional to quickly and clearly depict the entities involved and how they relate to each other. In a ERD entities are represented by rectangles, attributes by ellipses, and the relationships as lines between entities. In the rectangles and ellipses are placed the names of the different entities and attributes. The relationships have cardinalities — **1:1** (one-to-one), **1:M** (one-to-many), and **M:N** (many-to-many) — and may be mandatory or optional — e.g., a vehicle may not have any parking space assigned, but to each parking space is assigned one, and one only, vehicle.

Several notations can be used, namely, crow's foot, UML, Chen, Barker, etc [52]. In crow's foot notation:

- A multiplicity of one and a mandatory relationship is represented by a straight line perpendicular to the relationship line.
- A multiplicity of many is represented by the three-pronged 'crow-foot' symbol.
- An optional relationship is represented by an empty circle.

Fig. 3.23 illustrates an example of an Entities-Relationships diagram (ERD) using crow's foot notation for an a company. There are three entity sets — **Customers**, **Orders**, and **Shipments**. Within each of these are the attributes, with the primary key being underlined. Additionally it also indicates the foreign key that resolves the **one-to-many** relationship. Thus, a customer can place **0** or **many** orders, which, in turn, can have **0** or **many** shipment methods.

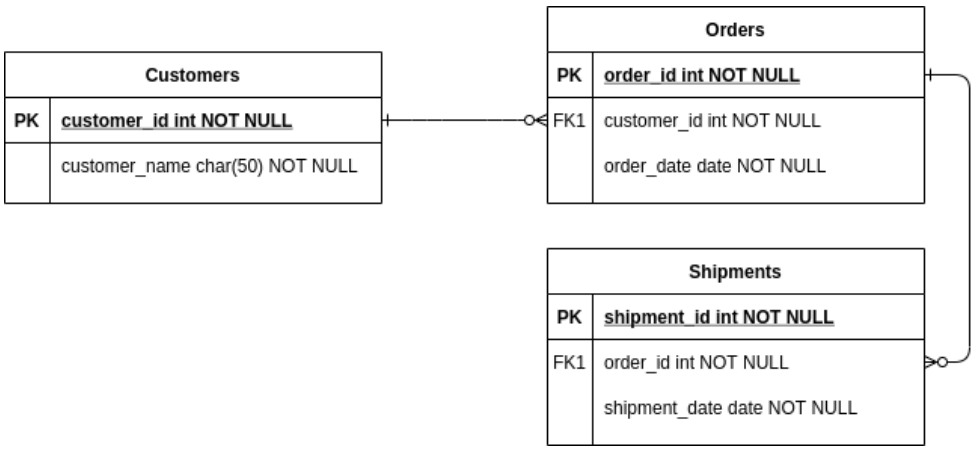


Figure 3.23.: Example of an ERD

3.11.8. Choice of the RDBMS

In this section are presented the most relevant RDBMSs, its advantages and disadvantages, and the best use case, which can help the database designer to appropriately choose a RDBMS. Special focus is given to the free systems, namely **MySQL** and**SQLite**, due to budget restrictions.

The most relevant RDBMSs are [53]:

- **Oracle Database**: Oracle has provided high-quality database solutions since the 1970s. The most recent version of Oracle Database was designed to integrate with cloud-based systems, and it allows you to manage massive databases with billions of records.
    - Advantages: the most advanced technology and a wide range of solutions.

- Disadvantages: an expensive solution and system upgrades might be required — many businesses have to upgrade their hardware before using Oracle solutions.
- Best use case: if you're a large organization that needs to manage a massive amount of data, Oracle could be the ideal choice.

- **Microsoft SQL Server**: it is a database engine that is compatible with, both, on-site and cloud-based servers, and supports Windows and Linux OSes.
  - Advantages: it is mobile: this database engine allows you to access dashboard graphics and visuals via mobile devices. It integrates with Microsoft products. It is fast and stable.
  - Disadvantages: an expensive solution and requires a lot of HW resources.
  - Best use case: if you're an enterprise-level corporation that relies heavily on Microsoft products, the speed, agility, and reliability of Microsoft SQL Server could be an excellent choice.
- **MySQL**: MySQL is a free, open-source RDBMS solution that Oracle owns and manages. Even though it's freeware, MySQL benefits from frequent security and features updates. Large enterprises can upgrade to paid versions of MySQL to benefit from additional features and user support.
  - Advantages: it is open-source, free of charge (freeware) and highly compatible with many other database systems.
  - Disadvantages: lacking features common to other RDBMSs: because MySQL prioritizes speed and agility over features, some of the standard features found in other solutions may be missing, e.g., the ability to create incremental backups. Challenges getting quality support: The free version of MySQL does not come with on-demand support. However, MySQL does have an active volunteer community, useful forums, and a lot of documentation.
  - Best use case: MySQL is a particularly valuable RDBMS solution for businesses that need a solution with enterprise-level capabilities, but are operating under strict budget constraints. It is an extremely powerful and reliable modern RDBMS with a free tier.
- **SQLite** [54]: it is a C-language library that implements a small, fast, self-contained SQL database engine — an embedded DB — which means the DB engine runs as a part of the app.
  - Advantages: it is open-source, free of charge (freeware), a server-less and file-based database, and self-contained. It has a small storage footprint (the **SQLite** library is 250 KB in size, while the **MySQL** server is about 600 MB). It directly stores information into a single file, making it easy to copy, and no configuring is required.
  - Disadvantages: it lacks user management (not suitable for multiple users) and security features (the database can be accessed by anyone). It is not easily scalable and cannot be customized.

