

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

February 7, 2022

# Contents

<b>Contents</b>	i
<b>List of Figures</b>	vi
<b>List of Tables</b>	xi
<b>List of Listings</b>	xii
<b>List of Abbreviations</b>	xiv
<b>1 Introduction</b>	1
1.1 Context and motivation	1
1.2 Problem statement	2
1.3 Market research	2
1.4 Project goals	3
1.5 Project planning	4
1.6 Report Outline	5
<b>2 Analysis</b>	7
2.1 Requirements and Constraints	7
2.1.1 Functional requirements	7
2.1.2 Non-functional requirements	8
2.1.3 Technical constraints	8
2.1.4 Non-technical constraints	8
2.2 System overview	8
2.2.1 MDO Remote Client	9
2.2.2 MDO Remote Server	9
2.2.3 MDO Local system	10
2.3 System architecture	11

## Contents

---

2.3.1	Hardware architecture	11
2.3.2	Software architecture	12
2.4	Subsystem decomposition	15
2.4.1	Remote Client	16
2.4.2	Remote server	27
2.4.3	Local system	38
2.5	Budget estimation	52
<b>3</b>	<b>Theoretical foundations</b>	<b>56</b>
3.1	Project methodology	56
3.1.1	Waterfall model	56
3.1.2	Unified Modeling Language (UML)	57
3.2	Concurrency	58
3.3	Threads versus Processes	59
3.3.1	Pthreads API	60
3.4	Communications	62
3.4.1	IEEE 802.11 – Wi-Fi	62
3.4.2	Network programming – sockets	63
3.4.3	Client/server model	64
3.5	Daemons	66
3.5.1	What is a Daemon?	66
3.5.2	How to create a Daemon	67
3.5.3	How to handle errors	69
3.5.4	Communicating with a Daemon	70
3.6	Device drivers	70
3.6.1	Kernel mode vs Application	71
3.7	Build system and Makefiles	72
3.7.1	Makefile syntax and example	73
3.7.2	Other build systems for C/C++	78
3.8	Source code documentation	78
3.8.1	Doxygen	79
3.9	Scenting technologies	95
3.9.1	Overview	96
3.9.2	Ultrasonic diffusion	99
3.10	Computer vision	103

## Contents

---

3.10.1	Computer vision frameworks	104
3.10.2	Face detection	106
3.10.3	Hand gesture recognition	109
3.11	RDBMS	111
3.11.1	Description and storage of data in a DBMS	112
3.11.2	Relational model	113
3.11.3	Levels of abstraction in a DBMS	113
3.11.4	Transaction management	114
3.11.5	Structure of a RDBMS	115
3.11.6	Database design overview	117
3.11.7	Entity-Relationship model	118
3.11.8	Choice of the RDBMS	120
3.11.9	SQL	121
3.11.10	MySQL Interfaces	122
3.12	Motion detection	125
3.12.1	Different types of detection	125
3.12.2	Trade-off between sensors	126
3.12.3	Ultrasonic sensor	127
3.13	Camera recording and codecs	128
3.13.1	Camera recording	128
3.13.2	Video files encoding (codecs) and formats	129
3.13.3	Video players for Raspberry Pi	131
3.13.4	Playing video in C++	132
3.14	Image filtering	134
3.14.1	Using filters with OpenCV	135
3.15	GIF generation	137
3.15.1	C/C++ libraries and APIs	138
3.16	Social media sharing APIs	139
3.16.1	Twitter API	141
3.17	UI framework	146
3.17.1	Qt	146
3.18	File transfer protocols	148
3.18.1	Protocols Overview	149
3.18.2	Which protocol is more efficient?	149

## Contents

---

3.18.3 Example of how to transfer files	150
3.18.4 File Transfer Protocol Application Programming Interface (API)s	150
<b>4 Design</b>	<b>155</b>
4.1 Hardware specification	155
4.1.1 Architecture	155
4.1.2 Main Controller	155
4.1.3 Motion Detection	157
4.1.4 Fragrance Diffusion Actuator	158
4.1.5 Camera	159
4.1.6 LCD Display	160
4.1.7 Speakers	160
4.1.8 Power Supply	161
4.1.9 On/Off button	162
4.1.10 Total Hardware (HW) cost	162
4.2 Hardware interfaces definition	163
4.2.1 Peripherals Mapping	163
4.2.2 Test Cases	164
4.2.3 Printed Circuit Board (PCB) Design	164
4.3 Mechanical structure	167
4.4 Software specification	170
4.4.1 Software architecture	170
4.4.2 Deployment specification	175
4.4.3 Database design	176
4.4.4 Data formats	177
4.4.5 Static architecture – Class diagrams	179
4.4.6 Threads specification	183
4.4.7 Flowcharts	186
4.4.8 Test cases	199
4.4.9 Commercial off-the-shelf (COTS) & third-party libraries	201
<b>5 Implementation</b>	<b>209</b>
5.1 Hardware	209
5.2 Remote Client	209
5.2.1 Classes	209

## Contents

---

5.3	Remote Server	228
5.3.1	Data Base	229
5.3.2	Threads	232
5.4	Local System	236
5.4.1	Buildroot configuration	236
5.4.2	Automated build system for Local system software	237
5.4.3	System initialization	241
5.4.4	Device drivers	242
5.4.5	Daemons	250
5.4.6	Classes	253
5.4.7	Threads	273
5.5	Deployment	279
<b>6</b>	<b>Testing</b>	<b>280</b>
6.1	Remote Client	280
6.1.1	Layout	280
6.1.2	Logic	280
6.2	Remote Server	282
6.3	Local System	283
6.3.1	Hardware Tests	283
6.4	Software	287
6.4.1	Local System	287
<b>7</b>	<b>Conclusion</b>	<b>292</b>
7.1	Conclusions	292
7.2	Prospect for Future Work	293
<b>Bibliography</b>		<b>294</b>
<b>Appendices</b>		<b>303</b>
<b>A Project Planning — Gantt diagram</b>		<b>304</b>

# List of Figures

1.1	Example of a Digital Outdoor, withdrawn from [4]	3
1.2	Attractive Opportunities in the Digital Scent Technology Market, withdrawn from [8]	4
2.1	Marketing Digital Outdoor (MDO) system overview	9
2.2	HW architecture diagram	11
2.3	Software (SW) architecture diagram: remote client	13
2.4	SW architecture diagram: remote server	14
2.5	SW architecture diagram: local system	16
2.6	User mockups: remote client	17
2.7	Use cases: remote client	20
2.8	State Machine Diagram: remote client	21
2.9	State Machine Diagram: remote client – Communication Manager	22
2.10	State Machine Diagram: remote client – App Manager	23
2.11	Sequence Diagram: remote client – Login	24
2.12	Sequence Diagram: remote client – admin statistics	25
2.13	Sequence Diagram: remote client – admin users	26
2.14	Sequence Diagram: remote client – admin ads to activate	27
2.15	Sequence Diagram: remote client – admin test operation	28
2.16	Sequence Diagram: remote client – admin logout	29
2.17	Sequence Diagram: remote client - brand	30
2.18	User mock-ups: Remote Server	31
2.19	Use cases: remote server	32
2.20	State machine diagram: Remote server	33
2.21	State machine diagram: Remote Server – Comm Manager	34
2.22	State machine diagram: Remote Server – Database (DB) Manager	35
2.23	State machine diagram: Remote Server – Request Handler	35
2.24	Sequence diagram: Remote Server	36
2.25	Sequence diagram: Remote Server – Authentication	37

---

## List of Figures

---

2.26 Sequence diagram: Remote Server – Manage Databases	37
2.27 Sequence diagram: Remote Server – Test Operation	38
2.28 Sequence diagram: Remote Server – Test Operation Camera	39
2.29 Sequence diagram: Remote Server – Test Operation Share	40
2.30 Sequence diagram: Remote Server – logout	40
2.31 User mock-ups: local system	41
2.32 Use cases diagram: local system	43
2.33 State machine diagram: local system	44
2.34 State machine diagram: local system – Comm Manager	45
2.35 State machine diagram: local system – DB Manager	46
2.36 State machine diagram: local system – Supervisor	48
2.37 Sequence diagram: local system – Normal mode	49
2.38 Sequence diagram: local system – Interaction mode	50
2.39 Sequence diagram: local system – Multimedia mode (select image filter)	51
2.40 Sequence diagram: local system – Multimedia mode (take picture)	52
2.41 Sequence diagram: local system – Multimedia mode (create Graphics Interchange Format (GIF))	52
2.42 Sequence diagram: local system – Sharing mode	55
3.1 Waterfall model diagram	57
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])	59
3.3 Open Systems Interconnection (OSI) model	63
3.4 Steps to obtain a connected socket (withdrawn from [16])	65
3.5 Overview of UNIX system calls with sockets implementing a server/client paradigm (withdrawn from [13])	66
3.6 Example of the usage of a device driver (withdrawn from [21])	70
3.7 Application accessing to module services (withdrawn from [21])	71
3.8 Examples of drivers event and corresponding interface functions between kernel space and user space (withdrawn from [21])	72
3.9 Doxygen output: readme file	96
3.10 Doxygen output: data structures' overview	97
3.11 Doxygen output: collaboration graph for a structure	98
3.12 Doxygen output: header file – dependency graph and <code>typedef</code>	99
3.13 Doxygen output: list of public prototypes for <code>Activity</code> 's module	100

---

**List of Figures**

---

3.14 Doxygen output: implementation file – function details	101
3.15 Micro-porous mesh piezoelectric transducer for ultrasonic diffusion – withdrawn from [31]	102
3.16 Micro-porous mesh piezoelectric-based scent dispense module – withdrawn from [33]	103
3.17 Commercial PCB for driving micro-porous mesh piezoelectric transducers [34]	103
3.18 Schematic of a driving circuit for micro-porous mesh piezoelectric transducers [35]	104
3.19 Examples of Haar features (withdrawn from [51])	108
3.20 Integral image creation illustration (withdrawn from [52])	108
3.21 Illustration of a boosting algorithm (withdrawn from [52])	109
3.22 Flowchart of a cascade classifiers (withdrawn from [52])	109
3.23 Basic steps of hand gesture recognition (withdrawn from [54])	110
3.24 Hand gesture recognition workflow illustrated: example – adapted from [57] and [58]	111
3.25 Levels of abstraction in a DBMS (withdrawn from [59])	114
3.26 Architecure of a DBMS (withdrawn from [59])	116
3.27 Example of an Entities-Relationships diagram (ERD)	119
3.28 SQL data definition commands – withdrawn from [63]	122
3.29 SQL data manipulation commands – withdrawn from [63]	123
3.30 Example of the behaviour of an ultrasonic sensor(withdrawn from [68])	127
3.31 Example of Snapchat like filter(withdrawn from [74])	135
3.32 Usage example of Qt(withdrawn from [100])	147
3.33 Selection of qt package in buildroot(withdrawn from [101])	148
3.34 Configuring HTTP Connection - 1(withdrawn from [103])	151
3.35 Configuring HTTP Connection - 2(withdrawn from [103])	152
3.36 Downloading a File using HTTP - 1(withdrawn from [103])	153
3.37 Downloading a File using HTTP - 2(withdrawn from [103])	153
4.1 HW architecture Block Diagram	156
4.2 Raspberry Pi model 4B	157
4.3 HC-SR04 Pinout (withdrawn from [106])	158
4.4 Fragrance module (withdrawn from [107])	159
4.5 Camera module (withdrawn from [108])	159
4.6 Display (withdrawn from [109])	160
4.7 Display Interfaces (withdrawn from [109])	161
4.8 Speakers (withdrawn from [111])	161
4.9 Power Supply (withdrawn from [112])	162
4.10 Power Supply (withdrawn from [113])	163

---

## List of Figures

---

4.11 Peripheral Mapping	164
4.12 PCB schematic	165
4.13 PCB layout	167
4.14 Mechanical design: Enclosure perspective	168
4.15 Mechanical design: Hardware fixation panel	169
4.16 Mechanical design: Top panel	170
4.17 SW architecture: component diagram – Remote Client	172
4.18 SW architecture: component diagram – Remote Server	172
4.19 SW architecture: component diagram – Local system	174
4.20 Deployment diagram	176
4.21 MDO Entity-Relationship Diagram: conceptual design	178
4.22 Data formats	179
4.23 Class diagram: Remote Client	180
4.24 Class diagram: Remote Server	181
4.25 Class diagram: Local System – part 1/2	182
4.26 Class diagram: Local System – part 2/2	183
4.27 Thread specification: Remote Server	184
4.28 Thread specification: Local System	186
4.29 Flowchart: Remote Client – Login	187
4.30 Flowchart: Remote Client – Register	188
4.31 Flowchart: Remote Client – Delete User	189
4.32 Flowchart: Remote Client – Rent Ad	190
4.33 Flowchart: Remote Server – Receive Thread	191
4.34 Flowchart: Remote Server – Parser	192
4.35 Flowchart: Remote Server – handleQuery	193
4.36 Flowchart: Remote Server – handleCmd	194
4.37 Flowchart: Remote Server – updateLocalSys	195
4.38 Flowchart: Local System – UserDetect thread	196
4.39 Flowchart: Local System – FrameGrabber thread	196
4.40 Flowchart: Local System – LocalRx thread	197
4.41 Flowchart: Local System – LocalTx thread	198
4.42 Flowchart: Local System – AppParser thread	199
4.43 Flowchart: Local System – CmdHandler thread	200
4.44 Flowchart: Local System – TwitterShare thread	201

---

## List of Figures

---

4.45 Flowchart: Local System — FileTransfer thread	202
4.46 Flowchart: Local System — VidMan thread	203
4.47 Flowchart: Local System — FragMan thread	204
4.48 Flowchart: Local System — GIFGenerator thread	205
5.1 MainWindow views	210
5.2 Final PCB	211
5.3 MainWindow views	217
5.4 BrandWindow views	223
5.5 AdminWindow views	228
5.6 Buildroot setup: libcurl	237
5.7 Buildroot setup: Qt5	238
5.8 Buildroot setup: Opencv4	238
5.9 Buildroot setup: Networking	239
6.1 MDO Local System (MDO-L) Layout Test Cases	281
6.2 MDO-L Logic (interface based) Test Cases	281
6.3 Bad Login Test Case	282
6.4 Register Test Case	282
6.5 Database Test Case	283
6.6 Communication Between systems Test Case	283
6.7 Update and upload Test	284
6.8 Ultrasonic Sensors Test Cases	285
6.9 Fragrance Diffuser Output Test	285
6.10 Camera Output Test	286
6.11 Computer visions tests	288
6.12 Normal mode: testing	289
6.13 Twitter sharing: testing	290
6.14 Download Ad: testing	291
A.1 Project planning — Gantt diagram	305

# List of Tables

2.1	Events: remote client	19
2.2	Events: Remote Server	31
2.3	Events: local system	42
2.4	Budget estimation	53
3.1	An instance of the students relation – withdrawn from [59]	113
4.1	Total spending on Hardware	162
4.2	Pin Mapping	165
4.3	Hardware Test Cases	166
4.4	Remote Client Test Cases	206
4.5	Remote Server Test Cases	207
4.6	Local System Test Cases	208

# List of Listings

./listing/pthreads_api.c . . . . .	61
./listing/pthreads_api.c . . . . .	61
./listing/pthreads_api.c . . . . .	62
./listing/pthreads_api.c . . . . .	62
./listing/daemon_ex.c . . . . .	67
./listing/daemon_ex.c . . . . .	68
./listing/daemon_ex.c . . . . .	68
./listing/daemon_ex.c . . . . .	69
./listing/daemon_ex.c . . . . .	69
./listing/daemon_ex.c . . . . .	69
3.1 Makefile example . . . . .	75
./listing/doxygen_example.h . . . . .	80
./listing/doxygen_example.h . . . . .	80
./listing/doxygen_example.h . . . . .	80
./listing/doxygen_example.c . . . . .	80
./listing/doxygen_example.h . . . . .	81
3.2 Example of a documented C header file using Doxygen – Javadoc style . . . . .	81
3.3 Example of a documented C implementation file using Doxygen – Javadoc style . . . . .	82
3.4 Example of a Doxyfile – excerpt . . . . .	85
3.5 MySQL Connector example – adapted from [66] . . . . .	124
3.6 Example of camera frames acquisition (withdrawn from [70]) . . . . .	128
3.7 Example of video playback in OpenCV using C++ - Player class (withdrawn from [73]) . .	132
3.8 Example of video playback in OpenCV using C++ - Player class implementation(withdrawn from [73]) . . . . .	133
3.9 Example on how to implement filters in faces . . . . .	135
3.10 Implementing a simple window in Qt . . . . .	147
3.11 POST example using <curlpp.h> . . . . .	151

---

## LIST OF LISTINGS

---

5.1	Declaration of MainWindow class . . . . .	210
5.2	Implementation of the constructor <b>MainWindow</b> . . . . .	213
5.3	Declaration of BrandWindow class . . . . .	216
5.4	Implementation of rent function on BrandWindow class . . . . .	219
5.5	Declaration of AdminWindow class . . . . .	222
5.6	Implementation of Manage Users from AdminWindow class . . . . .	225
5.7	Implementation of Ads TO Activate from AdminWindow class . . . . .	226
5.8	Script to create Data Base . . . . .	229
5.9	Script to populate Data Base . . . . .	230
5.10	Implementation of Server Thread . . . . .	232
5.11	Implementation of Receive from MDO-L Thread . . . . .	234
5.12	Implementation of Send to MDO-L Thread . . . . .	234
5.13	Implementation of Update MDO-L Thread . . . . .	235
5.14	CMakeLists.txt file: main build configuration file . . . . .	237
5.15	raspToolchain.cmake file: toolchain setup file . . . . .	240
5.16	Initialization script . . . . .	241
5.17	Fragrance diffuser kernel module (excerpt) . . . . .	242
5.18	Ultrasonic sensor kernel module (excerpt) — adapted from [114] . . . . .	244
5.19	User detection daemon . . . . .	250
5.20	NormalWindow — an example of a subordinate view . . . . .	254
5.21	<b>MainWindow</b> — interface . . . . .	255
5.22	<b>Ad</b> — interface . . . . .	262
5.23	<b>DigitalOut</b> — interface . . . . .	263
5.24	<b>Frag</b> — interface . . . . .	264
5.25	<b>fragDiffuser</b> — interface . . . . .	266
5.26	<b>fragManager</b> — interface . . . . .	268
5.27	<b>imgFilter</b> — interface . . . . .	270
5.28	<b>msgQueue</b> — interface . . . . .	271
5.29	<b>pEvent</b> — interface . . . . .	271
5.30	<b>Post</b> — interface . . . . .	272
5.31	Local system threads . . . . .	273
5.32	Main thread periodic task: checkMode . . . . .	277

# List of Abbreviations

<b>Notation</b>	<b>Description</b>	<b>First used on page nr.</b>
AC	Alternating Current	12
ACF	Aggregated channel feature	106
ANN	Artificial Neural Networks	110
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	104
COTS	Commercial off-the-shelf	5
CPS	Cyber–Physical Systems	1
CPU	Central Processing Unit	107
CSI	Camera Serial Interface	155
CSV	Comma Separated Values	197
CV	Computer Vision	15
DB	Database	12
DBMS	Database Management System	112
DC	Direct Current	12
DDL	Data Definition Language	121
DML	Data Manipulation Language	122
DOOH	Digital Out-Of-Home	2
DPM	Deformable part models	106
ER	Entity-Relationship	112, 114

## List of Abbreviations

---

<b>Notation</b>	<b>Description</b>	<b>First used on page nr.</b>
ERD	Entities-Relationships diagram	119
FIFO	First-In, First-Out	185
FPS	Frames per Second	128
GIF	Graphics Interchange Format	2
GPIO	General Purpose Input/Output	155
GPU	Graphics Processing Unit	107
GUI	Graphical User Interface	146
HD	High-Definition	105
HDMI	High-Definition Multimedia Interface	155
HOG	Histogram of Oriented Gradients	106
HTML	Hypertext Markup Language	85
HTTP	Hypertext Transfer Protocol	139
HW	Hardware	5
I/O	Input/Output	65
IC	Integrated Circuit	102
IOT	Internet of Things	105
IP	Internet Protocol	15
IPC	Inter-Process Communication	59
IR	infrared	127
JSON	Javascript Object Notation	142
KNN	K-nearest neighbor	110
LAN	Local Area Network	62
LCD	Liquid Crystal Display	168
LSB	Least Significant Bit	137
mA	milliampere	157
MAC	Media Access Control	62
MDO	Marketing Digital Outdoor	2

---

**List of Abbreviations**

---

<b>Notation</b>	<b>Description</b>	<b>First used on page nr.</b>
MDO-L	MDO Local System	9
MDO-RC	MDO Remote Client	9
MDO-RS	MDO Remote Server	9
MOSFET	Metal Oxide Semiconductor Field-Effect Transistor	103
MW	Microwave	126
NB	Naive Bayes	110
OCR	Optical Character Recognition	104
OMT	Object-Modeling Technique	58
OOP	Object Oriented Programming	145
OOSE	Object Oriented Software Engineering	58
OS	Operating System	13
OSI	Open Systems Interconnection	13
PCB	Printed Circuit Board	5
PGID	Process Group ID	68
PID	Process ID	68
PIR	Passive Infrared	125
PoE	Power over Ethernet	156
POSIX	Portable Operating System Interface	58
PWM	Pulse-Width Modulation	96
R&D	Research and Development	3
RDBMS	Relational Database Management System	14
REST	Representation State Transfer	142
ROI	Region Of Interest	110
SDK	Software Development Kit	139
SID	Session ID	68
SoC	System-on-a-Chip	12
SQL	Structured Query Language	112
SVM	Support Vector Machine	106
SW	Software	5
TCP	Transmission Control Protocol	63

---

**List of Abbreviations**

---

<b>Notation</b>	<b>Description</b>	<b>First used on page nr.</b>
TCP/IP	Transmission Control Protocol/Internet Protocol	9
TTL	Transistor-Transistor Logic	155
UDP	User Datagram Protocol	63
UI	User Interface	6
UML	Unified Modeling Language	57
URL	Uniform Resource Locator	145
USD	United States Dollar	105
WAL	Write-Ahead Log	115
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 tell 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].

#### 1.4. Project goals

---

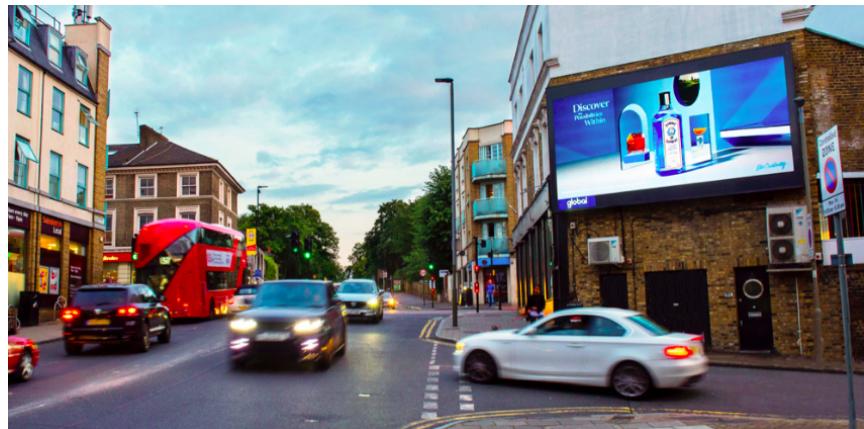


Figure 1.1.: Example of a Digital Outdoor, withdrawn from [4]

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

## 1.5. Project planning



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 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

## 1.6. Report Outline

---

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 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 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 PCB is designed, manufactured and assembled. After designing the PCB, the enclosure is designed to accommodate 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

## **1.6. Report Outline**

---

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.
- In Chapter 4 the theoretical foundations are used to design a viable solution, accordingly to the requirements and constraints listed and the information gathered in the analysis phase. In the design phase, the product development starts, specifying the system in terms of hardware and software and its associated interfaces, the error handling required, and the design verification.
- Lastly, the appendices (see Section 7.2) 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).

## 2.2. System overview

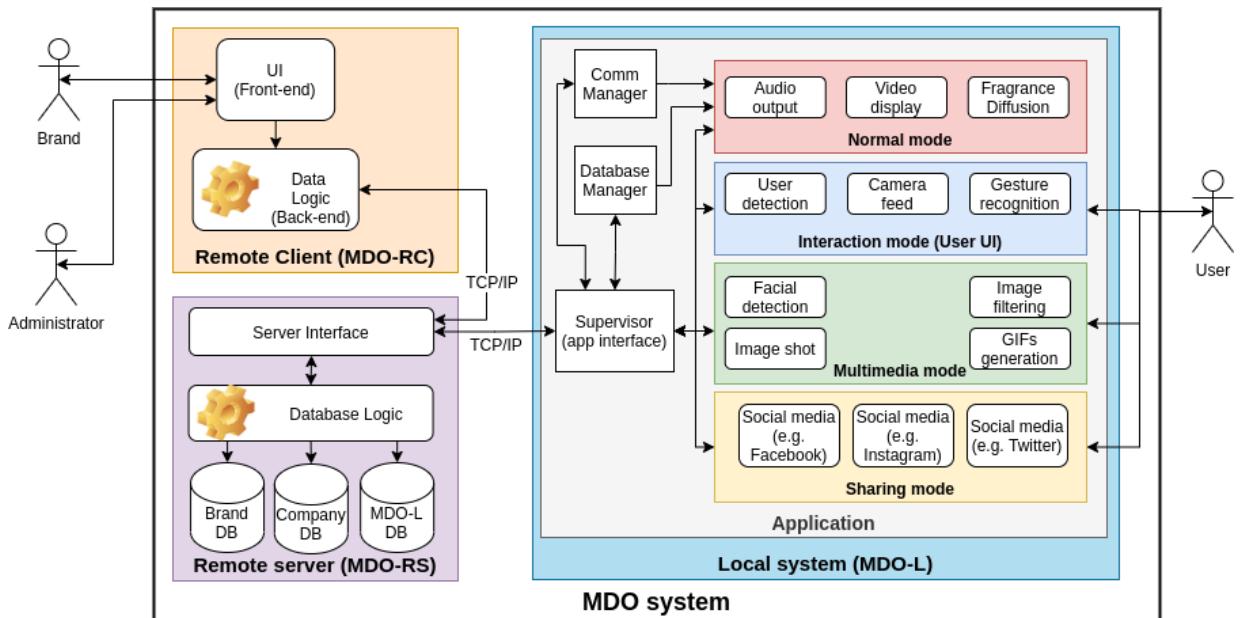


Figure 2.1.: MDO system overview

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-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

## 2.2. System overview

---

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

### 2.3. System architecture

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**.

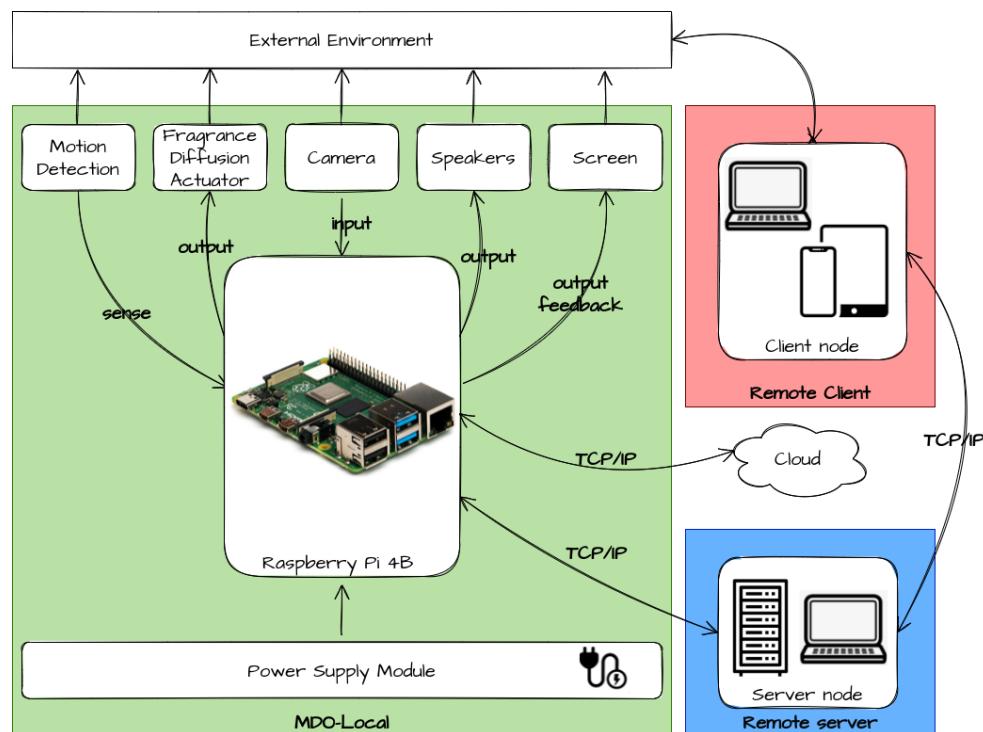


Figure 2.2.: HW architecture diagram

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).

### 2.3. System architecture

---

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.

### 2.3. System architecture

- 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.

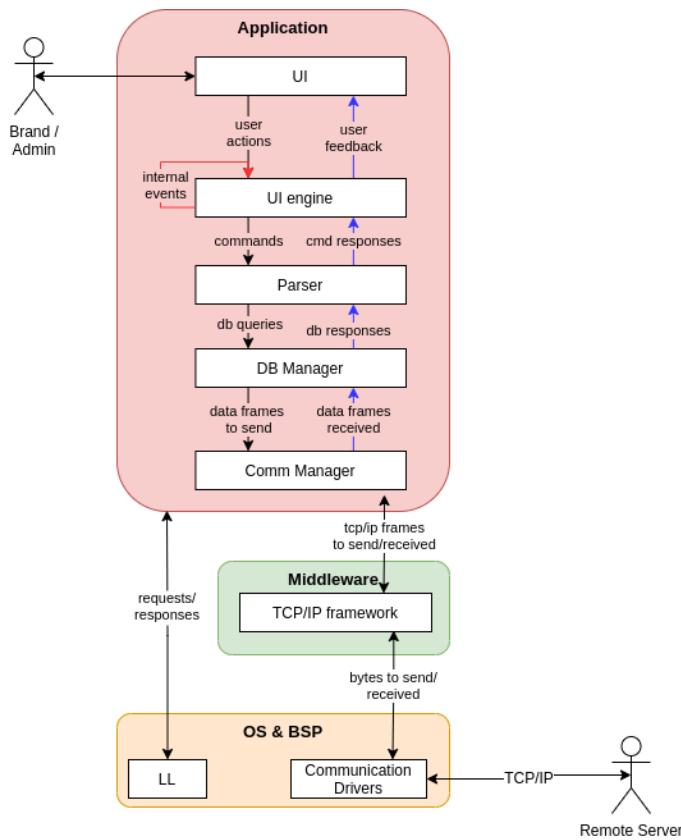


Figure 2.3.: SW architecture diagram: remote client

### 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**

### 2.3. System architecture

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 handle requests from the **Remote Client**, and the **File I/O** drivers to manipulate DB transactions from/to storage.

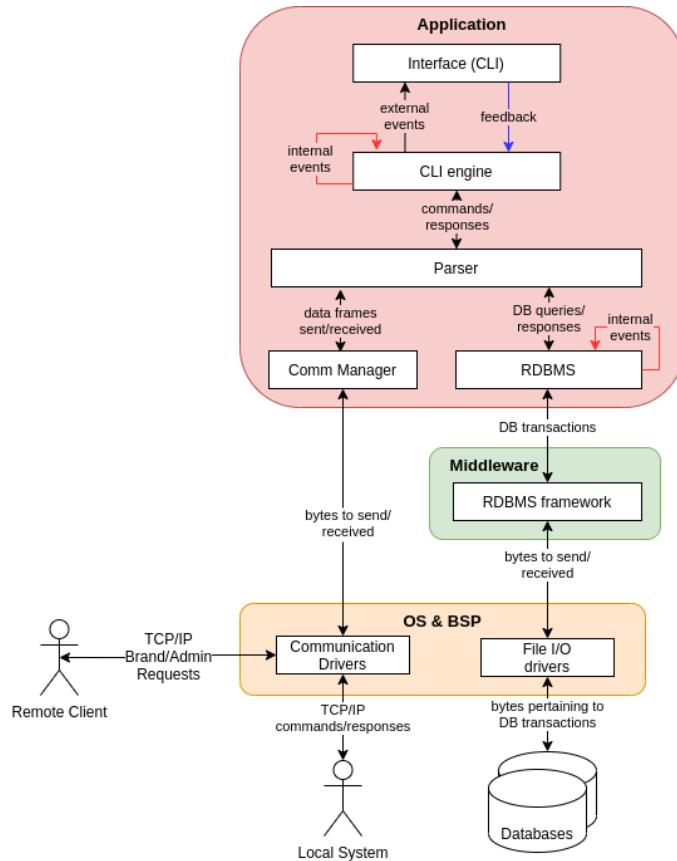


Figure 2.4.: SW architecture diagram: remote server

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;

#### 2.4. Subsystem decomposition

---

- 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 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).

## 2.4. Subsystem decomposition

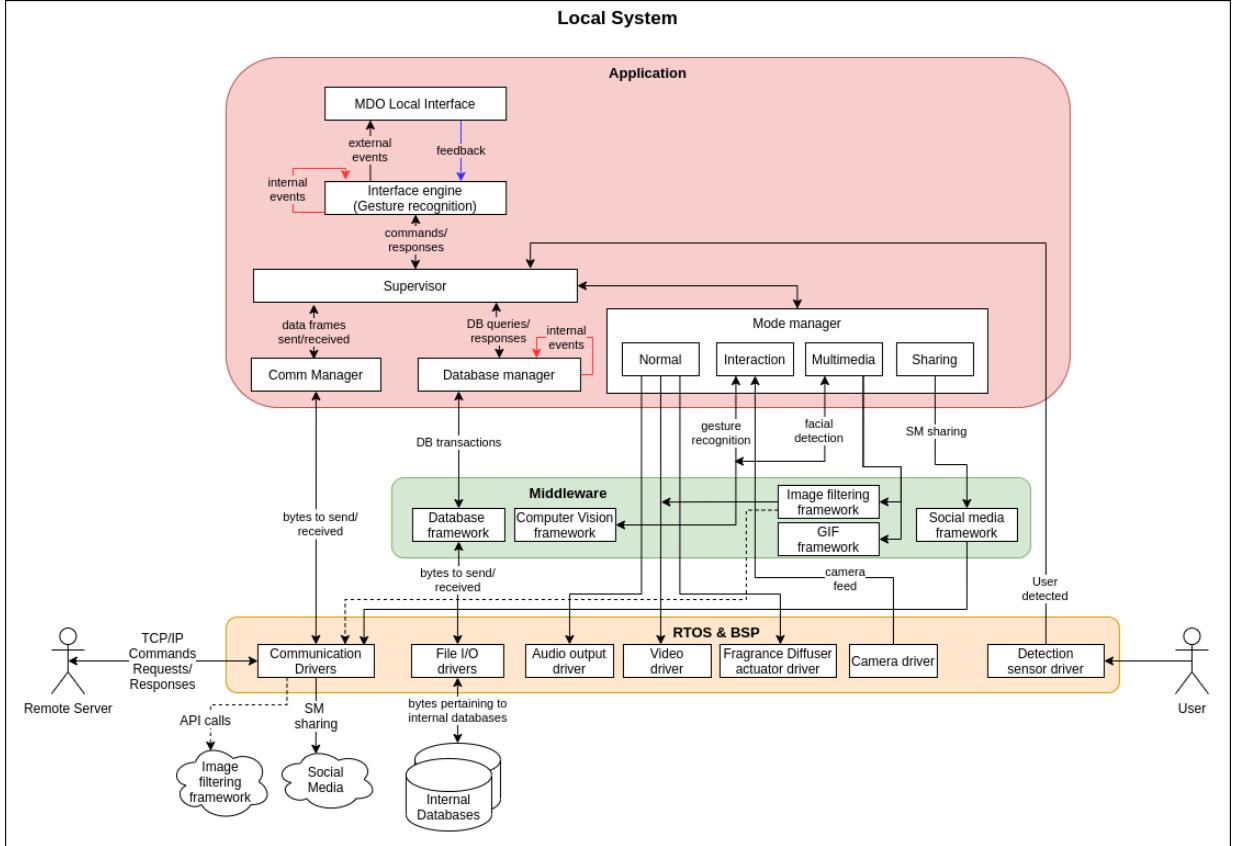


Figure 2.5.: SW architecture diagram: local system

### 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:

## 2.4. Subsystem decomposition

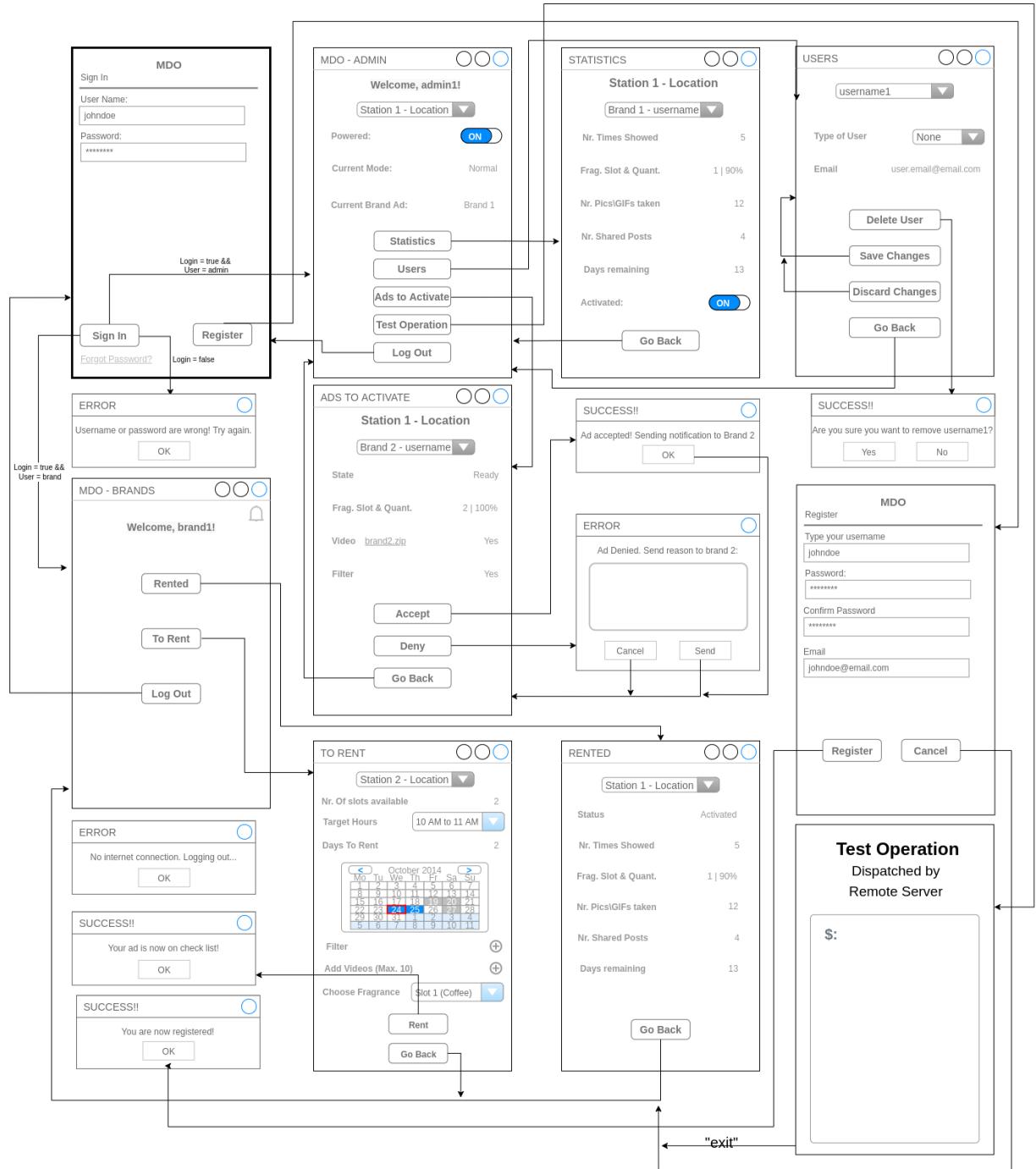


Figure 2.6.: User mockups: remote client

- 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:

## 2.4. Subsystem decomposition

---

- 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.

#### 2.4. Subsystem decomposition

---

Table 2.1.: Events: remote client

<b>Event</b>	<b>System response</b>	<b>Source</b>	<b>Type</b>
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.

## 2.4. Subsystem decomposition

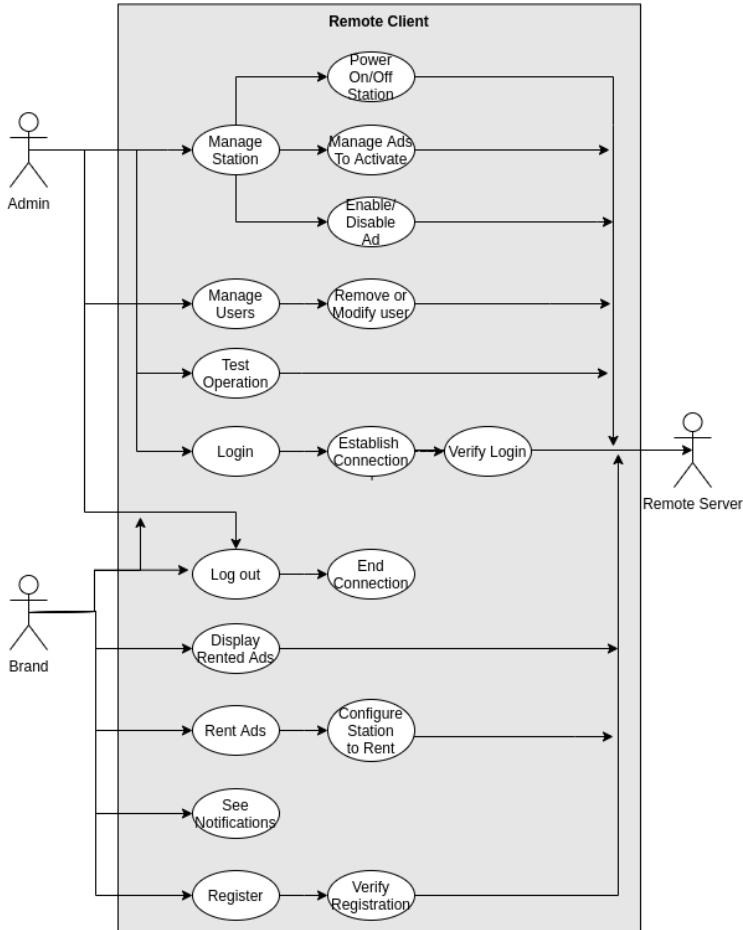


Figure 2.7.: Use cases: remote client

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

## 2.4. Subsystem decomposition

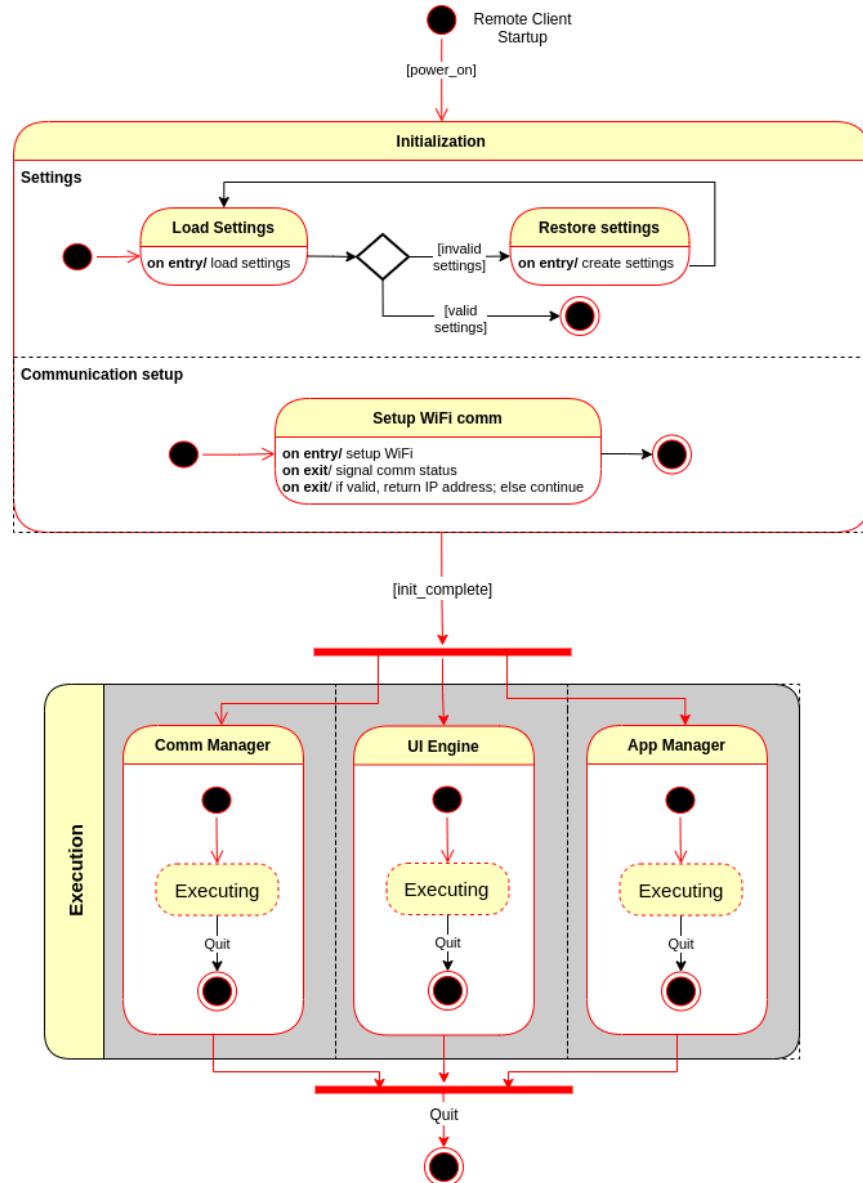


Figure 2.8.: State Machine Diagram: remote client

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.