- Best use case: **SQLite** is best suited for developing small standalone apps or smaller projects which do not require much scalability.

Summary

Oracle Database and **Microsoft SQL server** are proprietary solutions, specially suited for large organizations, and can be very expensive. On the other hand, **MySQL** and **SQLite** are open-source and freeware solutions, which can be used to quickly test and iterate the DB design before moving into production. However, **SQLite** does not have user authentication or security features, and it is not easily scalable. Thus, **MySQL** arises as the best option, suited for distributed architectures, as a trade-off between cost, ease of use, scalability, security and user management.

3.11.9. SQL

Ideally, a database language allows the creation of a database and table structures, the execution of basic data management tasks (add, delete, and modify), and the execution of complex queries designed to transform the raw data into useful information. Moreover, it must provide a clear and easy suntax, it must be portable and conform to some basic standard. SQL complies well to these requirements [55].

SQL functions fit into two broad categories [55]:

1. It is a Data Definition Language (DDL): SQL includes commands to create database objects such as tables, indexes, and views, as well as commands to define access rights to those database objects (see Fig. 3.24).
2. It is a Data Manipulation Language (DML): SQL includes commands to insert, update, delete, and retrieve data within the database tables (see Fig. 3.25).

3.11.10. MySQL Interfaces

MySQL works under the client–server paradigm. It has several client interfaces that can interact with the server, through connectors and APIs, i.e., the drivers and libraries that one can use to connect applications in different programming languages to **MySQL** database servers. The application and database server can be on the same machine, or communicate across the network [56]. The following interfaces are available: **Java**, **Python**, **JavaScript**, **C++**, **C**, **C#**, **PHP**, **OBDC**, **NBD Cluster**, **MySQL Shell**, and **X DevAPI**.

COMMAND OR OPTION	DESCRIPTION
CREATE SCHEMA AUTHORIZATION	Creates a database schema
CREATE TABLE	Creates a new table in the user's database schema
NOT NULL	Ensures that a column will not have null values
UNIQUE	Ensures that a column will not have duplicate values
PRIMARY KEY	Defines a primary key for a table
FOREIGN KEY	Defines a foreign key for a table
DEFAULT	Defines a default value for a column (when no value is given)
CHECK	Validates data in an attribute
CREATE INDEX	Creates an index for a table
CREATE VIEW	Creates a dynamic subset of rows/columns from one or more tables
ALTER TABLE	Modifies a tables definition (adds, modifies, or deletes attributes or constraints)
CREATE TABLE AS	Creates a new table based on a query in the user's database schema
DROP TABLE	Permanently deletes a table (and its data)
DROP INDEX	Permanently deletes an index
DROP VIEW	Permanently deletes a view

Figure 3.24.: SQL data definition commands — withdrawn from [55]

COMMAND OR OPTION	DESCRIPTION
INSERT	Inserts row(s) into a table
SELECT	Selects attributes from rows in one or more tables or views
WHERE	Restricts the selection of rows based on a conditional expression
GROUP BY	Groups the selected rows based on one or more attributes
HAVING	Restricts the selection of grouped rows based on a condition
ORDER BY	Orders the selected rows based on one or more attributes
UPDATE	Modifies an attribute's values in one or more table's rows
DELETE	Deletes one or more rows from a table
COMMIT	Permanently saves data changes
ROLLBACK	Restores data to their original values
COMPARISON OPERATORS	
=, <, >, <=, >=, <>	Used in conditional expressions
LOGICAL OPERATORS	
AND/OR/NOT	Used in conditional expressions
SPECIAL OPERATORS	Used in conditional expressions
BETWEEN	Checks whether an attribute value is within a range
IS NULL	Checks whether an attribute value is null
LIKE	Checks whether an attribute value matches a given string pattern
IN	Checks whether an attribute value matches any value within a value list
EXISTS	Checks whether a subquery returns any rows
DISTINCT	Limits values to unique values
AGGREGATE FUNCTIONS	Used with SELECT to return mathematical summaries on columns
COUNT	Returns the number of rows with non-null values for a given column
MIN	Returns the minimum attribute value found in a given column
MAX	Returns the maximum attribute value found in a given column
SUM	Returns the sum of all values for a given column
AVG	Returns the average of all values for a given column