## 2.4. Subsystem decomposition

### 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 connect, 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.

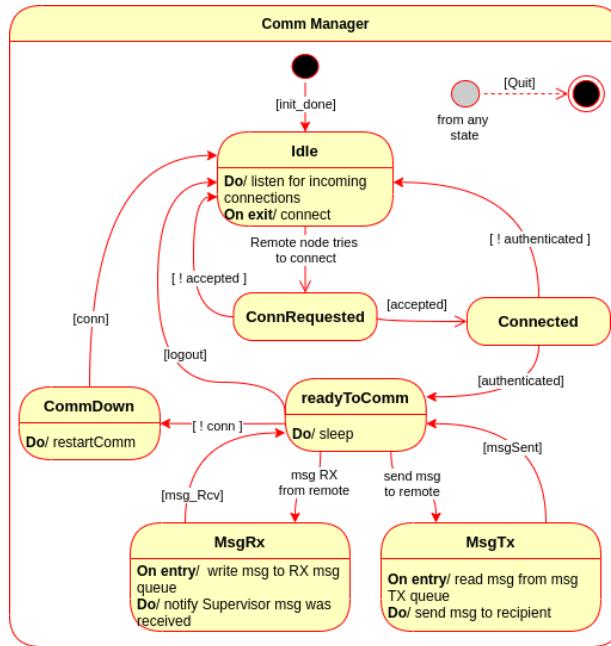


Figure 2.9.: State Machine Diagram: remote client – Communication Manager

### 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 activate (**Ads To Activate**) and test operations on the machines (**Test Operation**).

## 2.4. Subsystem decomposition

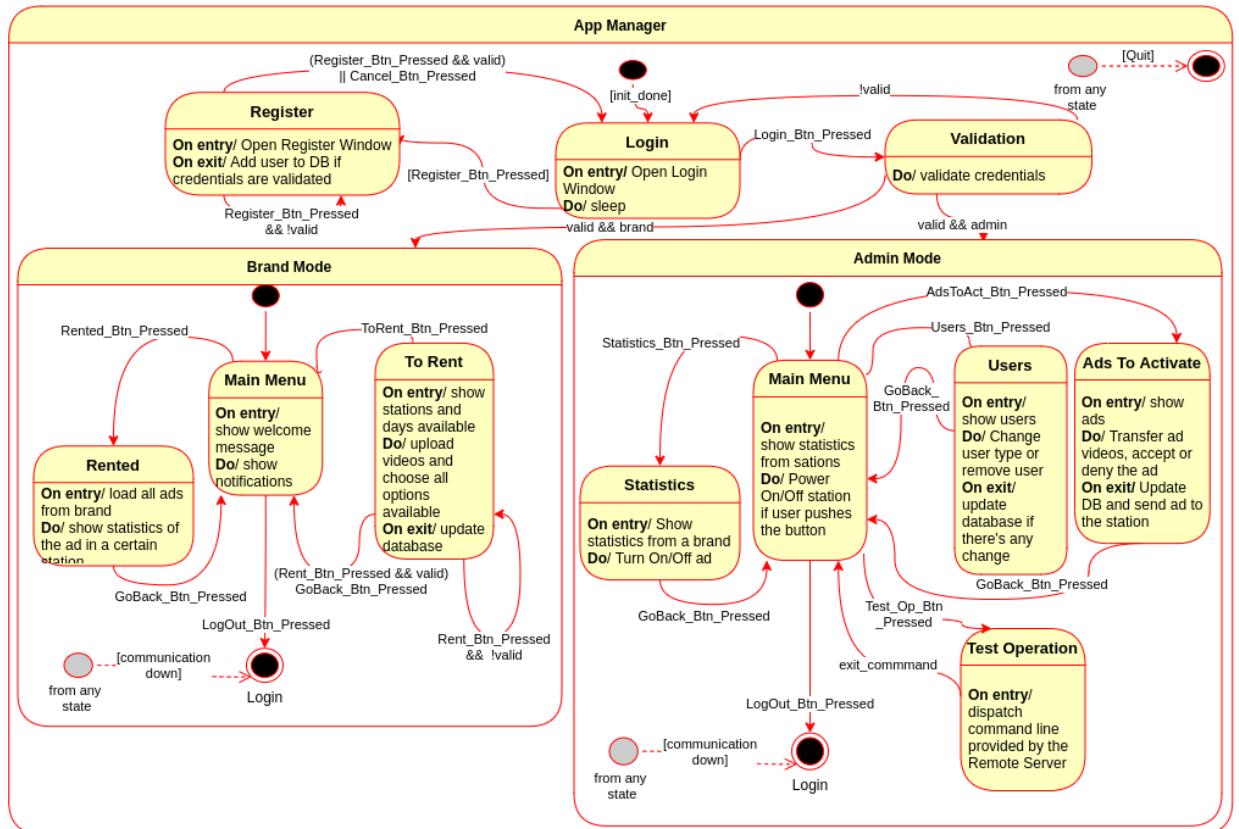


Figure 2.10.: State Machine Diagram: remote client – App Manager

- **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

## 2.4. Subsystem decomposition

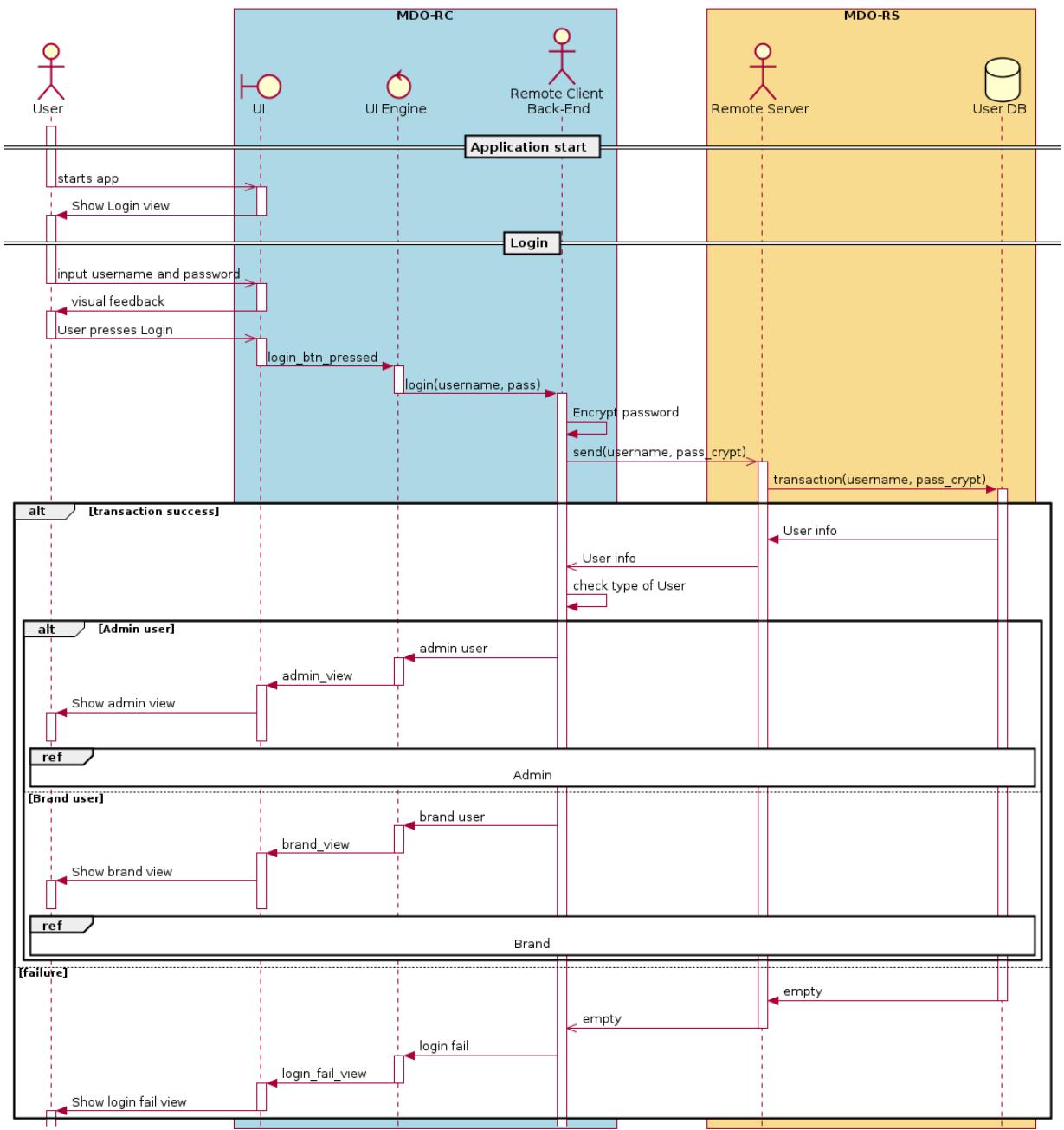


Figure 2.11.: Sequence Diagram: remote client – Login

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.