Figure 3.25.: SQL data manipulation commands — withdrawn from [55]

C++ connector

From the list of available interfaces, the most well suited to interface the RDBMS are the **C API** and the **C++ connector**, as they are the most well known programming languages by the authors and the better ones in terms of performance. However, the **C++ connector** was chosen because it offers the following benefits over the MySQL C API provided by the MySQL client library [57]:

- Convenience of pure C++.
- Support for the object-oriented programming paradigm.
- Support for these application programming interfaces: X DevAPI, X DevAPI for C, Legacy JDBC 4.0-based API.
- Reduced development time.
- Licensed under the GPL with the FLOSS License Exception.
- Available under a commercial license upon request.

Listing 3.5 illustrates an application to connect to the **MySQL** server using the **C++** connector [58]. Starting from the line 24 the application tries to connect to the **MySQL** database and execute a query and, if any exception is thrown, it is caught by the **catch** block at line 50. Looking into more detail at the **try** block, the pointers to objects **sql::Driver**, **sql::Connection**, **sql::Statement** and **sql::ResultSet** are instantiated for later usage. A connection to the **MySQL server** is established at line 32 — via **driver** — using the **TCP/IP** protocol at **localhost** address and port **3306** (the default for **MySQL server**), and passing the username and the password. Next, the specific database — **test** — is defined for access.

Then, at line 36, a statement is created to execute a query (line 37), with the result being captured by **res** (the **sql::ResultSet** object). A loop is performed to retrieve all data, and lines 41 and 44 illustrate different methods to access the column data — by column name or by numeric offset, respectively. Finally, all objects dynamically allocated are released (lines 46-48) and, if everything works well, the application returns successfully (line 61).

```
/* Standard C++ includes */
#include <stdlib.h>
#include <iostream>
/*
  Include directly the different
  headers from cppconn/ and mysql_driver.h + mysql_util.h
  (and mysql_connection.h). This will reduce your build time!
*/
#include "mysql_connection.h"

#include <cppconn/driver.h>
#include <cppconn/exception.h>
```

```
#include <cppconn/resultset.h>
#include <cppconn/statement.h>

using std::cout;
using std::endl;

int main(void)
{
    cout << endl;
    cout << "Running 'SELECT 'Hello World!' » AS _message'..." << endl;

    try {
        sql::Driver *driver;
        sql::Connection *con;
        sql::Statement *stmt;
        sql::ResultSet *res;

        /* Create a connection */
        driver = get_driver_instance();
        con = driver->connect("tcp://127.0.0.1:3306", "root", "root");
        /* Connect to the MySQL test database */
        con->setSchema("test");

        stmt = con->createStatement();
        res = stmt->executeQuery("SELECT 'Hello World!' AS _message");
        while (res->next()) {
            cout << "\t... MySQL replies: ";
            /* Access column data by alias or column name */
            cout << res->getString("_message") << endl;
            cout << "\t... MySQL says it again: ";
            /* Access column data by numeric offset, 1 is the first column */
            cout << res->getString(1) << endl;
        }
        delete res;
        delete stmt;
        delete con;

    } catch (sql::SQLException &e) {
        cout << "# ERR: SQLException in " << __FILE__;
        cout << "(" << __FUNCTION__ << ") on line " »
             << __LINE__ << endl;
        cout << "# ERR: " << e.what();
        cout << " (MySQL error code: " << e.getErrorCode();
```

```
56          cout << ", SQLState: " << e.getSQLState() << " )" << endl;
    }

    cout << endl;

    return EXIT_SUCCESS;
62 }
```

Listing 3.5: MySQL Connector example — adapted from [58]

3.12. Motion detection

The **MDO-L** needs to be capable of detecting a user, in order to switch between **normal mode** to **interaction mode**. In order to do such thing, it is mandatory to have a system that detects the motion of a user that approaches to the local system **MDO-L**.