## 2.4. Subsystem decomposition

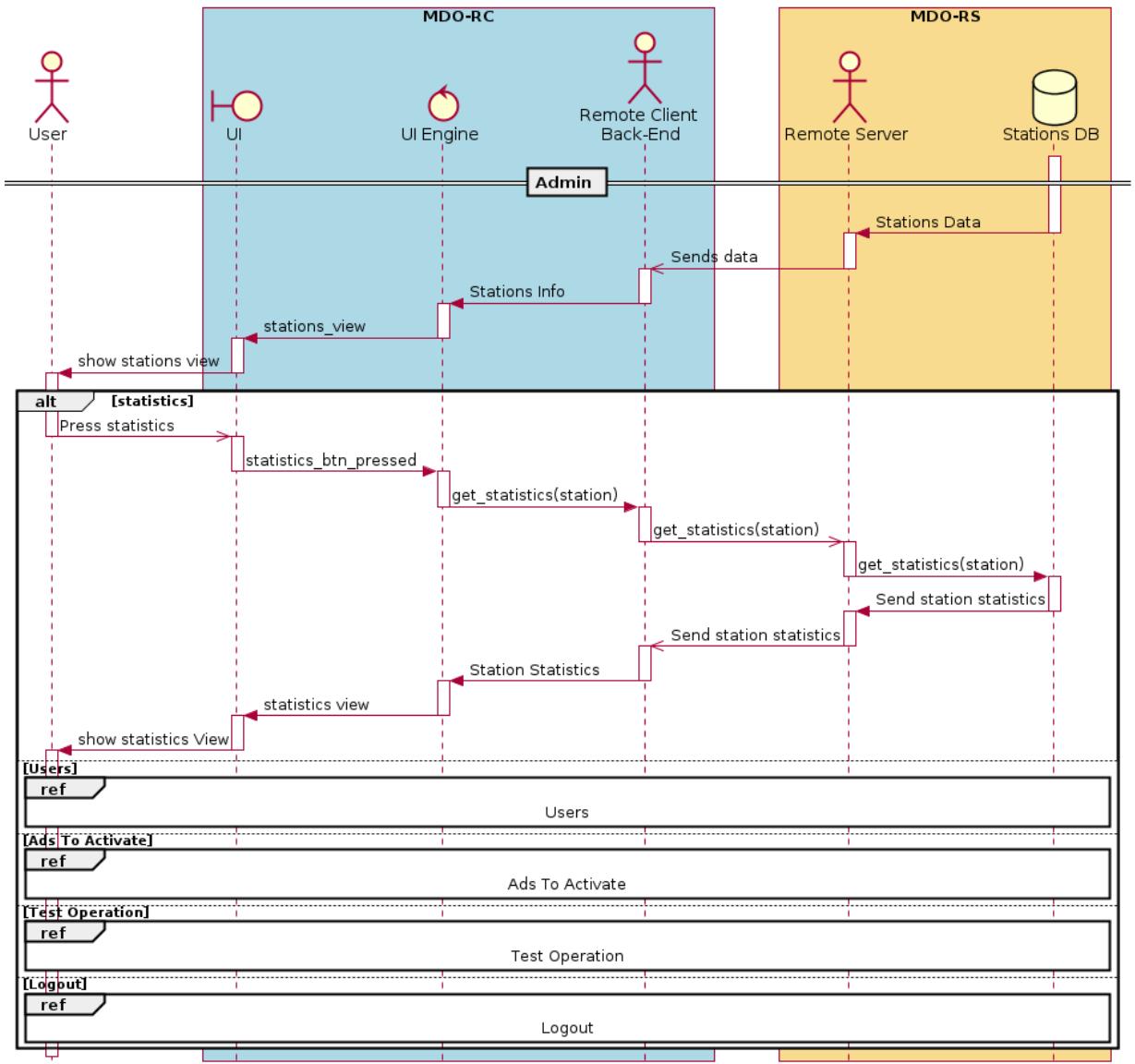


Figure 2.12.: Sequence Diagram: remote client – admin statistics

===== 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

## 2.4. Subsystem decomposition

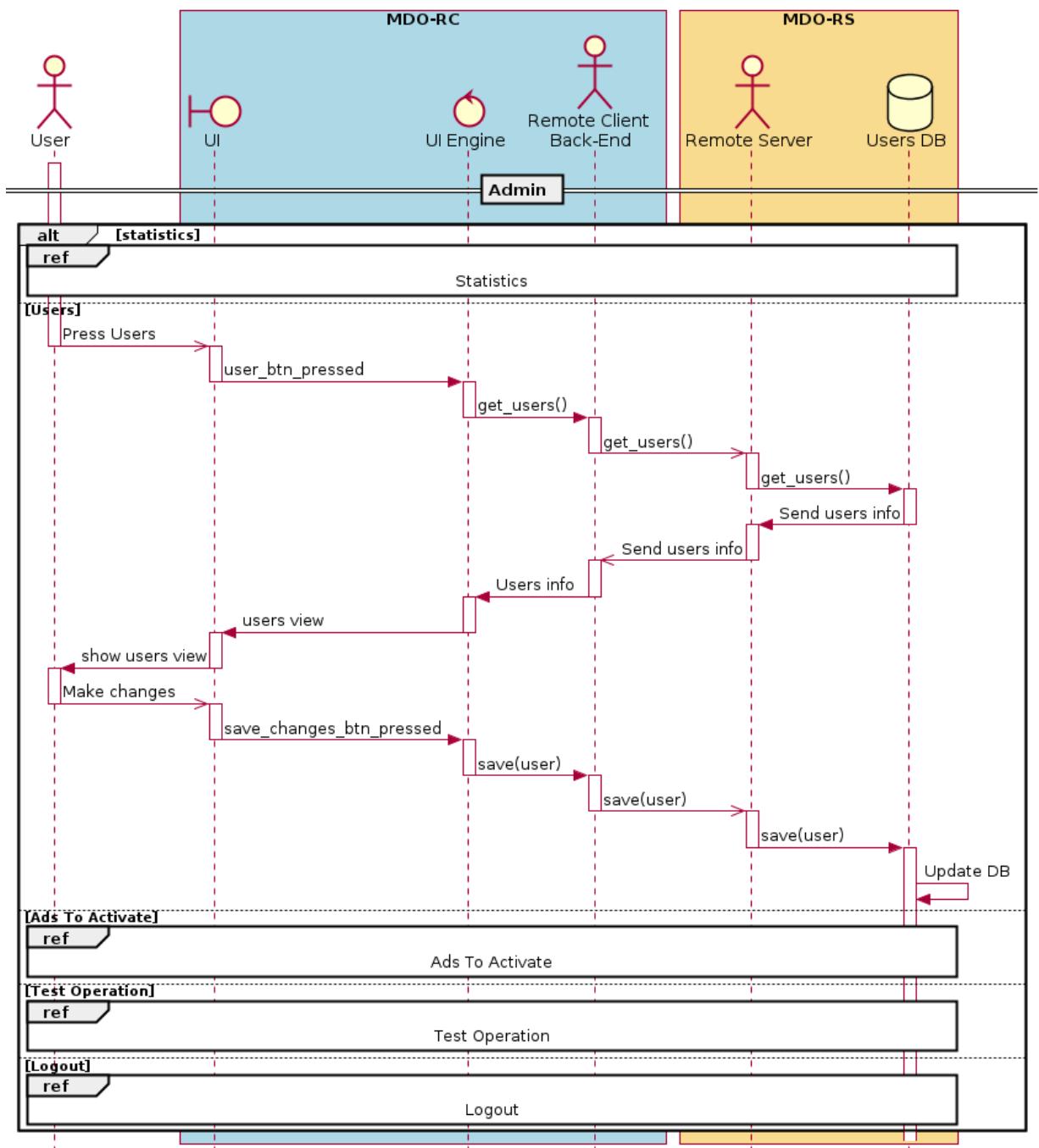


Figure 2.13.: Sequence Diagram: remote client – admin users

query or update them. »»»> 1b64a1fa2e419d79801eaaa806f4d9675f06902e

## 2.4. Subsystem decomposition

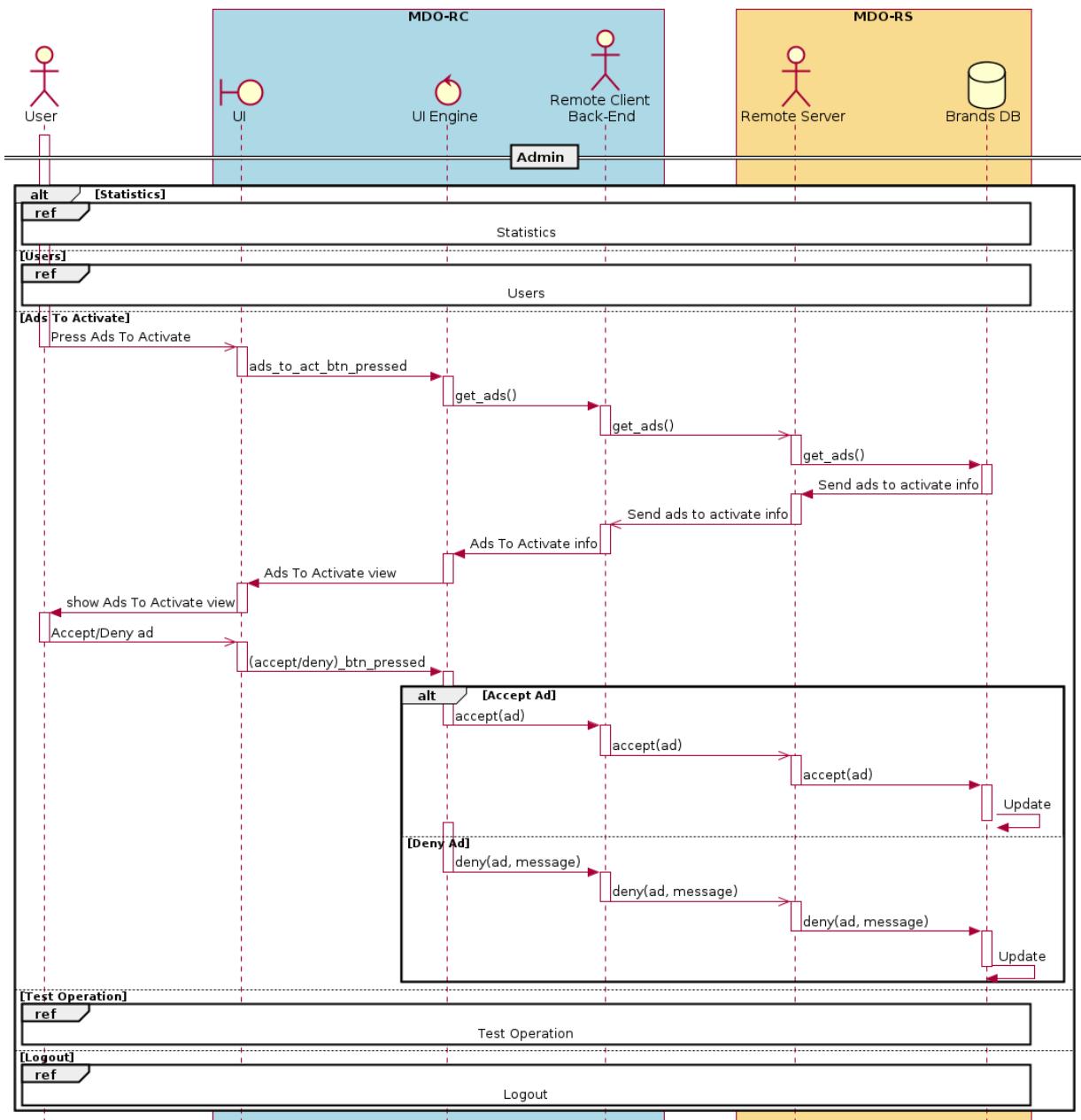


Figure 2.14.: Sequence Diagram: remote client — admin ads to activate

### 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.

## 2.4. Subsystem decomposition

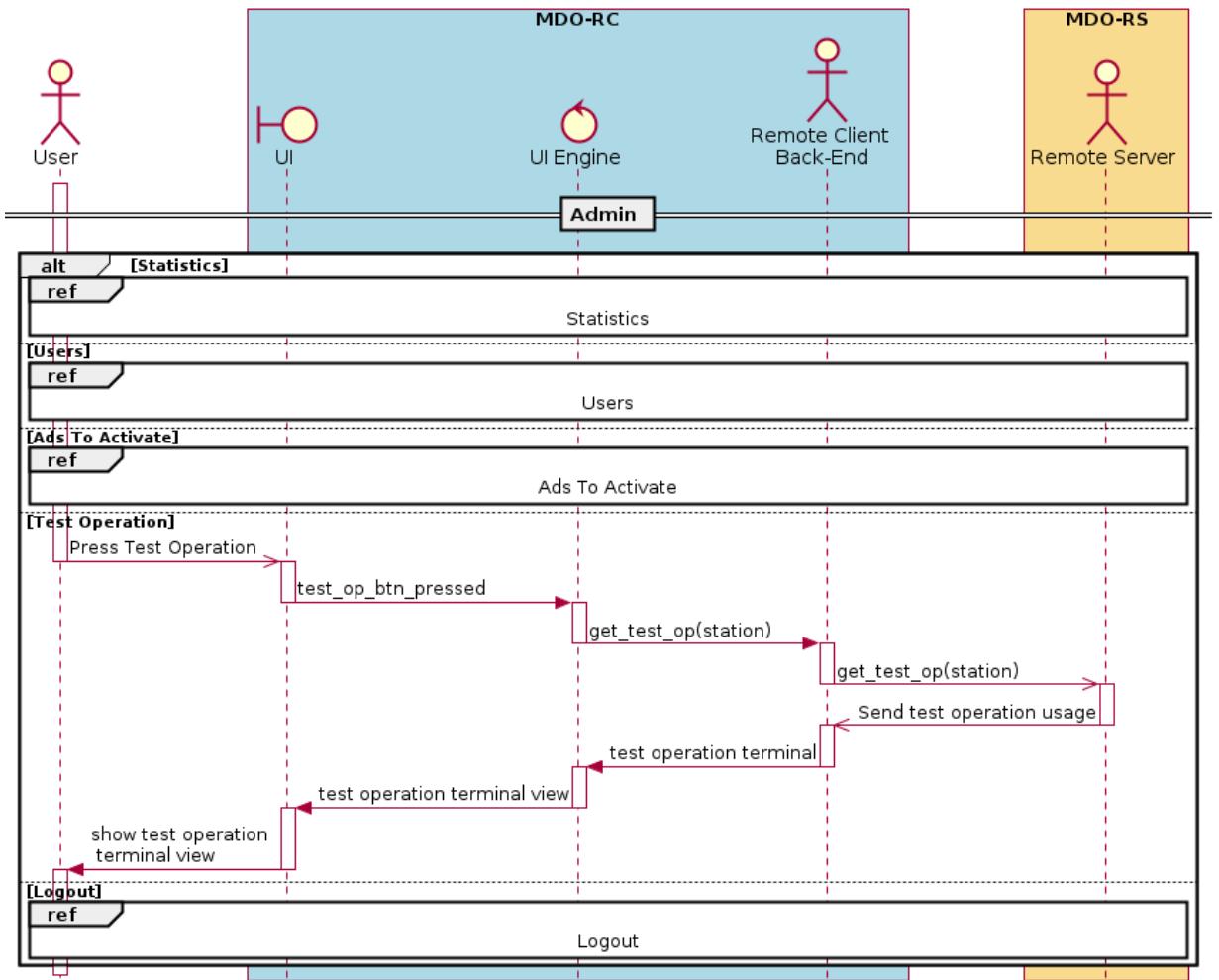


Figure 2.15.: Sequence Diagram: remote client – admin test operation

### 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;

## 2.4. Subsystem decomposition

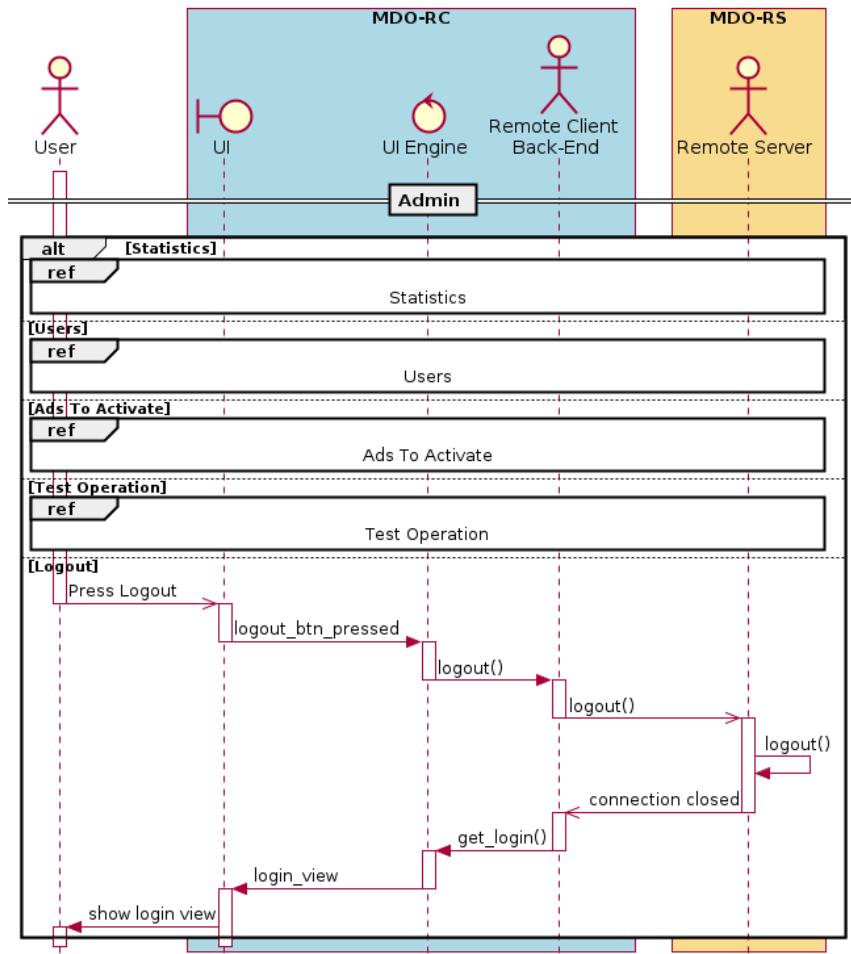


Figure 2.16.: Sequence Diagram: remote client – admin logout

- 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.

## 2.4. Subsystem decomposition

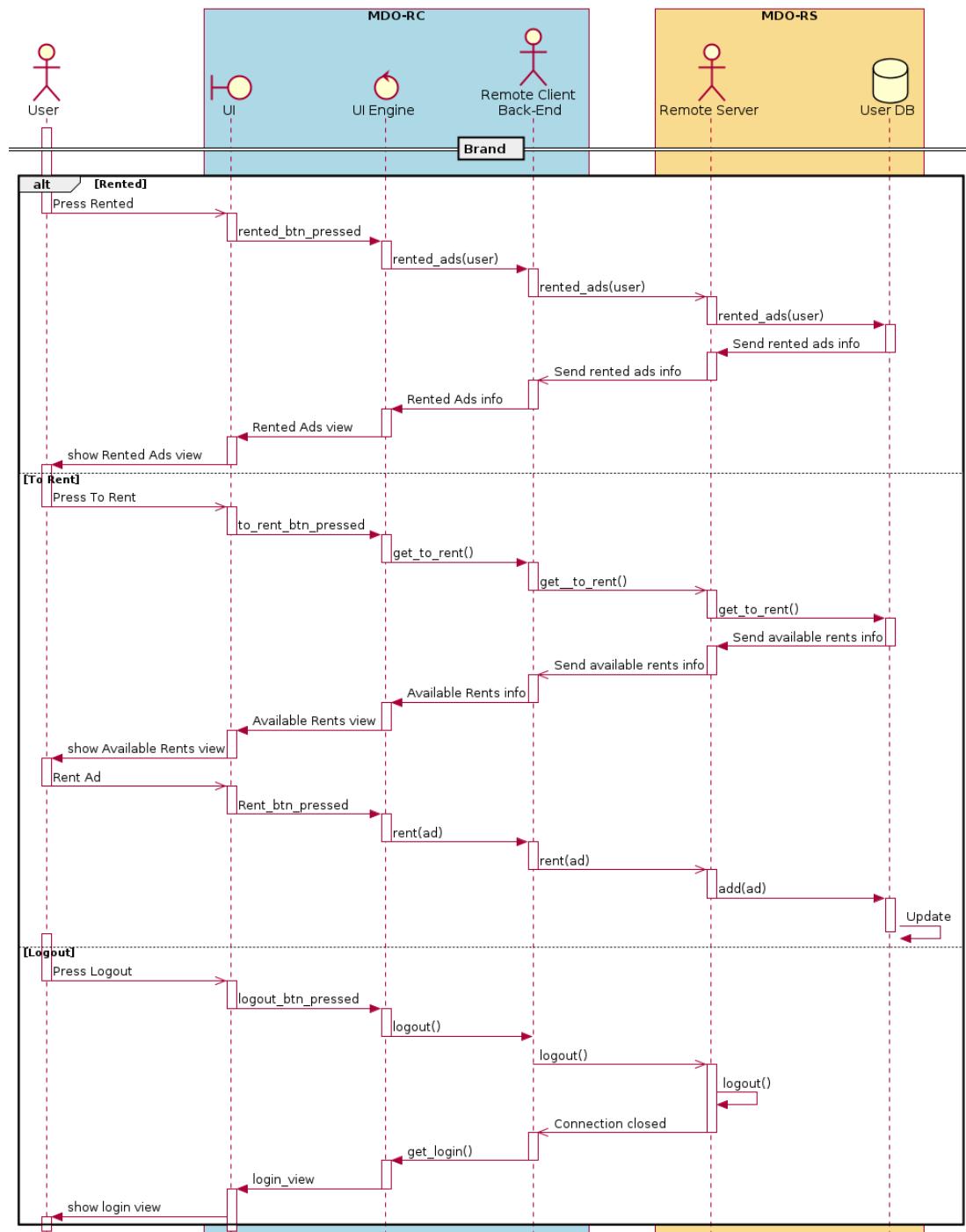


Figure 2.17.: Sequence Diagram: remote client - brand

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

## 2.4. Subsystem decomposition

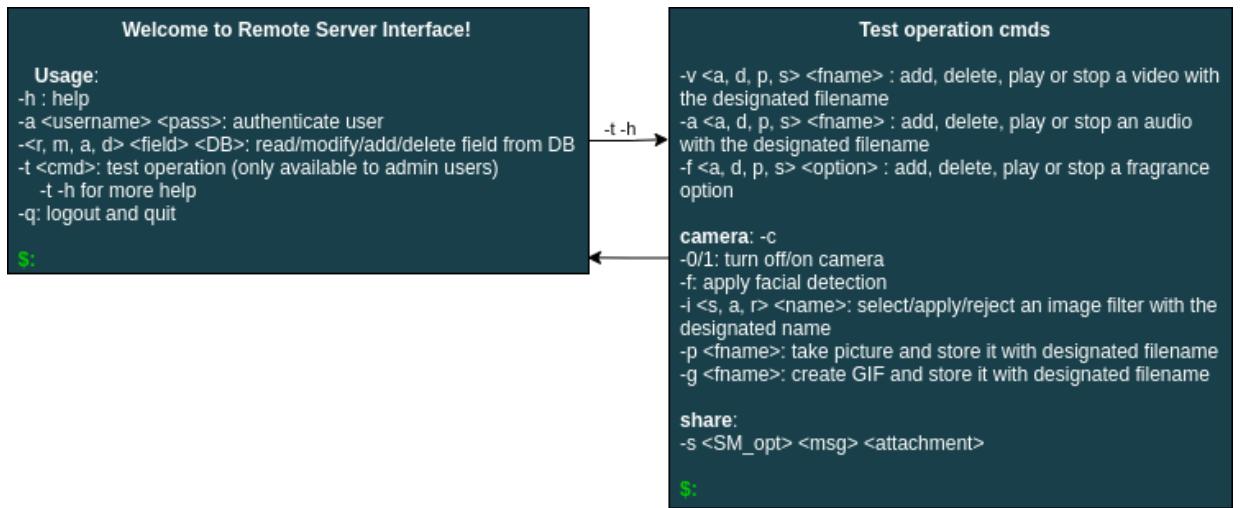


Figure 2.18.: User mock-ups: Remote Server

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.

## 2.4. Subsystem decomposition

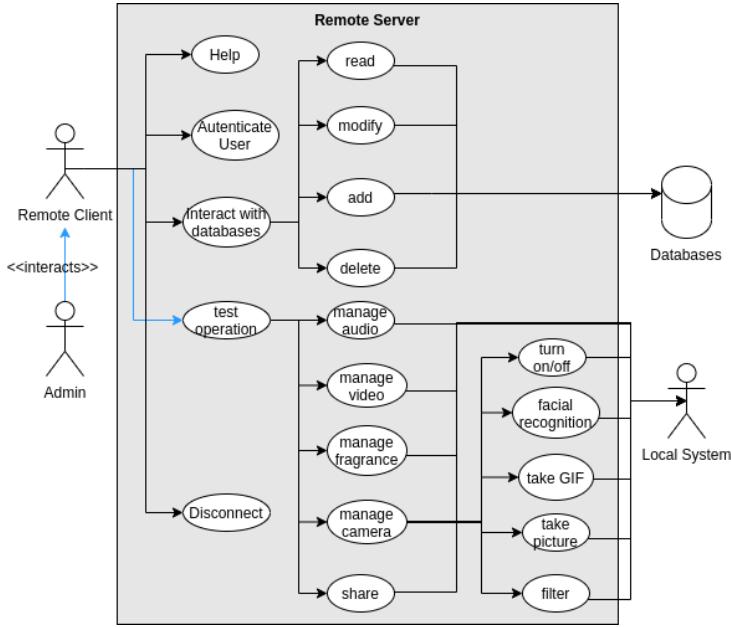


Figure 2.19.: Use cases: remote server

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.

## 2.4. Subsystem decomposition

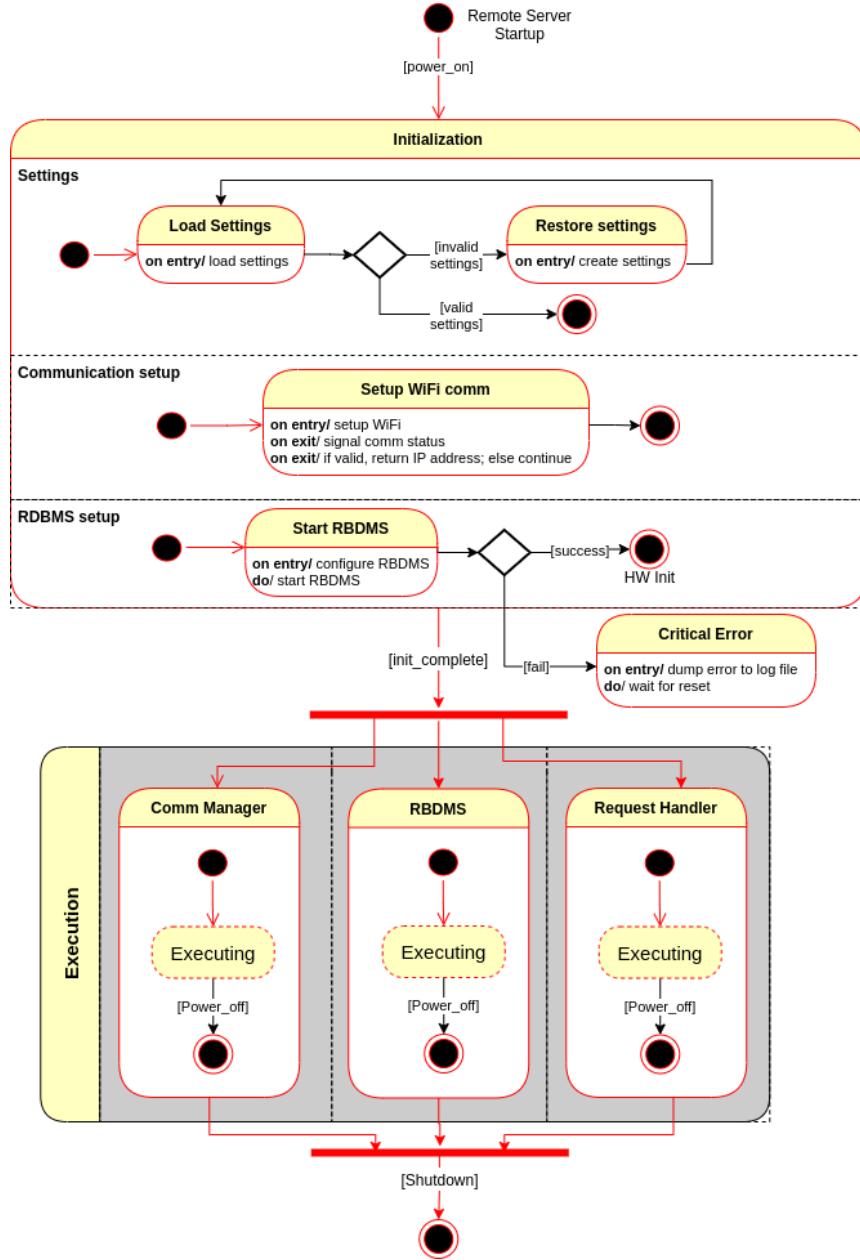


Figure 2.20.: State machine diagram: Remote server

### 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**

## 2.4. Subsystem decomposition

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.

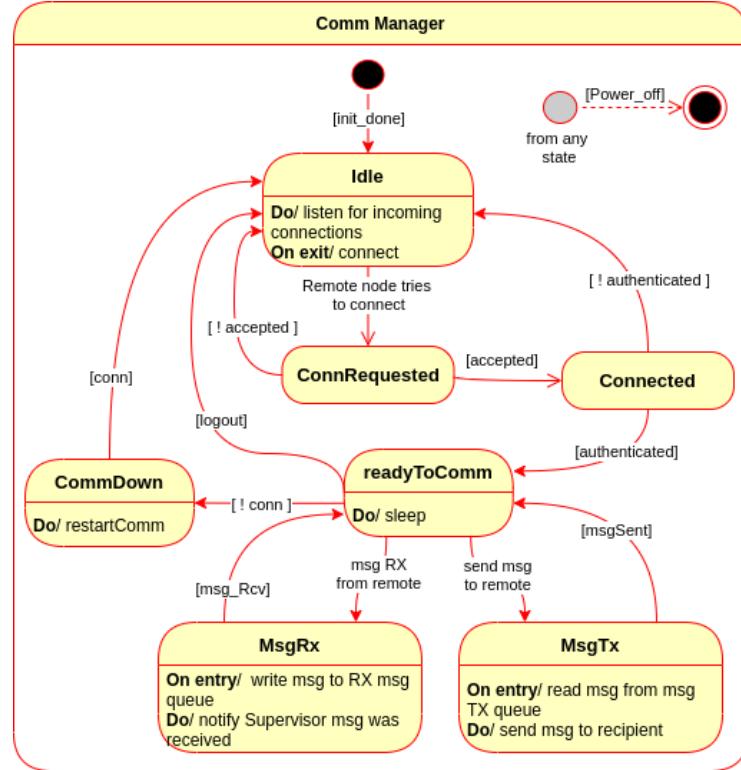


Figure 2.21.: State machine diagram: Remote Server – Comm Manager

### 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.

## 2.4. Subsystem decomposition

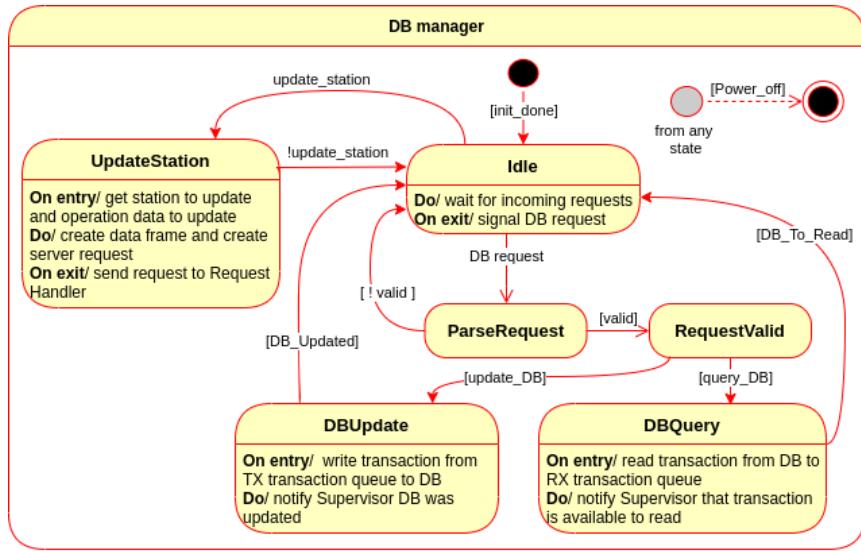


Figure 2.22.: State machine diagram: Remote Server – DB Manager

### 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.

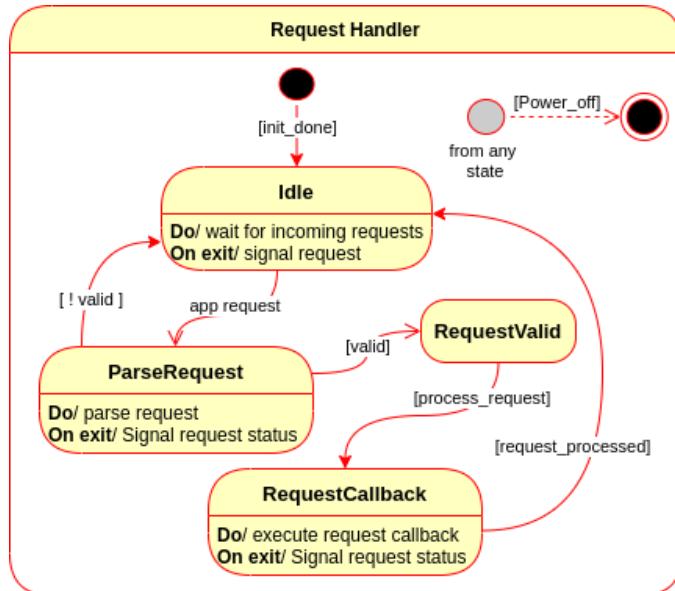


Figure 2.23.: State machine diagram: Remote Server – Request Handler

## 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).

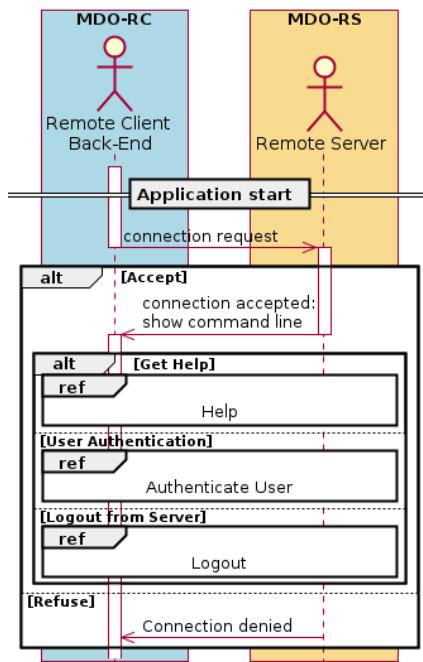


Figure 2.24.: Sequence diagram: Remote Server

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 user-name and the password to the **Remote Server**. The latter searches for the username in the **User DB**, if it is not found, the authentication 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

## 2.4. Subsystem decomposition

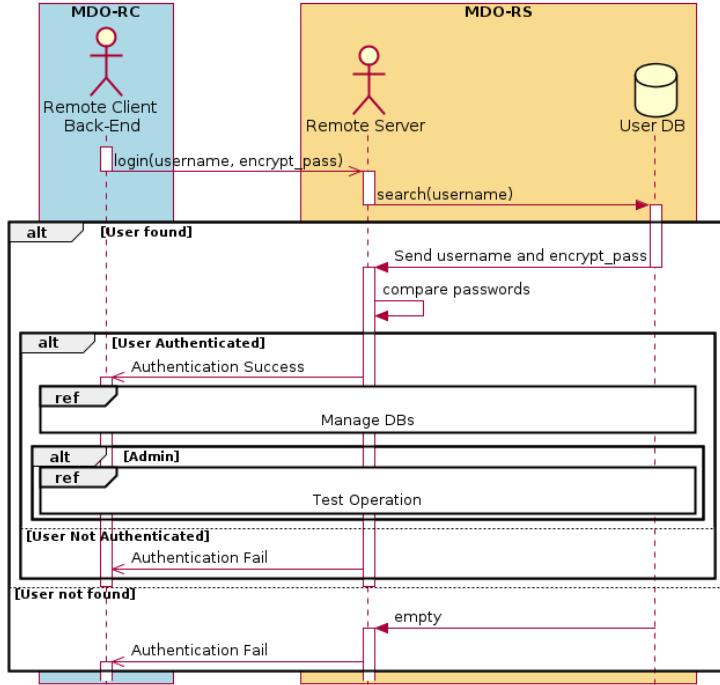


Figure 2.25.: Sequence diagram: Remote Server – Authentication

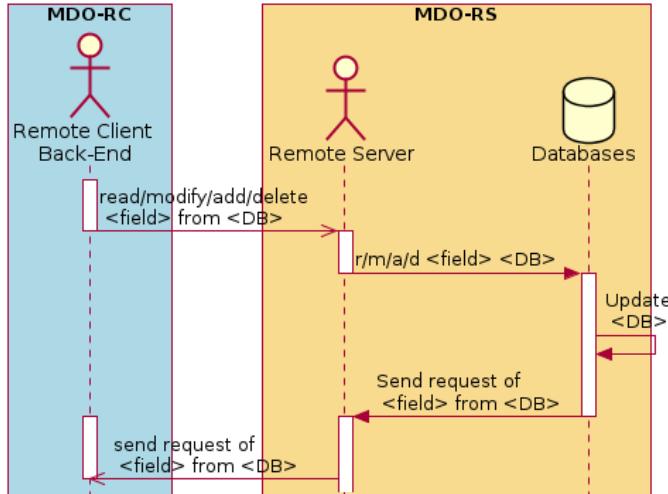


Figure 2.26.: Sequence diagram: Remote Server – Manage Databases

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.

## 2.4. Subsystem decomposition

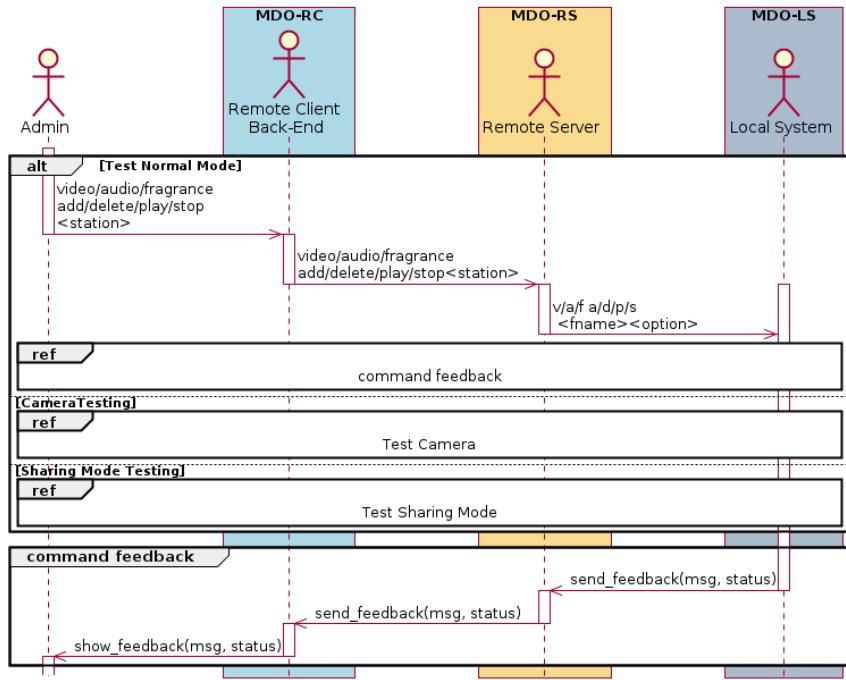


Figure 2.27.: Sequence diagram: Remote Server – Test Operation

### 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 be seen, 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 mode.

## 2.4. Subsystem decomposition

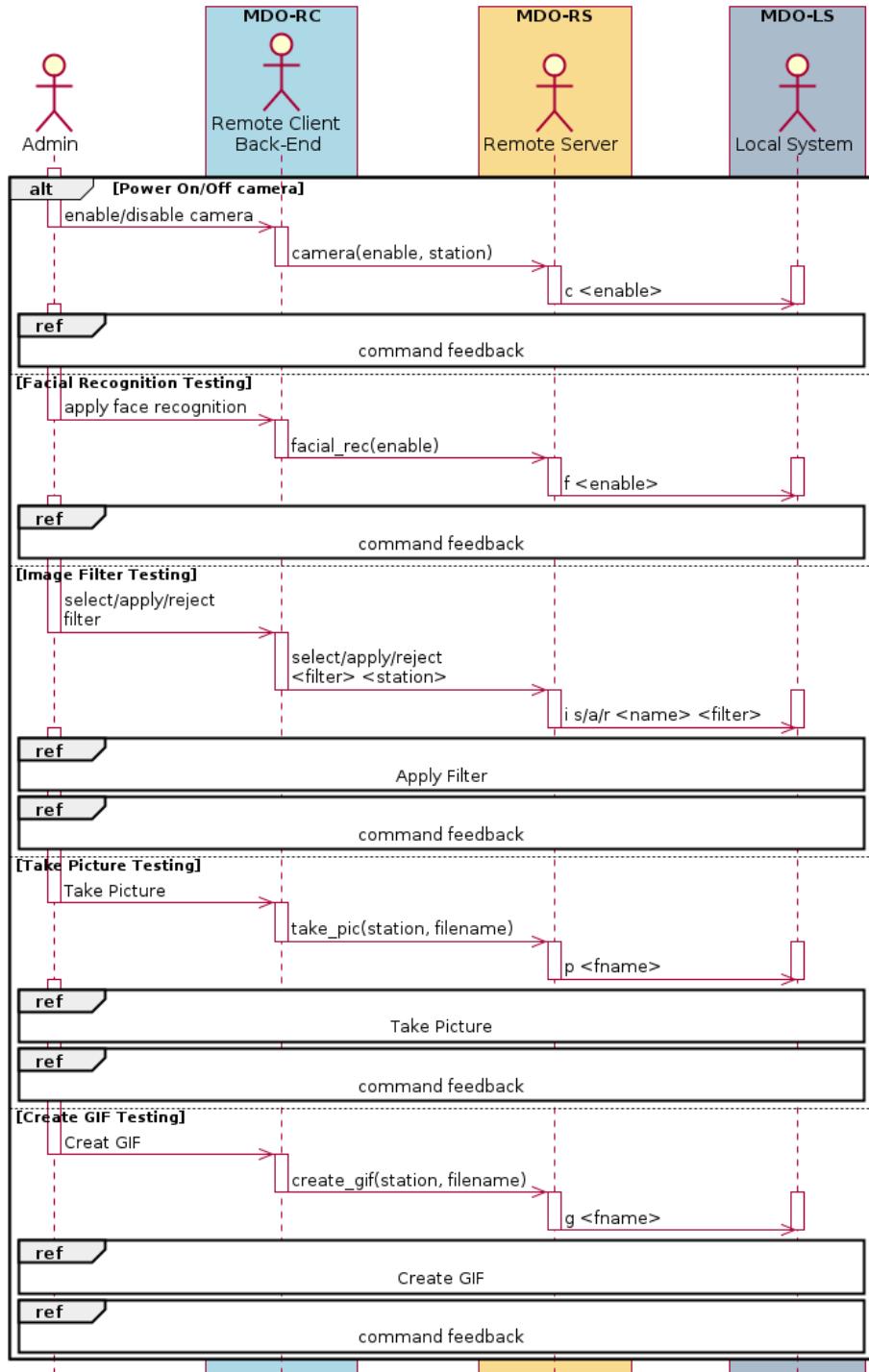


Figure 2.28.: Sequence diagram: Remote Server — Test Operation Camera

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

## 2.4. Subsystem decomposition

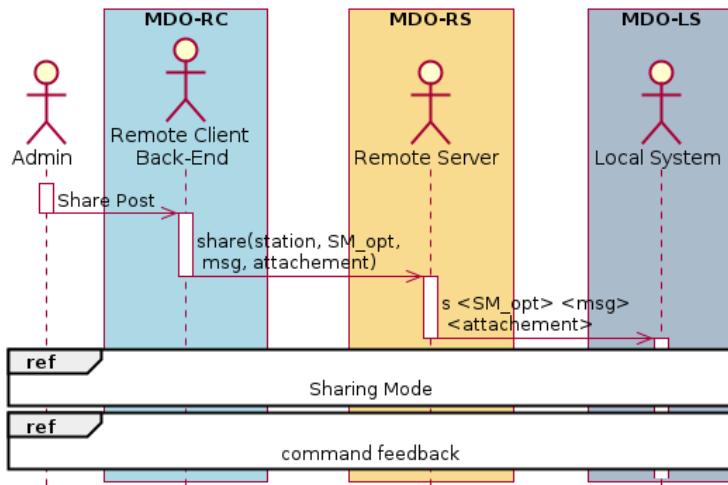


Figure 2.29.: Sequence diagram: Remote Server – Test Operation Share

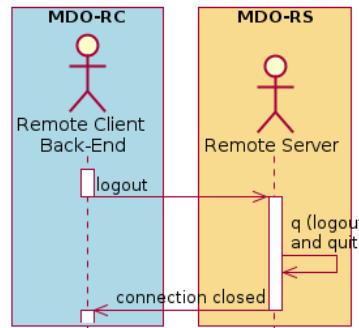


Figure 2.30.: Sequence diagram: Remote Server – logout

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.

## 2.4. Subsystem decomposition

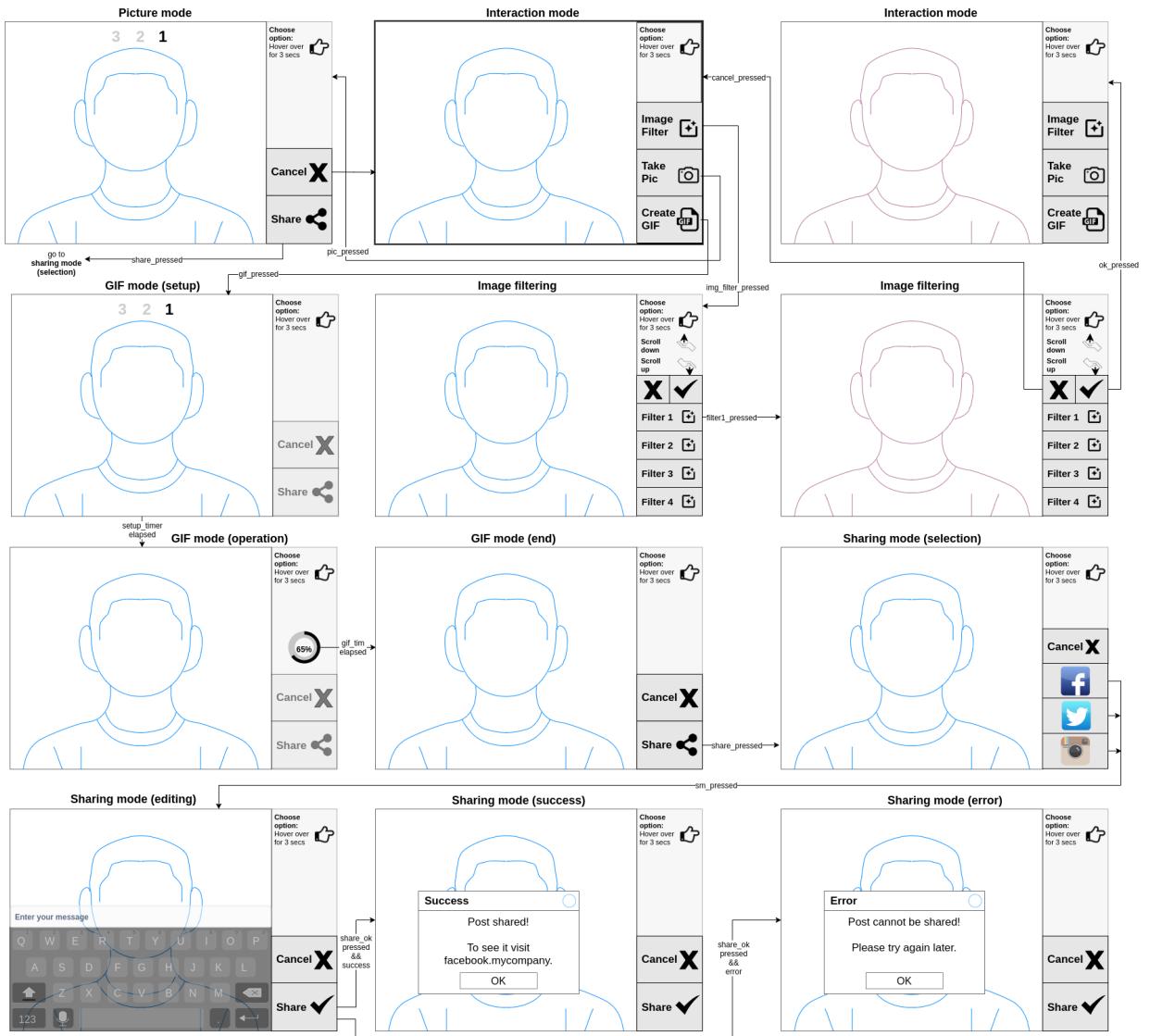


Figure 2.31.: User mock-ups: local system

## 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.

## 2.4. Subsystem decomposition

---

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
<b>UI events</b>			
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.

## 2.4. Subsystem decomposition

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.

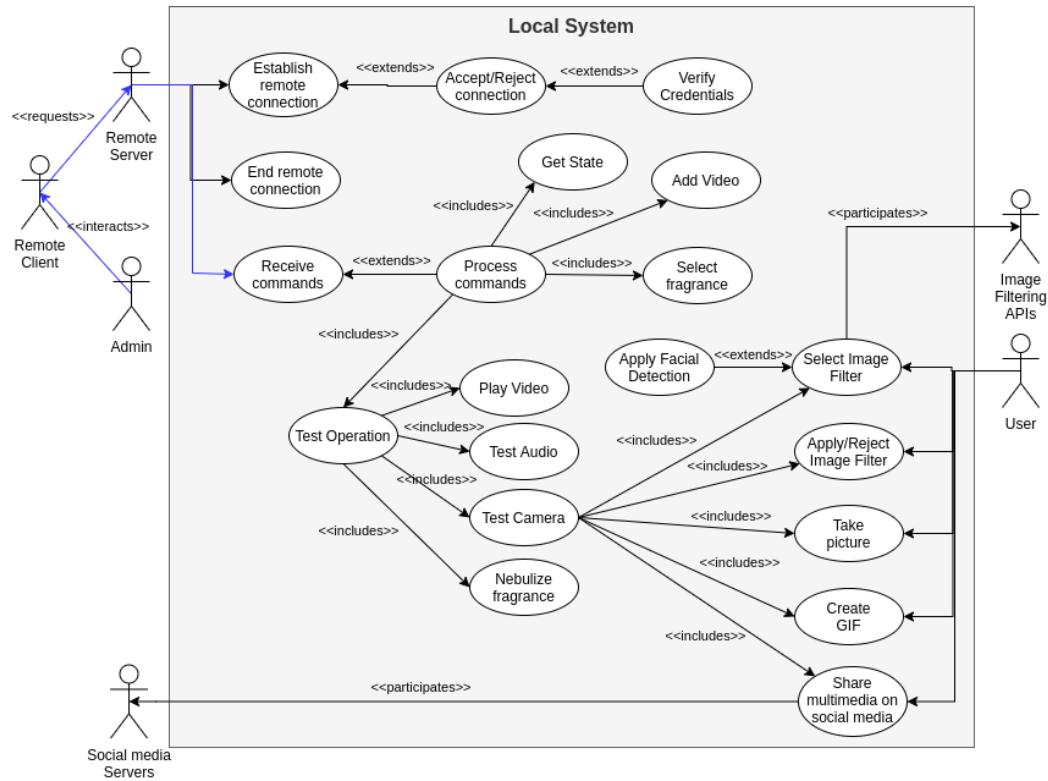


Figure 2.32.: Use cases diagram: local system

## 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

## 2.4. Subsystem decomposition

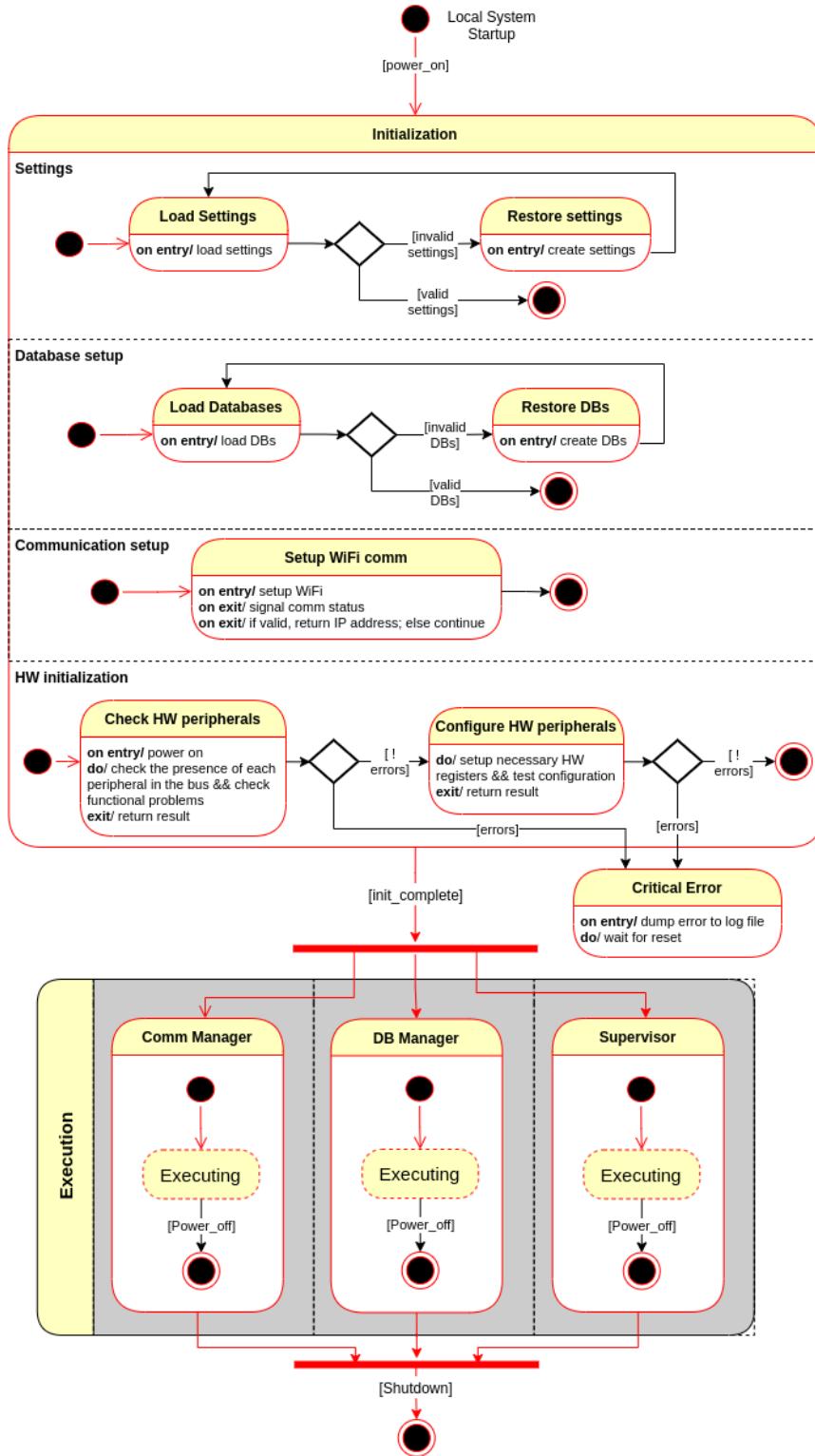


Figure 2.33.: State machine diagram: local system

## 2.4. Subsystem decomposition

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.

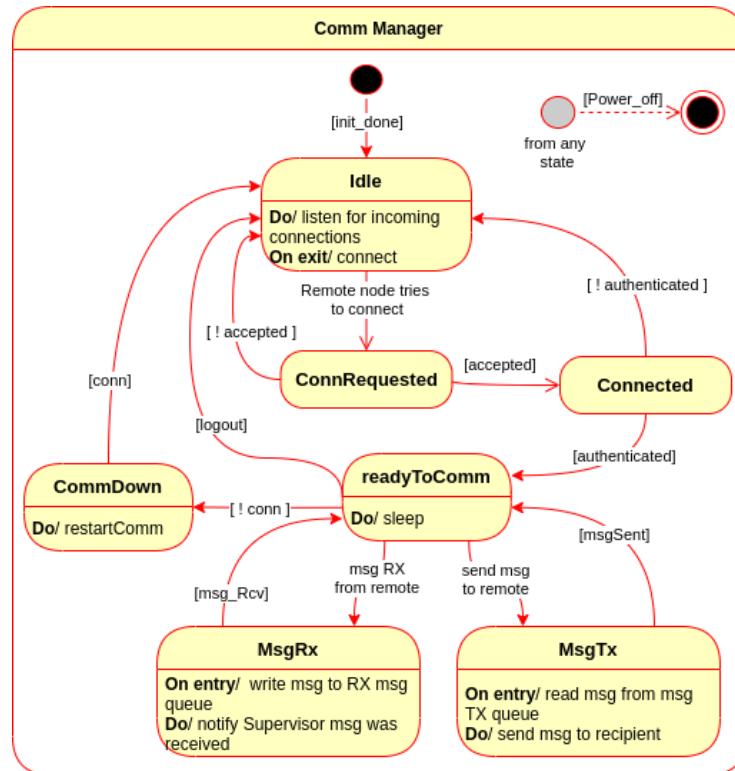


Figure 2.34.: State machine diagram: local system – Comm Manager

## 2.4. Subsystem decomposition

### 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.

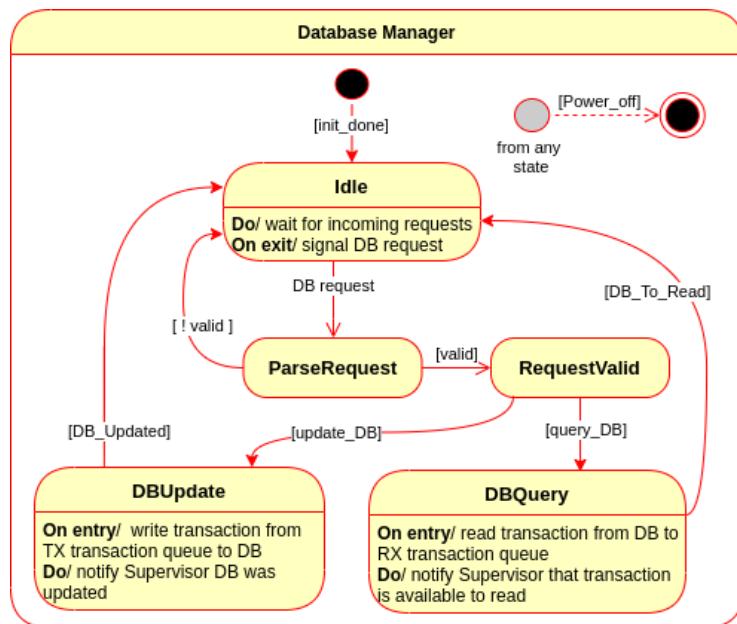


Figure 2.35.: State machine diagram: local system – DB Manager

### 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.

## 2.4. Subsystem decomposition

---

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).