Different types of detection

There are several ways to do motion detection, through different sensors, such as [59]:

- Passive Infrared (PIR): A passive infrared sensor detects body heat (infrared energy) by looking for changes in temperatures. This is the most-widely-used motion sensor in home security systems [59].

  Once the PIR motion sensor warms up, it can detect heat and movement in the surrounding areas, creating a protective "grid". If a moving object blocks too many grid zones and the infrared energy levels change rapidly, the infrared sensor triggers an alarm [59].

- Mircowave (MW): This type of sensor sends out microwave pulses and measures the reflections off of moving objects. They cover a larger area than infrared sensors but are more expensive and vulnerable to electrical interference [59].

- Dual technology motion sensors: Some motion sensors can combine multiple detection methods in an attempt to reduce false alarms. For example, it's not uncommon for a dual technology sensor to combine a PIR sensor with a MW sensor [59].

- Less common types of motion detectors:

- Area reflective sensors: This kind of sensors emit infrared rays from an LED and use the reflection of those rays to measure the distance to the person or object, allowing for detection when the subject moves within the designated area [59].
- Ultrasonic motion sensors: This sensors measure the reflections off of moving objects via pulses of ultrasonic waves [59].
- Vibration motion sensors: This type of sensors detect small vibrations that people cause when they move through a room [59].

- Specialized motion sensors:
  - Contact sensors: Contact sensors use a magnet to spot movement on a door or window. When the sensor and corresponding magnet move apart as a door or window opens, the sensor triggers [59].
  - Pet-immune motion sensors: Most passive infrared sensors can ignore animals up to a certain weight. A dual technology motion sensor is more pet resistant to false alarms because it requires two sensors to be triggered a certain way [59].

Trade-off between sensors

As it can be seen previously, there are several types of sensors that can be used. However, it is mandatory to make a trade-off between each type in order to choose the better case, according several factors such as price, range, applicability, conditions of use and others.

For this specific case, it is necessary a set of sensors with a low range detection (more or less than 1,5 meters), a good reflection and low consumption. Thus, the type of sensors that match all these criteria are ultrasonic sensors. The reasons to discard the others sensors are simple:

Firstly, the PIR sensor actuates with changes in radiation, which means that some other case can occur where is not a human approaching the MDO-L like, for example, an animal.

In second place, the Infrared Reflective sensors have too specific characteristics of behavior. For example, there are some surfaces that do not reflect quite good the infrared (IR) radiation, which could easily result on a bad behavior for two different users.

Lastly, the contact sensors are out of question for obvious reasons: there is no need to have a contact between the MDO-L and an user.

Concluding, the best option is to use a set of ultrasonic sensors, in order to avoid some specific situations and make the user detection as better as possible.

Ultrasonic sensor

Ultrasonic sensors work by sending out a sound wave at a frequency above the range of human hearing. Ultrasonic waves are sound waves emitted at a frequency higher than can be detected by human hearing — typically above 20 kHz [60].

As it can be seen in Fig. 3.26, many ultrasonic sensors use a single transducer to send a pulse and to receive the echo, others use one transducer to send the pulse and another to receive it. The sensor determines the distance to a target by measuring time lapses between the sending and receiving of the ultrasonic pulse [61].

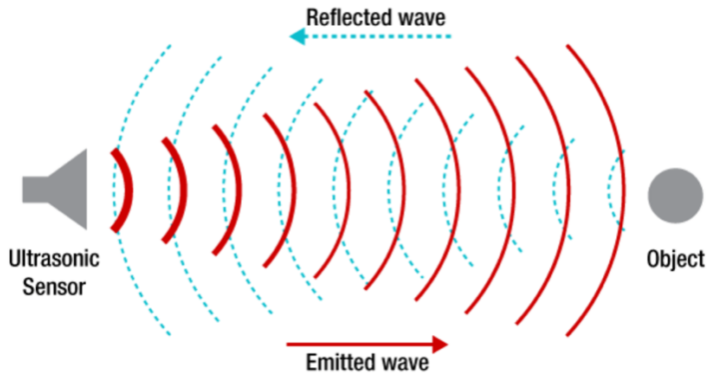


Figure 3.26.: Example of the behaviour of an ultrasonic sensor(withdrawn from [60])

3.13. Camera recording and codecs

aaaaaaaa

3.14. Image filtering

aaaaaaaa

3.15. GIF generation

aaaaaaaa

3.16. Social media sharing APIs

aaaaaaaa

3.17. UI framework

For the development of the **Local System** and the **Remote Client**, it is necessary to use a UI Framework, in order to develop a UI, making it more user friendly and interactive. There are several frameworks that can be used in Linux, such as [62]:

- Qt;
- Sciter;
- Noesis GUI;
- wxWidgets;
- GTK+;
- and so on.