## 2.4. Subsystem decomposition

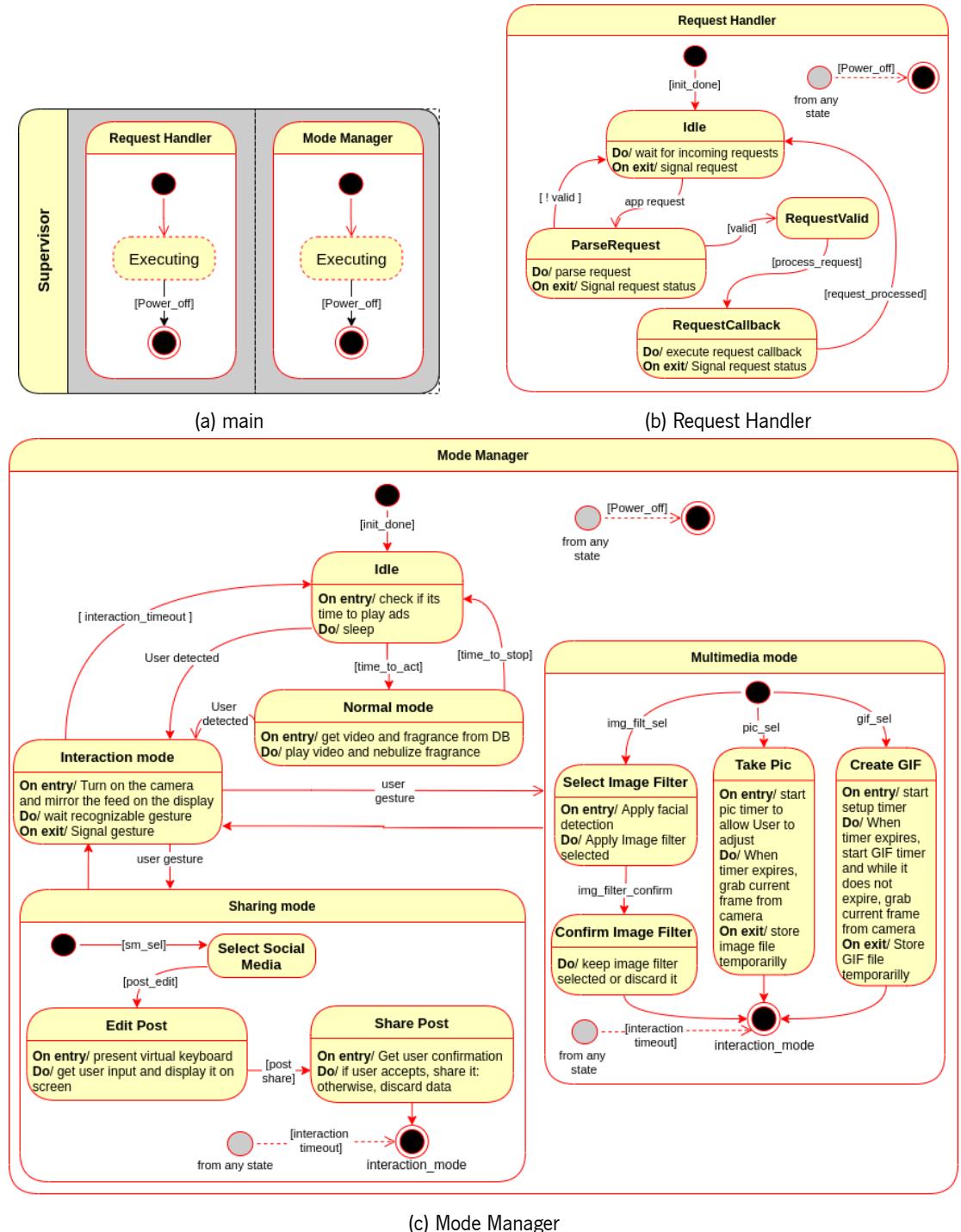


Figure 2.36.: State machine diagram: local system – Supervisor

### Multimedia mode

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

## 2.4. Subsystem decomposition

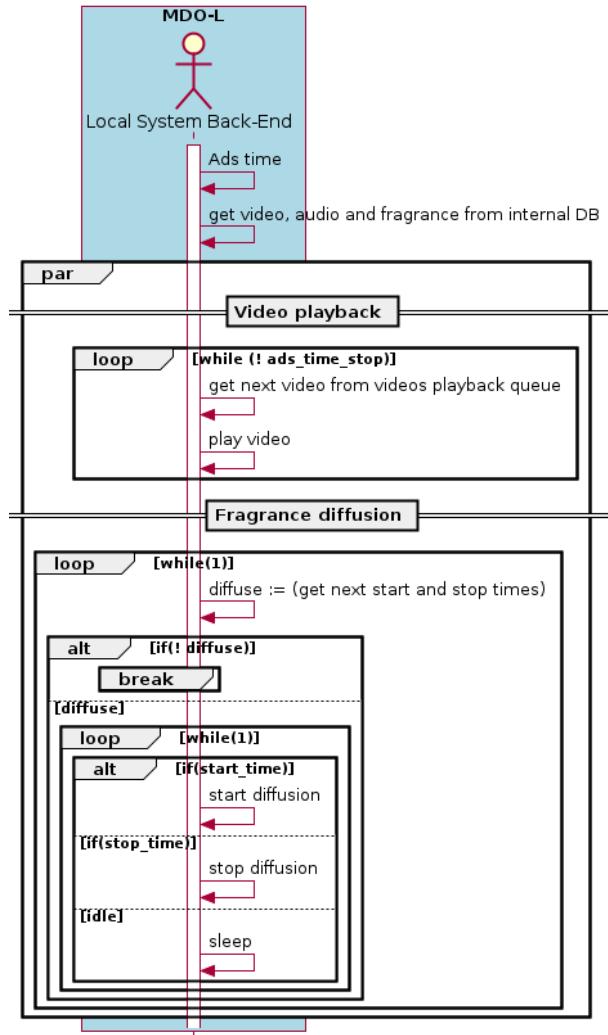


Figure 2.37.: Sequence diagram: local system – Normal mode

- **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, if 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.

## 2.4. Subsystem decomposition

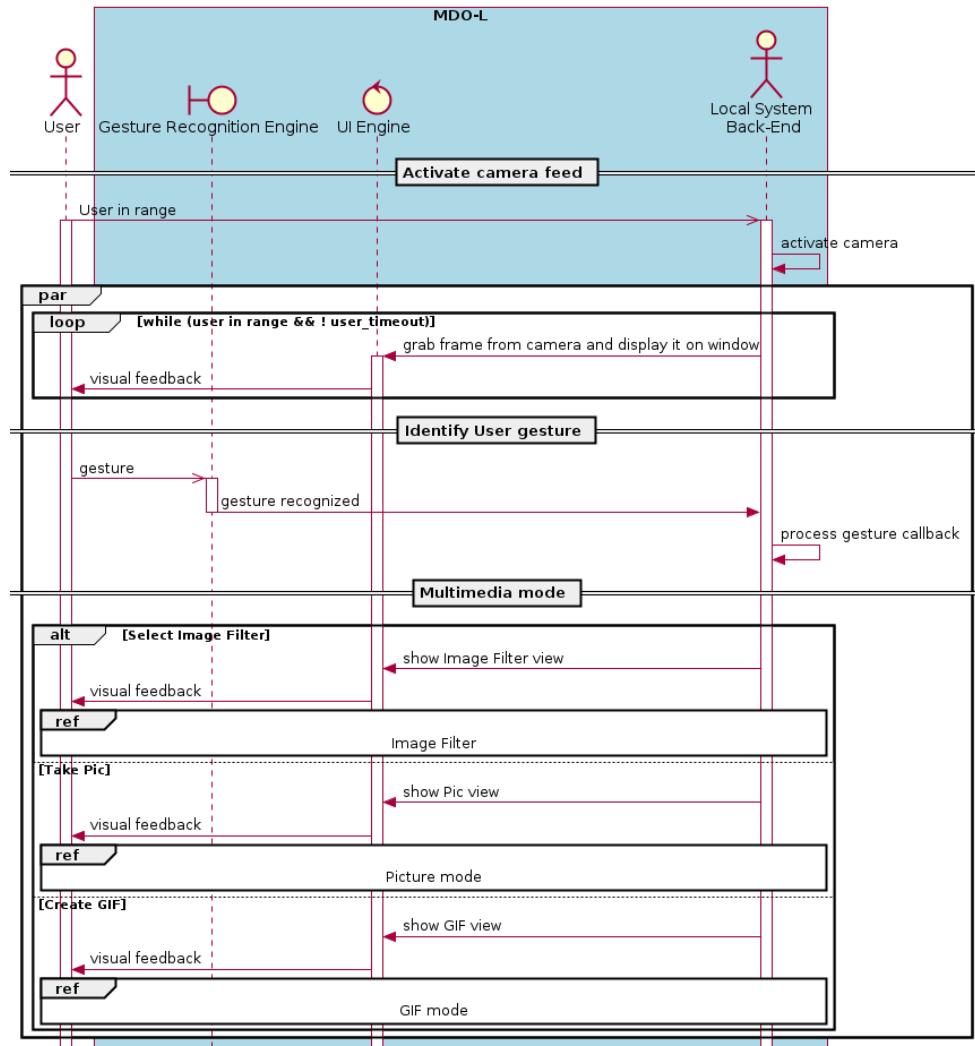


Figure 2.38.: Sequence diagram: local system – Interaction mode

- **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-

## 2.4. Subsystem decomposition

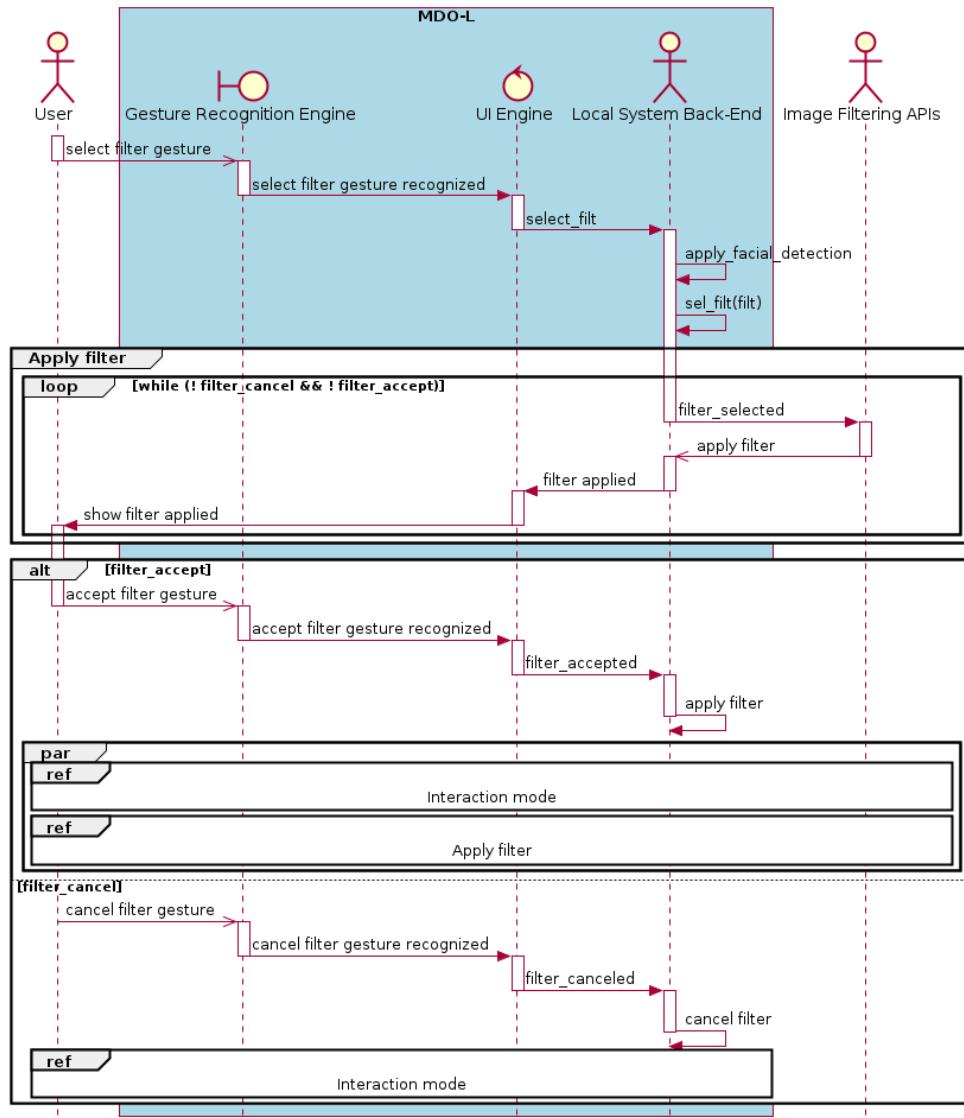


Figure 2.39.: Sequence diagram: local system – Multimedia mode (select image filter)

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

## 2.5. Budget estimation

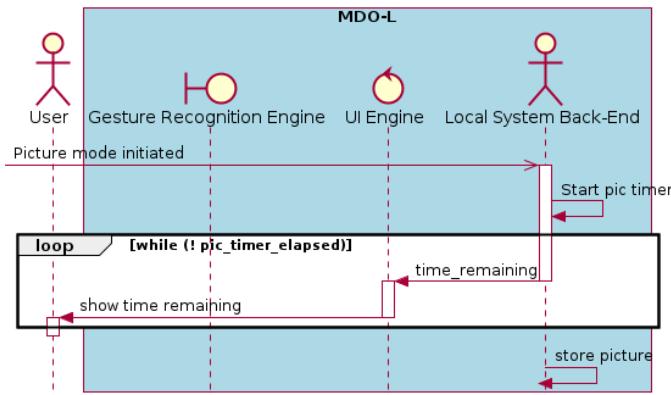


Figure 2.40.: Sequence diagram: local system – Multimedia mode (take picture)

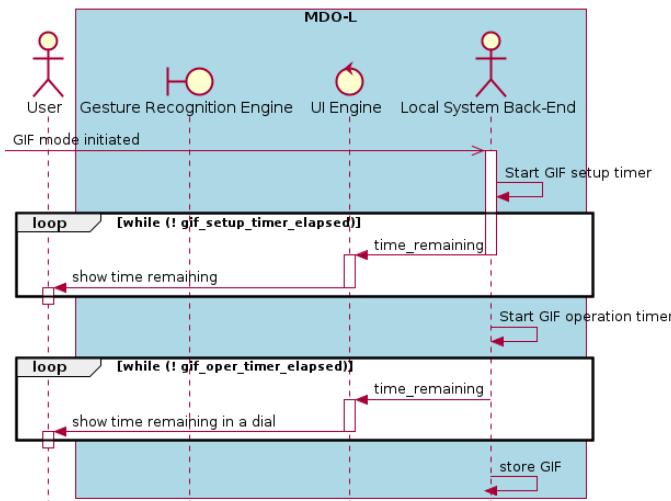


Figure 2.41.: Sequence diagram: local system – Multimedia mode (create GIF)

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:

## 2.5. Budget estimation

Table 2.4.: Budget estimation

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

\* 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

## **2.5. Budget estimation**

---

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.

## 2.5. Budget estimation

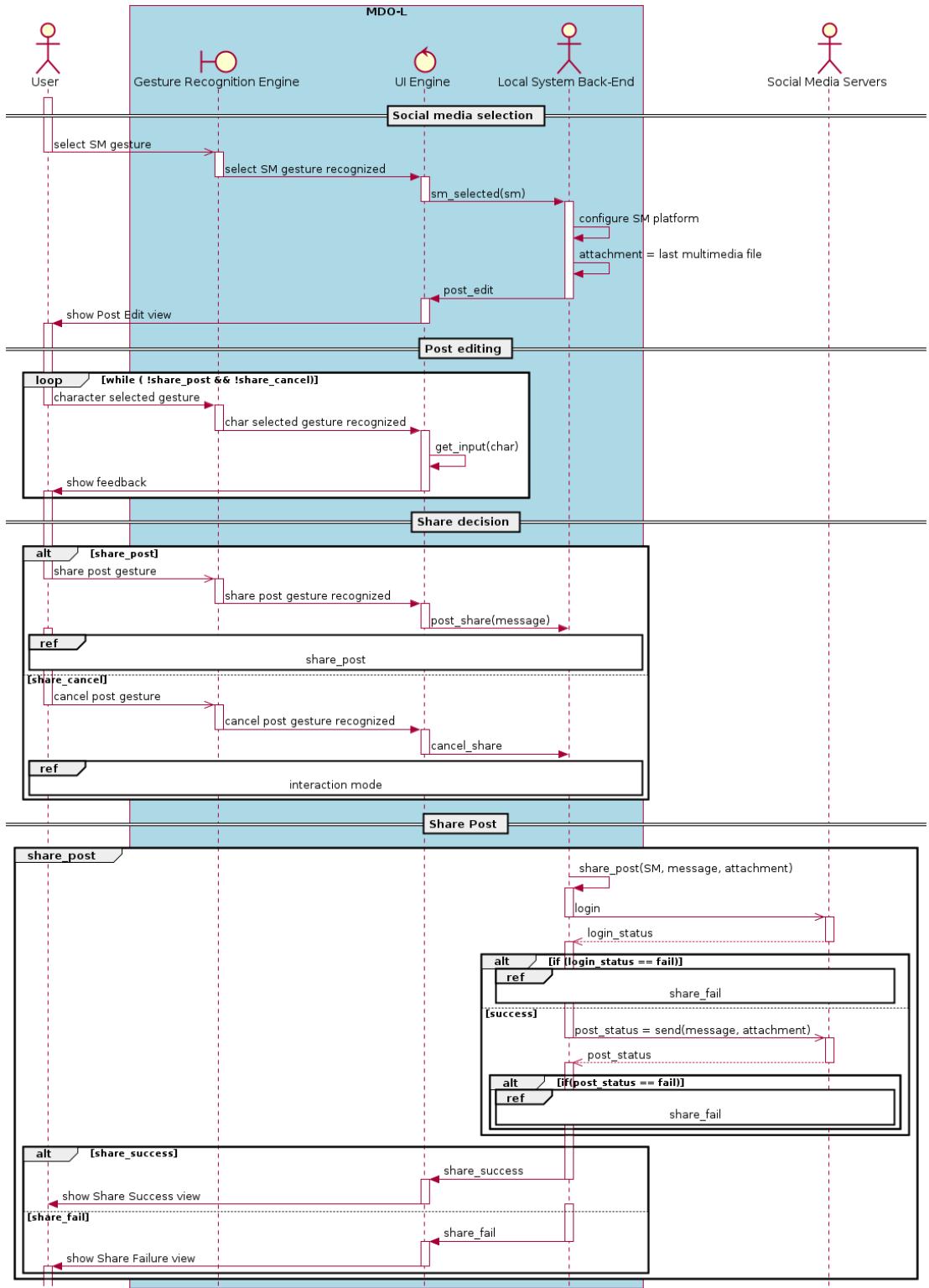


Figure 2.42.: Sequence diagram: local system – Sharing mode

# **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.

### 3.1. Project methodology

---

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]).

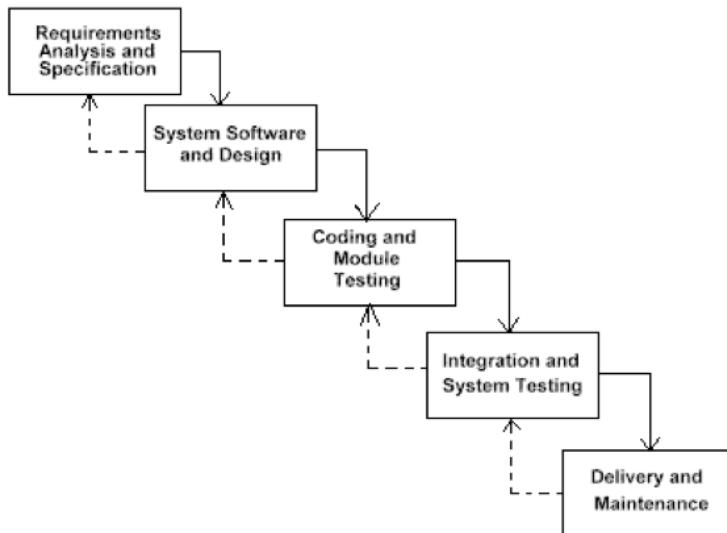


Figure 3.1.: Waterfall model diagram

### 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.

### 3.3. Threads versus Processes

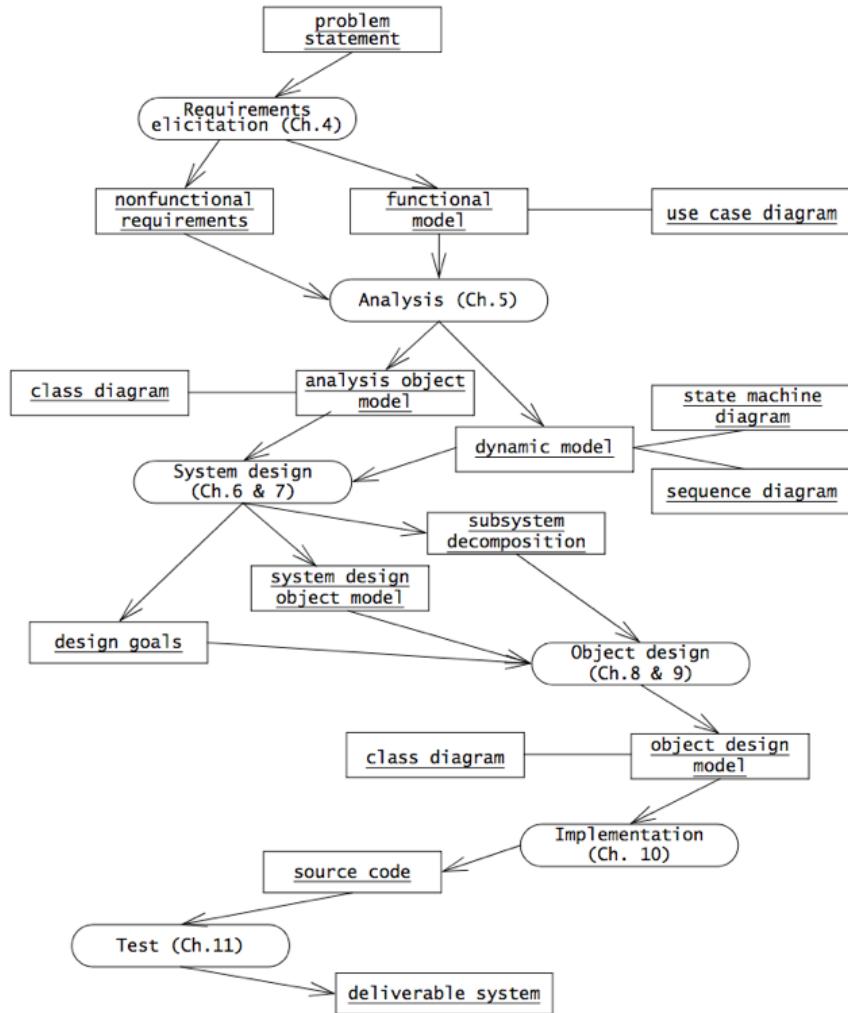


Figure 3.2.: An overview of the object-oriented software engineering development and their products. This diagram depicts only logical dependencies among work products (withdrawn from [11])

## 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.

### 3.3. Threads versus Processes

---

- 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]:

```
1 #include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
3                 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()`.

```
1 #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].

```
1 #include <pthread.h>
2 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 a 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].

```
1 #include <pthread.h>
2 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 spectrum 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.

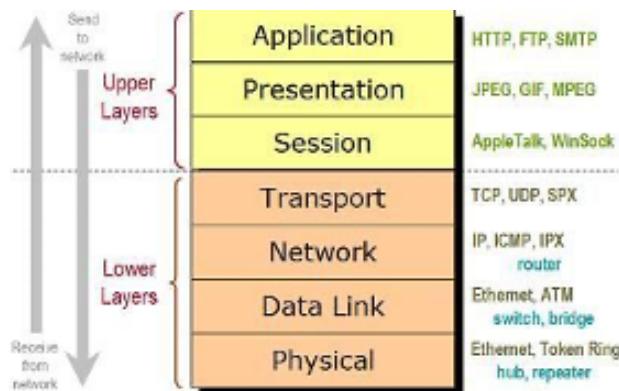


Figure 3.3.: OSI model

### 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 incoming 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.

### 3.4. Communications

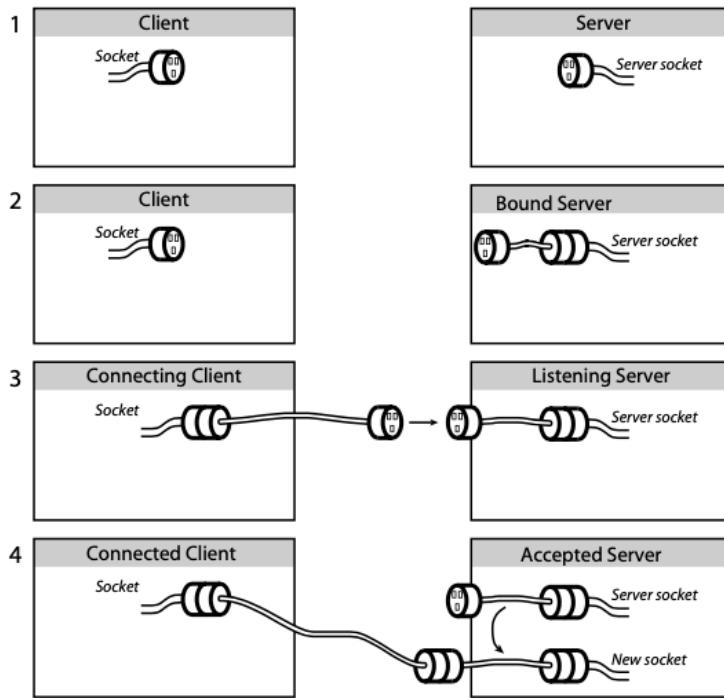


Figure 3.4.: Steps to obtain a connected socket (withdrawn from [16])

- 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.

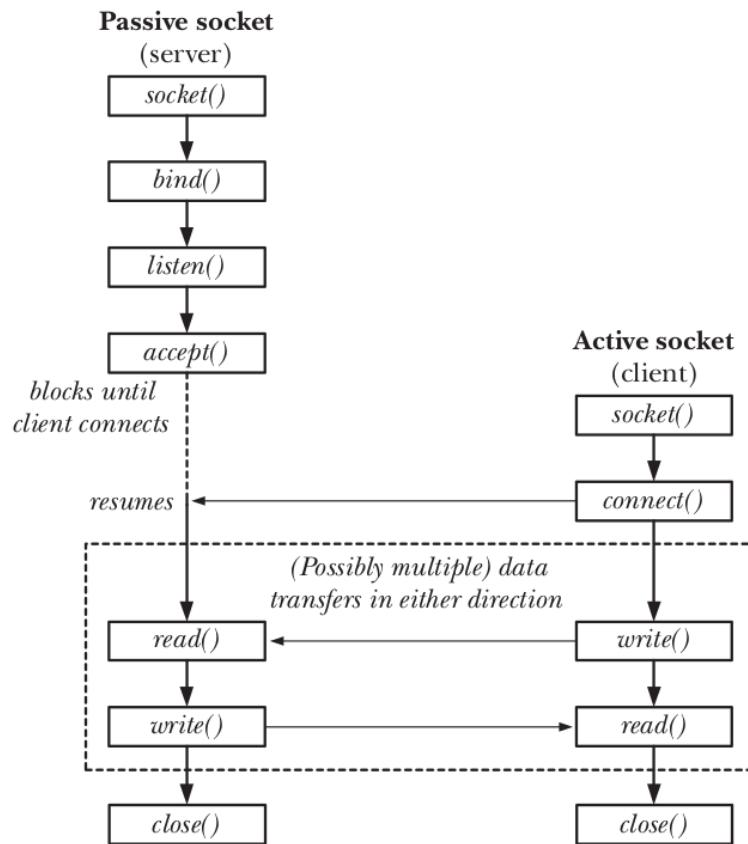


Figure 3.5.: Overview of UNIX system calls with sockets implementing a server/client paradigm (withdrawn from [13])

## 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 privileged 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

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

10 if (pid > 0) // parent process (exit)
    exit(EXIT_SUCCESS);
12 // 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.

```
1 pid_t sid;
2 sid = setsid(); // create a new session
3 if (sid < 0) {
4     ERROR("setsid failure");
5     exit(EXIT_FAILURE);
6 }
```

- 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
2 if (chdir("/") < 0) {
3     ERROR("chdir failure");
4     exit(EXIT_FAILURE);
5 }

// continue executing the child process
8 ...
```

## 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.

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

3 // continue executing the child process
4 ...
5 ...
```

### 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.

```
1 // close unneeded file descriptors
2 close(STDIN_FILENO);
3 close(STDOUT_FILENO);
4 close(STDERR_FILENO);

5 // continue executing the child process
6 ...
7 ...
```

### 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:

```
1 #include <syslog.h>

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

### 3.5.4. Communicating with a Daemon

To communicate with a daemon, you send 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.

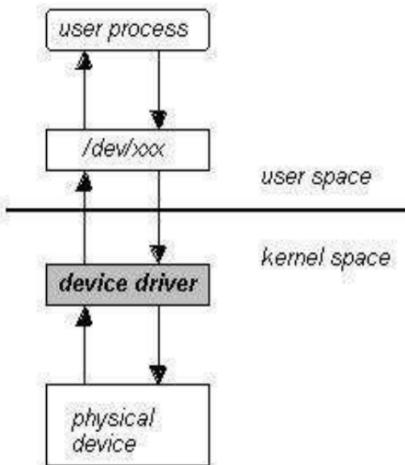


Figure 3.6.: Example of the usage of a device driver (withdrawn from [21])

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.

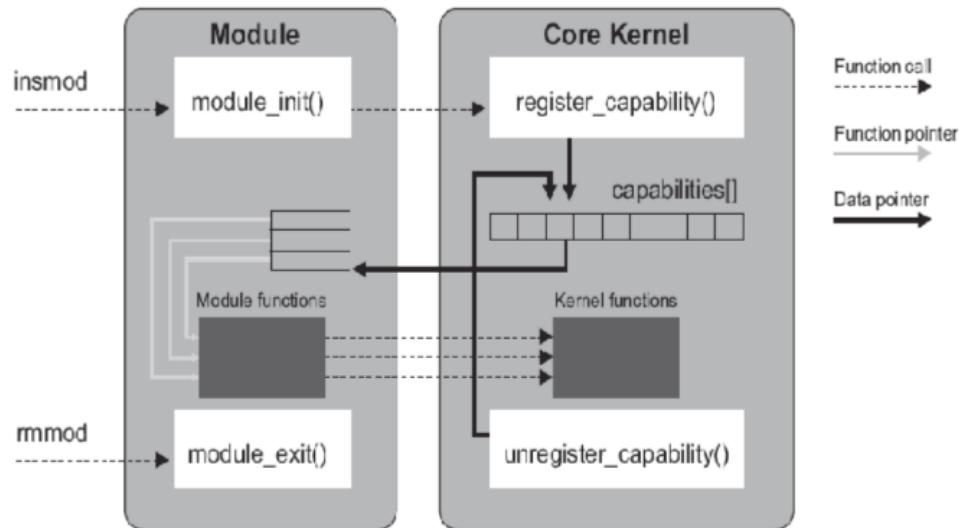


Figure 3.7.: Application accessing to module services (withdrawn from [21])

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.

### 3.7. Build system and Makefiles

---

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 exist.

#### 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`,

### 3.7. Build system and Makefiles

---

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
```

### 3.7. Build system and Makefiles

---

```
# -----
# ----- make install -----
# Automating the program's installation with the install: rule
#   - install: Places the binary or executable inside a given folder , e.g., /bin
46 #   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
51 #   - and we add the rule install:all

# PROJ name
PROJ=Electric -gym

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

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

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

# Databases
DB = $(wildcard *.db)

71 SRC := $(wildcard *.c)

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

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

### 3.7. Build system and Makefiles

---

```
# Building executable (w/ linking)
86 $(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
91     @echo "Creating executable"
    $(CC) -o $@ $^ ${LIBS}

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

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

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

111 .PHONY: clean mrproper clean-all doc
clean-all: clean mrproper
clean:
    @- $(RM) $(OBJ)
    # @- $(RM) $(DB)
116 mrproper: clean
    @$(RM) $(PROJ)
    # Documentation
doc:
    @echo "Generating documentation"
121 @$(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 a lot of intermediate files.

## 3.8. Source code documentation

Source code documentation is critical for software maintenance, especially on large development teams. As software applications grow in size and evolve, several modifications from base source code arise, possibly 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 [28]:

- 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 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 require 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 [28]:

- 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 [28]:

- 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 Mac OS, Linux and Windows platforms [29].

Doxygen supports the user in two ways [29]:

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 [30]. 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 [30]:

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:

```
1 /**
 * Javadoc style ... text ...
3 */
```

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

```
1 /*!
 * Qt style ... text ...
3 */
```

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:

```
1 ///
// slash style ... text ...
3 ///
```

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.

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

### 3.8. Source code documentation

---

- 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.

```
1  /**
2   * @file App.h
3   * @author Jose Pires
4   * @date 12 Jan 2019
5   *
6   * @brief App module containing the application logic
7   *
8   * It contains only two public functions:
9   * 1. Init - to initialize the app's memory
10  *    2. Exec - contains all application logic
11  */
```

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.

```
1 /**
2  * @file App.h
3  * @author Jose Pires
4  * @date 12 Jan 2019
5  *
6  * @brief App module containing the application logic
7  *
8  * It contains only two public functions:
9  * 1. Init - to initialize the app's memory
10 *    2. Exec - contains all application logic
11 */

13 #ifndef App_H
14 #define App_H

/**
```

### 3.8. Source code documentation

---

```
17 * @brief opaque pointer to struct App_T.  
18 * It hides the implementation details (allows modularity)  
19 */  
20  
21 typedef struct App_T *App_T;  
  
22 /* ===== External accessable functions ===== */  
23  
24 /**  
25 * allocate dynamic memory and initialize App  
26 * @return initialized App_T  
27 */  
28  
29 App_T App_init();  
  
30  
31 /**  
32 * @brief App controller: handles all application's logic  
33 * @param app - an initialized application instance  
34 * @return an integer signaling the execution state  
35 *  
36 * Its the execution loop controlling the FSM machine behind the application's  
37 * logic. It starts in S_Login state. When end user quits the application,  
38 * the databases are saved, for future restoration at subsequent program  
39 * executions.  
40 */  
41 int App_exec(App_T app);  
42 /* ===== */  
43 #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 /**  
2 * @file App.c  
3 * @author Jose Pires  
4 * @date 12 Jan 2019  
5 *  
6 * @brief App's module implementation  
7 */
```

```

9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <assert.h>
12 #include <strings.h>
13 #include "App.h"
14 #include "list.h"
15 #include "User.h"
16 #include "Activity.h"
17 #include "Menu.h"
18 #include "Pack.h"
19 #include "Database.h"
20 #include "m-utils.h"

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

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

38 /**
39 * @brief Constants for the App's states. Used in FSM management.
40 */
41 enum App_state{
42     S_Login , /*< Login state */
43     S_Gerente , /*< Manager state */
44     S_Func , /*< Employee state */
45     S_Cliente , /*< Client state */
46     S_Manage_Cli , /*< Manage Client state */
47     S_Manage_Act , /*< Manage Activity state */
48     S_Manage_Pack , /*< Manage Pack state */
49 }

```

### 3.8. Source code documentation

---

```
        S_Edit_User ,/**< Edit User state */
53     S_Edit_Act ,/**< Edit Activity state */
54     S_Edit_Pack ,/**< Edit Pack state */
55     S_Activities ,/**< Client Activities state */
56     S_Logout ,/**< Logout state */
57     S_Quit/**< Quit state */
58 };

59 /**
60  * @brief App's struct: contains the relevant data members
61  */
62 struct App_T
63 {
64     List_T menus; /**< Menus list */
65     List_T users; /**< Users list */
66     List_T activities; /**< Activities list */
67     List_T packs; /**< List of available packs */
68     User_T cur_user; /**< Current user */
69     enum App_state state; /**< App's current state */
70     enum App_state prev_state; /**< App's previous state */
71     void * userdata; /**< ptr to generic data (used in App_state_functions) */
72     Database_T db_user; /**< Users database */
73     Database_T db_act; /**< Activities database */
74     Database_T db_pack; /**< Packs database */
75 };

76 /**
77  * @brief Allocates memory for an App's instance
78  * @return initialized memory for App
79  *
80  * It is checked by assert to determine if memory was allocated.
81  * If assertion is valid, returns a valid memory address
82  */
83 static App_T App_new()
84 {
85     App_T app = malloc(sizeof(*app));
86     assert(app);
87     return app;
88 }
```

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 3.4 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.
#
5 # 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:
10 # TAG = value [value, ...]
    # For lists, items can also be appended using:
    # TAG += value [value, ...]
```

### 3.8. Source code documentation

---

```
# Values that contain spaces should be placed between quotes (" \").  
  
15 # -----  
# Project related configuration options  
# -----  
  
# This tag specifies the encoding used for all characters in the configuration  
20 # 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  
30 # 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.  
  
40 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  
50 # pixels and the maximum width should not exceed 200 pixels. Doxygen will copy  
# the logo to the output directory.  
  
PROJECT_LOGO           =  
  
55 # The OUTPUT_DIRECTORY tag is used to specify the (relative or absolute) path
```

### 3.8. Source code documentation

---

```
# 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.

60 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.

65 # 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 ,
70 # Macedonian , Norwegian , Persian (Farsi) , Polish , Portuguese , Romanian , Russian ,
# Serbian , Serbian-Cyrillic , Slovak , Slovene , Spanish , Swedish , Turkish ,
# Ukrainian and Vietnamese.

# The default value is: English.

75 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
85 # 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 compatibility issues.

# The default value is: YES.

MARKDOWN_SUPPORT        = YES

# -----
# Build related configuration options

95 # -----
```

### 3.8. Source code documentation

---

```
# class members and static file members will be hidden unless the
100 # 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.

105 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.
115 # 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
120 # included in the documentation.
# The default value is: NO.

EXTRACT_STATIC          = YES

125 # 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
135 # -----

# 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.
140 # The default value is: NO.
```

### 3.8. Source code documentation

---

```
QUIET           = NO

# The WARNINGS tag can be used to turn on/off the warning messages that are
145 # 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
155 # will automatically be disabled.
# The default value is: YES.

WARN_IF_UNDOCUMENTED = YES

160 # 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
170 # -----


# 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
175 # spaces. See also FILE_PATTERNS and EXTENSION_MAPPING
# Note: If this tag is empty the current directory is searched.

INPUT           = /Users/zemiguel/Trab/code/

180 # 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.
```

### 3.8. Source code documentation

---

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

INPUT_ENCODING      = UTF-8

# If the value of the INPUT tag contains directories , you can use the
190 # 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
200 # (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

205 # -----
# Configuration options related to source browsing
# -----


# If the SOURCE_BROWSER tag is set to YES then a list of source files will be
210 # 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
220 # -----


# 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.
225 # 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.

230 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.

255 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

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

### 3.8. Source code documentation

---

```
# 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  
275 # 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.  
  
280 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.  
  
290 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  
300 # powerful graphs.  
# The default value is: YES.  
  
CLASS_DIAGRAMS      = YES  
  
305 # 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  
310 # default search path.  
  
MSCGEN_PATH          =
```

### 3.8. Source code documentation

---

```
# You can include diagrams made with dia in doxygen documentation. Doxygen will
315 # 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.

325 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
330 # 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.
340 # The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.

CLASS_GRAPH   = YES

345 # 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.
350 # 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
355 # groups, showing the direct groups dependencies.
# The default value is: YES.
```

### 3.8. Source code documentation

---

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

GROUP_GRAPHHS      = 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.
365 # 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
370 # 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.

375 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
380 # 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.
390 # The default value is: YES.
# This tag requires that the tag HAVE_DOT is set to YES.

INCLUDED_BY_GRAPH      = YES

395 # 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.
```

### 3.9. Scenting technologies

---

```
400 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
405 # 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/)).
415 # 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
420 # 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

### 3.9. Scenting technologies

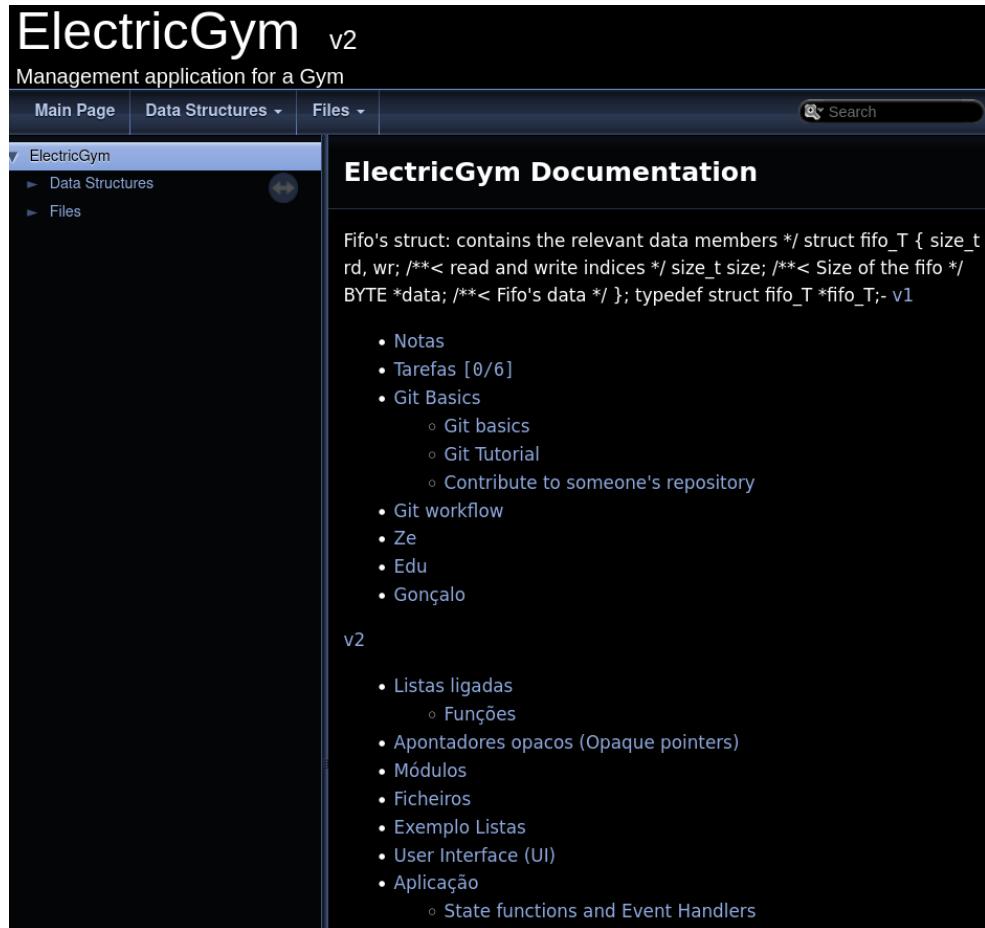


Figure 3.9.: Doxygen output: readme file

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 [31]:

- 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

### 3.9. Scenting technologies

Structure	Description
<code>Act_T</code>	Activity's structure: contains the relevant data members
<code>App_T</code>	App's struct: contains the relevant data members
<code>Database_T</code>	Database's struct: contains the relevant data members
<code>Fifo_T</code>	Fifo's struct: contains the relevant data members
<code>List_T</code>	List's structure: generic container for doubly linked list
<code>Menu_T</code>	Menu's structure: contains the relevant data attributes
<code>Node_T</code>	Structure Node: atomic unit that composes the list The list is doubly-linked, i.e., with pointers to prev and next elements and with a pointer to generic data
<code>Pack_T</code>	Pack's structure: contains the relevant data members
<code>User_T</code>	User's structure: contains the relevant data members

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 [32]:

- 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

### 3.9. Scenting technologies

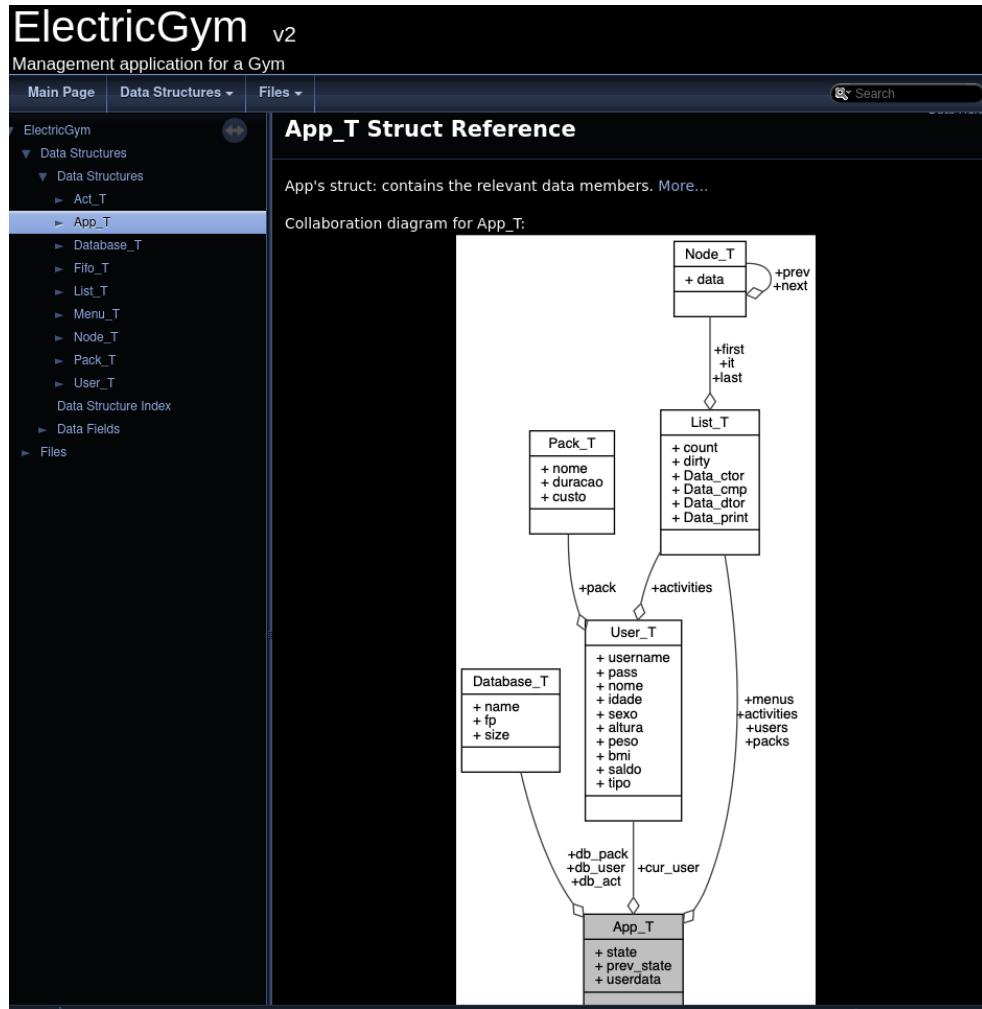


Figure 3.11.: Doxygen output: collaboration graph for a structure

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.

### 3.9. Scenting technologies

**ElectricGym v2**  
Management application for a Gym

Main Page Data Structures Files Search TypeDefs | Functions

**Activity.h File Reference**

Interface for the module Activity. More...