For this project, **Qt** was chosen due to the following reasons:

- Cross development: it is possible to 'develop graphical user interfaces and cross-platform applications, both desktop and embedded' [63]. The framework operates on different types of software and hardware.
- Cost: this framework is **cost-friendly**, not only because it has a free license, but also because its software development takes less time to develop due to the integrated environment assisting the developer.
- Implementation: it is implemented in C++, which means that it is possible to use many libraries. The wide choice of modules allow the project to have rich functionality and as a result, the software will have a Graphical User Interface (GUI) similar to a native one.

3.17.1. Qt

The Qt framework contains a comprehensive set of highly intuitive and modularized C++ library classes and is loaded with APIs to simplify application development [64].

Signals and slots

In Qt, there's an alternative to the callback technique, using **signals and slots**. A **signal** is emitted when a particular event occurs and a **slot** is a function that is called in response to a particular signal [65].

- Signals: Are emitted by an object when its internal state has changed in some way that might be interesting to the object's client or owner. Signals are also public access functions and can be emitted from anywhere [65].
- Slots: A slot is called when a signal connected to it is emitted. Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them [65].

Compared to callbacks, signals and slots are slightly slower because of the increased flexibility they provide, although the difference for real applications is insignificant [65].

Usage Example

In Fig.3.27 is an example on how Qt can be used and what it can generate:

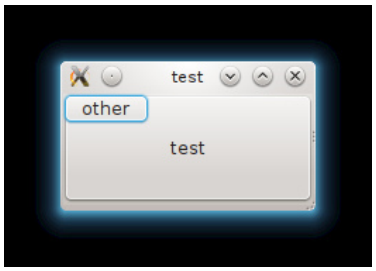


Figure 3.27.: Usage example of Qt(withdrawn from [66])

The code that generates this UI is the following:

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char **argv)
{
  // Application instanciation
  QApplication app (argc, argv);

  // Creating the two buttons
  QPushButton button1 ("test");
  QPushButton button2 ("other", &button1);

  // Show button
  button1.show();
```

```
16    // Exectution in loop of the application,
      // waiting for its events to trigger the functions
18    return app.exec();
  }
```

Listing 3.6: Implementing a simple window in Qt

Configuration with buildroot

The configuration of Qt in buildroot to run on Raspberry Pi is pretty simple. In the $menuconfig$, it is just needed to select **Target packages**, then select **Graphic libraries and applications (graphic/text)** and finally select **Qt5**.

Then on the Qt5 menu select the following options presented on Fig. 3.28.

```
--- Qt5
      Qt5 version (Latest (5.8))  --->
      *** qt53d module needs an OpenGL-capable backend ***
-*-   qt5base
()      Custom configuration options
[*]     Approve free license
()      Config file
[*]     Compile and install examples (with code)
[ ]     concurrent module
[ ]     MySQL Plugin
[ ]     PostgreSQL Plugin
        SQLite 3 support (No sqlite support)  --->
[*]     gui module
[*]       Widgets module
          *** OpenGL support needs an OpenGL-capable backend ***
-*-       linuxfb support
          *** directfb backend available if directfb is enabled ***
          *** X.org XCB backend available if X.org is enabled ***
          *** eglfs backend available if OpenGL and EGL are enabled ***
()        Default graphical platform
[ ]       fontconfig support
[*]       GIF support
[*]       JPEG support
[*]       PNG support
[ ]     DBus module
[*]     Enable ICU support
[*]     Enable Tslib support
```

Figure 3.28.: Selection of qt package in buildroot(withdrawn from [67])

In conclusion, **Qt** is a great option to use on the project due to its features and also because it is very simple to use it on the project's board (Raspberry Pi).

3.18. File transfer protocols

aaaaaaaa

Bibliography

[1] Fei Tao, Meng Zhang, and A.Y.C. Nee. Chapter 12 - digital twin, cyber–physical system, and internet of things. In Fei Tao, Meng Zhang, and A.Y.C. Nee, editors, Digital Twin Driven Smart Manufacturing, pages 243–256. Academic Press, 2019. ISBN 978-0-12-817630-6. doi: https://doi.org/10.1016/B978-0-12-817630-6.00012-6. URL https://www.sciencedirect.com/science/article/pii/B9780128176306000126.

[2] What the nose knows. URL https://news.harvard.edu/gazette/story/2020/02/how-scent-emotion-and-memory-are-intertwined-and-exploited/. accessed: 2021-10-23.

[3] Martin Lindstrom. Brand sense: How to build powerful brands through touch, taste, smell, sight and sound. Strategic Direction, 2006.

[4] Digital outdoor advertising: The what, the why, the how. URL https://bubbleoutdoor.com/digital-outdoor-advertising-what-why-how/. accessed: 2021-10-24.

[5] Digital outdoor market to be worth $55 BN in 5 years. URL https://www.decisionmarketing.co.uk/news/digital-outdoor-market-to-be-worth-55bn-in-5-years. accessed: 2021-10-24.

[6] Scent marketing; type of sensory marketing targeted at the olfactory sense, . URL https://www.air-aroma.com/scent-marketing/. accessed: 2021-10-24.

[7] Scent basics: What is scent marketing, scent branding and ambient scent, . URL https://reedpacificmedia.com/scent-basics-what-is-scent-marketing-scent-branding-and-ambient-scent/. accessed: 2021-10-24.

[8] Digital scent technology market with covid-19 impact analysis, . URL https://www.marketsandmarkets.com/Market-Reports/digital-scent-technology-market-118670062.html. accessed: 2021-10-24.

[9] Ian Sommerville. Software process models. ACM computing surveys (CSUR), 28(1):269–271, 1996.

[10] Michael A Cusumano and Stanley A Smith. Beyond the waterfall: Software development at microsoft. 1995.

[11] Bernd Bruegge and Allen H Dutoit. Object-Oriented Software Engineering Using UML, Patterns and Java-(Required), volume 2004. Prentice Hall, 2004.

[12] DICK AUTOR BUTTLAR, Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. Pthreads programming: A POSIX standard for better multiprocessing. " O'Reilly Media, Inc.", 1996.

[13] Michael Kerrisk. The Linux programming interface: a Linux and UNIX system programming handbook. No Starch Press, 2010.

[14] E Bryan Carne. A professional's guide to data communication in a TCP/IP world. Artech House, 2004.

[15] Gary R Wright and W Richard Stevens. Tcp/ip illustrated. vol. 2: The implementation. tii, 1995.

[16] Albert S Huang and Larry Rudolph. Bluetooth essentials for programmers. Cambridge University Press, 2007.

[17] M David Hanson. The client/server architecture. In Server Management, pages 17–28. Auerbach Publications, 2000.

[18] Client/server model. URL https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.1.0/com.ibm.cics.tx.doc/concepts/c_clnt_sevr_model.html. accessed: 2020-07-02.

[19] Prof. Adriano Tavares & Prof. Mongkol. Daemons class slides, 2021.

[20] Device drivers: Why they're important and how to work with them. URL https://www.lifewire.com/what-is-a-device-driver-2625796. accessed: 2021-12-1.

[21] Prof. Adriano Tavares & Prof. Mongkol. Linux + ddrivers class slides, 2021.

[22] Richard M Stallman and Roland McGrath. GNU Make: A Program for Directing Recompilation: GNU Make Version 3.79. 1. Free software foundation, 2002.

[23] Peter Baker. Using gnu make to manage the workflow of data analysis projects. Journal of Statistical Software, 94(1):1–46, 2020.

[24] Eric S Raymond. The art of Unix programming. Addison-Wesley Professional, 2003.

[25] Arnold Robbins. Unix in a Nutshell. "O'Reilly Media, Inc.", 2005.

[26] An overview over build systems (mostly over c++ projects), 2018. URL https://julienjorge.medium.com/an-overview-of-build-systems-mostly-for-c-projects-ac9931494444. accessed: 2021-12-1.

[27] Gnu automake manual, 2021. URL https://www.gnu.org/software/automake/manual/automake.html. accessed: 2021-12-1.

[28] Adrian Kaehler and Gary Bradski. Learning OpenCV 3: computer vision in C++ with the OpenCV library. " O'Reilly Media, Inc.", 2016.

[29] Top 10 computer vision frameworks you need to know in 2021. URL https://roboticsbiz.com/top-10-computer-vision-frameworks-you-need-to-know-in-2020/. accessed: 2021-11-21.

[30] Shuo Yang, Ping Luo, Chen-Change Loy, and Xiaoou Tang. Wider face: A face detection benchmark. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 5525–5533, 2016.

[31] Ming-Hsuan Yang, David J Kriegman, and Narendra Ahuja. Detecting faces in images: A survey. IEEE Transactions on pattern analysis and machine intelligence, 24(1):34–58, 2002.

[32] Cha Zhang and Zhengyou Zhang. A survey of recent advances in face detection. 2010.

[33] Paul Viola and Michael J Jones. Robust real-time face detection. International journal of computer vision, 57(2):137–154, 2004.

[34] Jianguo Li and Yimin Zhang. Learning surf cascade for fast and accurate object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3468–3475, 2013.

[35] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In 2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05), volume 1, pages 886–893. Ieee, 2005.

[36] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. IEEE transactions on pattern analysis and machine intelligence, 32(9):1627–1645, 2009.

[37] Piotr Dollár, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: A benchmark. In 2009 IEEE Conference on Computer Vision and Pattern Recognition, pages 304–311. IEEE, 2009.

[38] Bin Yang, Junjie Yan, Zhen Lei, and Stan Z Li. Aggregate channel features for multi-view face detection. In IEEE international joint conference on biometrics, pages 1–8. IEEE, 2014.

[39] Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, and Gang Hua. A convolutional neural network cascade for face detection. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 5325–5334, 2015.

[40] Shuo Yang, Ping Luo, Chen-Change Loy, and Xiaoou Tang. From facial parts responses to face detection: A deep learning approach. In Proceedings of the IEEE international conference on computer vision, pages 3676–3684, 2015.

[41] Face detection tips, suggestions and best-practices. URL https://www.pyimagesearch.com/2021/04/26/face-detection-tips-suggestions-and-best-practices/. accessed: 2021-11-22.

[42] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001, volume 1, pages I–I. Ieee, 2001.

[43] Cascade classifier. URL https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html. accessed: 2021-11-22.

[44] Haar cascades, explained. URL https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html. accessed: 2021-11-22.

[45] H Aashni, S Archanasri, A Nivedhitha, P Shristi, and SN Jyothi. International conference on advances in computing & communications. Science Direct, 115:367–374, 2017.

[46] Mais Yasen and Shaidah Jusoh. A systematic review on hand gesture recognition techniques, challenges and applications. PeerJ Computer Science, 5:e218, 2019.

[47] SG Vaibhavi, AK Akshay, NR Sanket, AT Vaishali, and SS Shabnam. A review of various gesture recognition techniques. International Journal of Engineering and Computer Science, 3:8202–8206, 2014.

[48] Ananya Choudhury, Anjan Kumar Talukdar, and Kandarpa Kumar Sarma. A review on vision-based hand gesture recognition and applications. In Intelligent applications for heterogeneous system modeling and design, pages 256–281. IGI Global, 2015.

[49] Taining a neural network to detect gestures with opencv in python. URL https://towardsdatascience.com/training-a-neural-network-to-detect-gestures-with-opencv-in-python-e09b0a12bdf1. accessed: 2021-11-28.

[50] Munir Oudah, Ali Al-Naji, and Javaan Chahl. Hand gesture recognition based on computer vision: a review of techniques. journal of Imaging, 6(8):73, 2020.

[51] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. Database management systems, volume 3. McGraw-Hill New York, 2003.

[52] Crow's foot notation. URL https://vertabelo.com/blog/crow-s-foot-notation/. accessed: 2021-11-29.

[53] Which modern database is right for your use case? URL https://www.xplenty.com/blog/which-database/. accessed: 2021-11-30.

[54] Sqlite vs mysql – what's the difference, . URL https://www.hostinger.com/tutorials/sqlite-vs-mysql-whats-the-difference/. accessed: 2021-11-30.

[55] Carlos Coronel and Steven Morris. Database systems: design, implementation, & management. Cengage Learning, 2016.

[56] Mysql connectors and apis, . URL https://dev.mysql.com/doc/index-connectors.html. accessed: 2021-11-30.

[57] Mysql c++ connector: Intro, . URL https://dev.mysql.com/doc/connector-cpp/8.0/en/connector-cpp-introduction.html#connector-cpp-benefits. accessed: 2021-11-30.

[58] Mysql c++ connector: Example 1, . URL https://dev.mysql.com/doc/connector-cpp/1.1/en/connector-cpp-examples-complete-example-1.html. accessed: 2021-11-31.

[59] The beginner's guide to motion sensors. URL https://www.safewise.com/resources/motion-sensor-guide/. accessed: 2021-12-6.

[60] Understanding how ultrasonic sensors work, . URL https://www.linearmotiontips.com/ultrasonic-sensors-for-linear-position-and-distance-measuring/. accessed: 2021-12-6.

[61] Understanding how ultrasonic sensors work, . URL https://www.maxbotix.com/articles/how-ultrasonic-sensors-work.htm. accessed: 2021-12-6.

[62] C++ ui libraries. URL https://philippegroarke.com/posts/2018/c++_ui_solutions/. accessed: 2021-12-2.

[63] Qt framework and qml, . URL https://www.sam-solutions.com/blog/qt-framework/. accessed: 2021-12-1.

[64] The building blocks of qt, . URL https://www.qt.io/product/frameworkhttps://www.qt.io/product/framework. accessed: 2021-12-2.

[65] Signals & slots | qt core 5.15.7, . URL https://doc.qt.io/qt-5/signalsandslots.html. accessed: 2021-12-2.

[66] Coffee machine example, . URL https://doc.qt.io/qt-5/qtdoc-demos-coffee-example.html. accessed: 2021-12-2.

[67] Embedded linux – working with qt, . URL https://devarea.com/embedded-linux-working-with-qt/#.Ya3vmWjP0kI. accessed: 2021-12-6.

Appendices

A. Project Planning — Gantt diagram

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

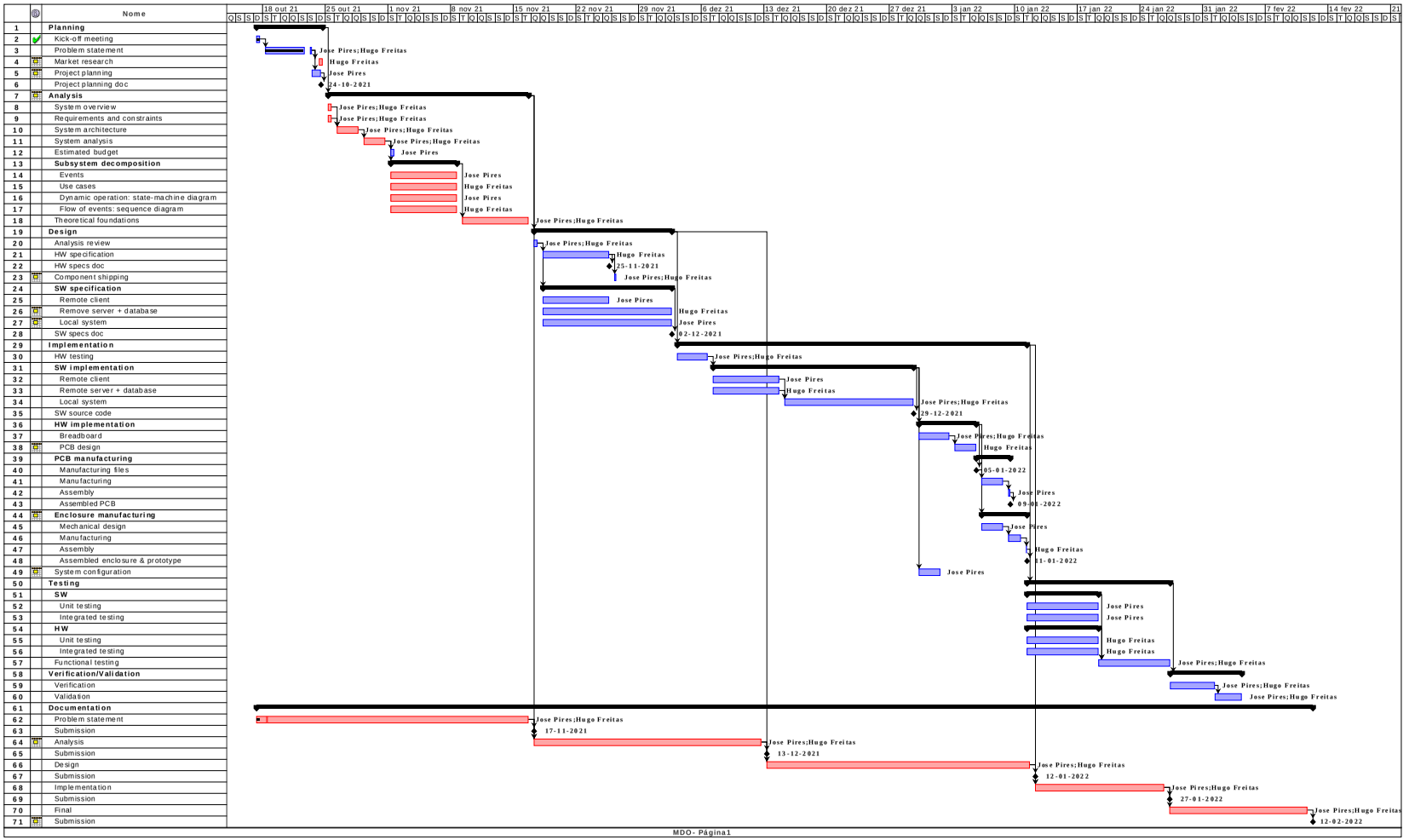


Figure A.1.: Project planning — Gantt diagram