```
#include "m-utils.h"
#include <stdio.h>
#include <stdbool.h>
#include "User.h"
#include "fifo.h"
```

Include dependency graph for Activity.h:

```
graph TD
    ActivityH[Activity.h] --> mutilsH[m-utils.h]
    ActivityH --> UserH[User.h]
    ActivityH --> fifoH[fifo.h]
    ActivityH --> stdioH[stdio.h]
    mutilsH --> stdargH[stdarg.h]
    mutilsH --> stdlibH[stdlib.h]
    UserH --> stdboolH[stdbool.h]
    stdioH --> stdboolH
    UserH <--> stdboolH
```

This graph shows which files directly or indirectly include this file:

```
graph TD
    ActivityH[Activity.h] <--> ActivityC[Activity.c]
    ActivityH <--> AppC[App.c]
    ActivityH <--> UserC[User.c]
```

Go to the source code of this file.

**TypeDefs**

```
typedef struct Act_T * Act_T
```

opaque pointer to struct **Act\_T**. It hides the implementation details (allows modularity)

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

## 3.9.2. Ultrasonic diffusion

Ultrasonic diffusion uses high frequency stimuli to break down water molecules into droplets, producing mist, diffusing the oils into the air. The high frequency stimuli is above 20 kHz, the upper threshold for audible human hearing — thus the ultrasonic naming — and it's accomplished using micro-mesh piezoelectric transducers.

Piezoelectric transducers are reciprocating transducers as they:

- respond to electric stimuli by generating mechanical displacement which, in turn, produces waves — inverse-piezoelectricity.
- respond to mechanical displacement (applied force) by generating an electrical voltage signal — piezoelectricity.

In the present case, one is more interested in the first phenomena. Piezoelectric transducers have a resonant frequency, which means they will go into a natural oscillation state, amplifying oscillations.

### 3.9. Scenting technologies



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

Thus, stimulating the piezoelectric actuator with a AC signal at the resonant frequency produces a strong mechanical oscillation, generating waves. For small resonant frequencies, the liquid, e.g. water, can easily follow the mechanical oscillation produced. However, when the resonant frequency is high enough (hundreds of kHz or MHz), the water particles cannot follow the oscillating surface, thus creating momentary vacuum, due to the negative amplitudes, which therefore creates air bubbles. Then, on positive amplitudes these air bubbles are pushed across the surface, catapulting water droplets into the air, quickly dissipating and turning into vapor form.

Fig. 3.15 illustrates a type of piezoelectric transducer for ultrasonic diffusion — the micro-porous mesh. It is comprised of [31]:

- rubber gasket: used to isolate electric conduction from other conducting materials and as cushion against vibration;

### 3.9. Scenting technologies



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

- metal substrate: it has a micro-porous metal mesh in the center with a high number of trumpet-shaped cylinder micro-pores, in which the upper cylindrical surface is smaller than the bottom.
- ring-shaped piezoelectric plate: a contact is attached to the piezoelectric plate, so that the power wire and ground wire can be connected between the piezoelectric plate and the metal substrate, enabling its electrical stimulation.

This micro-porous piezoelectric transducer is driven by an AC signal at a frequency of around 113 kHz and converts electric energy into kinetic energy due to inverse-piezoelectricity [31]. The metal substrate vibrates along with the vibration of the ring-shaped piezoelectric plate, and the mesh in the center of the metal substrate smashes the liquid beneath the transducer. Some liquid flows through those micro-pores and is emitted in micro-droplet form, due to the momentary vacuum created.

The rationale behind the micro-porous piezoelectric is due to its versatility regarding voltage supply, small size, easy control, low power consumption and variety of fluids that it can diffuse. A list of micro-porous piezoelectric film properties are listed below [31]:

1. Diameter: 13.8 mm.

### 3.9. Scenting technologies

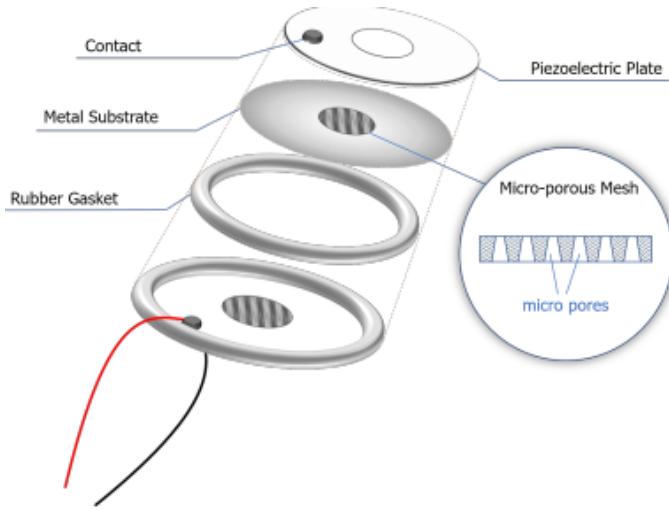


Figure 3.15.: Micro-porous mesh piezoelectric transducer for ultrasonic diffusion – withdrawn from [31]

2. Low driving voltage: 3–12 V.
3. High conversion efficiency, spray volume.
4. Exit aperture is very small 4  $\mu\text{m}$ .
5. Frequency:  $113 \text{ kHz} \pm 5 \text{ kHz}$ .
6. Capacitance:  $2700 \text{ pF} \pm 15\%$ .
7. Power: 1.5–2.0 W.
8. Spray volume 30 ml/h.
9. Can atomize essential oils, perfume, water based perfumes or even mixture of the mentioned materials.
10. life of more than 3000 hours.

Abid et al. [33] proposed a novel olfactory displays' scent dispersing module based on this type of transducer (Fig. 3.16). It contains a refillable fragrance reservoir, a cotton core and the housing for the micro-porous piezoelectric film. The fragrance ascends to the cotton core via capillary effect, impregnating it. When the micro-porous electric film is stimulated the fragrance in the cotton core is diffused through the aforementioned effect.

#### Driving circuit

There are several commercially available PCBs to drive micro-porous piezoelectric transducers. Fig. 3.17 illustrates a commercial PCB to drive piezoelectric transducer of 113 kHz resonance frequency.

As aforementioned, piezoelectric transducers for diffusion require an AC signal at the resonance frequency. Fig. 3.18 illustrates a possible driving circuit for these transducers. A 555-timer Integrated Circuit

### 3.10. Computer vision

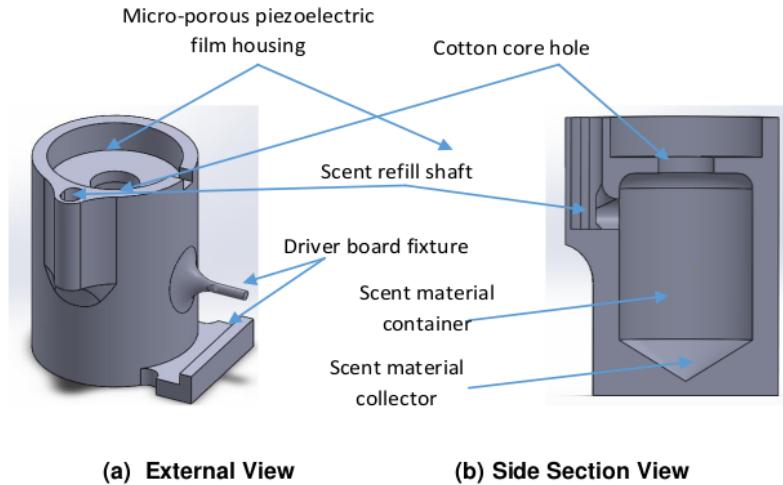


Figure 3.16.: Micro-porous mesh piezoelectric-based scent dispense module – withdrawn from [33]



Figure 3.17.: Commercial PCB for driving micro-porous mesh piezoelectric transducers [34]

(IC) is used in astable mode to generate a square wave at the resonance frequency that drives a gate of power Metal Oxide Semiconductor Field-Effect Transistor (MOSFET). This MOSFET which feeds an resonant circuit to generate an AC signal which stimulates the piezoelectric transducer, producing mechanical oscillations at the resonance frequency. This application requires a fast-switching power MOSFET, like the IRLZ44N, as the piezoelectric transducer consumes up to 300 mA of current.

## 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 [36].

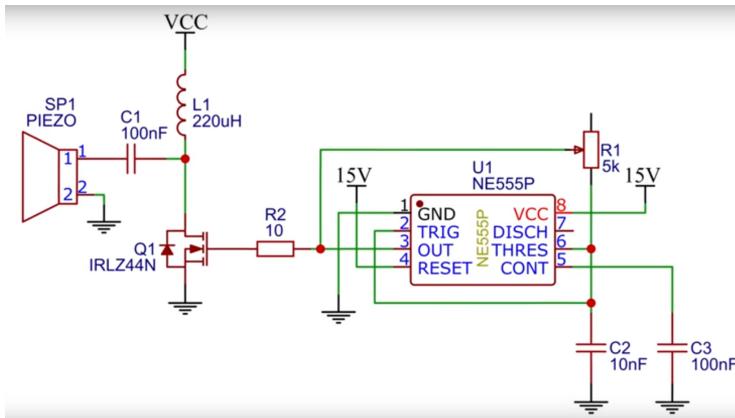


Figure 3.18.: Schematic of a driving circuit for micro-porous mesh piezoelectric transducers [35]

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 [36].

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.

### 3.10.1. Computer vision frameworks

There are several noteworthy computer vision frameworks, namely [37]:

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 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 backends 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 [38]: cascade based methods, part based methods, channel feature based methods, and neural network based methods. A detailed survey can be found in [39, 40].

The seminal work by Viola and Jones [41] 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 [42] 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 [43] 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) [44]. 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. [45] to solve pedestrian detection. Later on, Yang et al. [46] 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 [47, 48] 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 [49]:

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 a 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 [50]. 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 [51].

The algorithm can be explained in four stages [52]:

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. 3.19.
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. 3.20).
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. 3.21).

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. 3.22). 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.

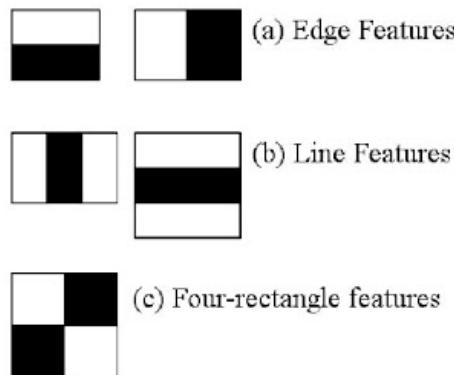


Figure 3.19.: Examples of Haar features (withdrawn from [51])

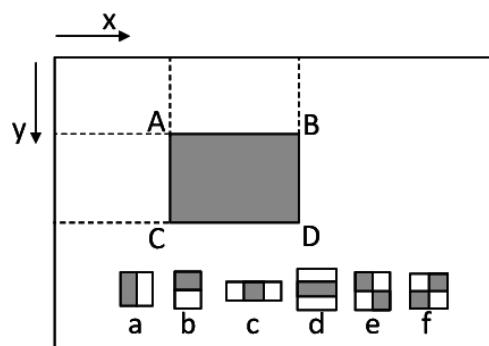


Figure 3.20.: Integral image creation illustration (withdrawn from [52])

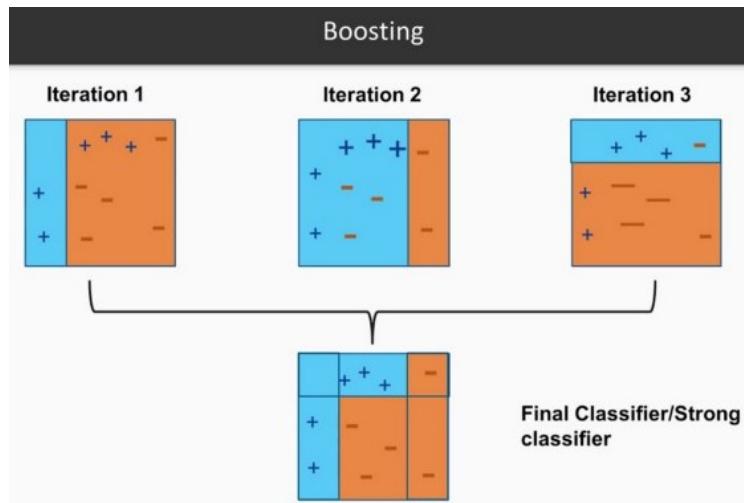


Figure 3.21.: Illustration of a boosting algorithm (withdrawn from [52])

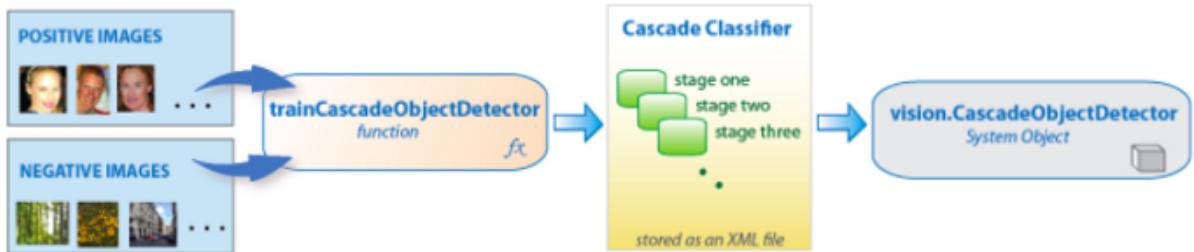


Figure 3.22.: Flowchart of a cascade classifiers (withdrawn from [52])

### 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 [53]. 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. [54].

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 [55]. 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 [54].

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. 3.23, comprising the following stages [54]:

- image frame acquisition: capture the human gesture image by computer, usually performed using a web camera or depth camera [56]. 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 [56]. 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 [54].
- 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 [56]. 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 [54].
- 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 [54].

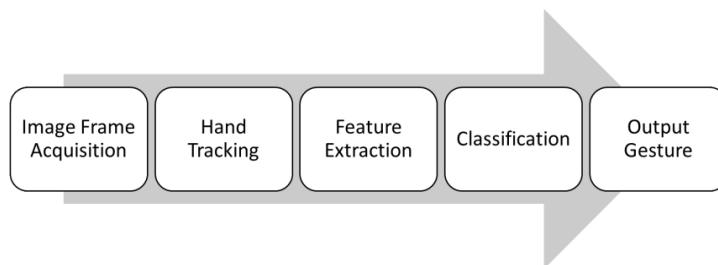


Figure 3.23.: Basic steps of hand gesture recognition (withdrawn from [54])

Fig. 3.24 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

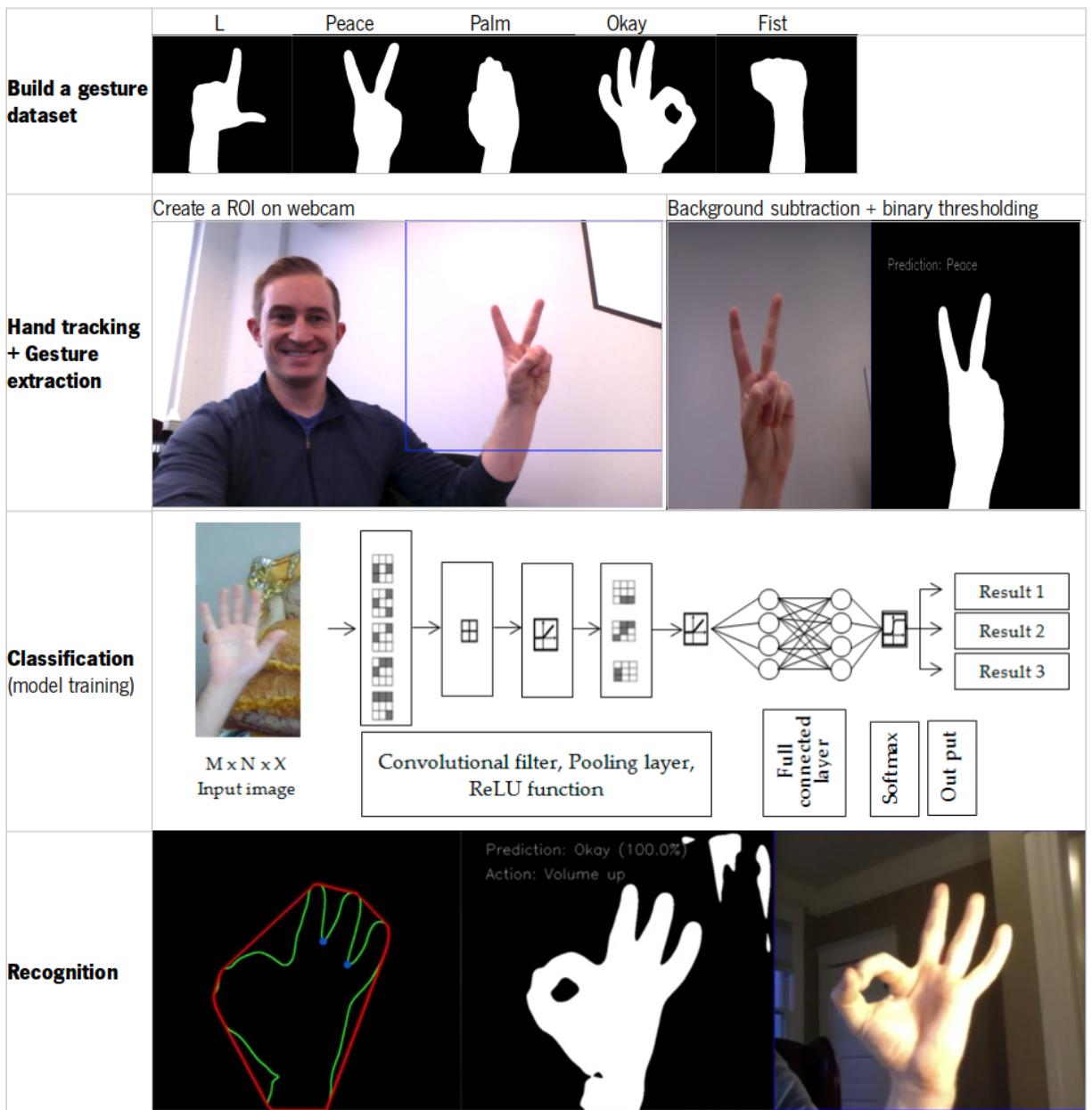


Figure 3.24.: Hand gesture recognition workflow illustrated: example – adapted from [57] and [58]

## 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 [59]:

- 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 Management System (DBMS) with relationship between tables (entities) and rows (entities' attributes). It follows the relational model, introduced by E.F. Codd in 1970 [59], 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 [59]. 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 [59]. 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 [59].

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 [59]:

- 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 [59]. 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 [59]:

```
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. An example instance of the student relation appears in Table 3.1 [59].

Table 3.1.: An instance of the students relation – withdrawn from [59]

<b>sid</b>	<b>name</b>	<b>login</b>	<b>age</b>	<b>gpa</b>
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 [59]. 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 [59].

### 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. 3.26, namely [59]:

- 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.

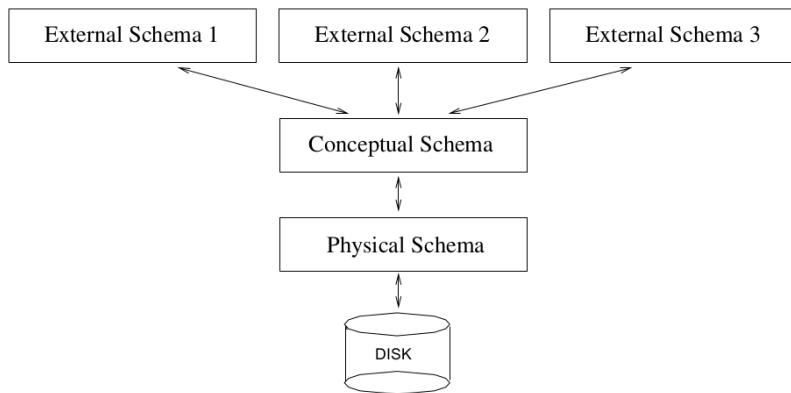


Figure 3.25.: Levels of abstraction in a DBMS (withdrawn from [59])

### 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 [59].

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 [59].

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 [59].

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 [59].

Summarizing, the main takeaways for DBMS support for concurrency control and recovery are [59]:

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

It is comprised of [59]:

- 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.).

### 3.11. RDBMS

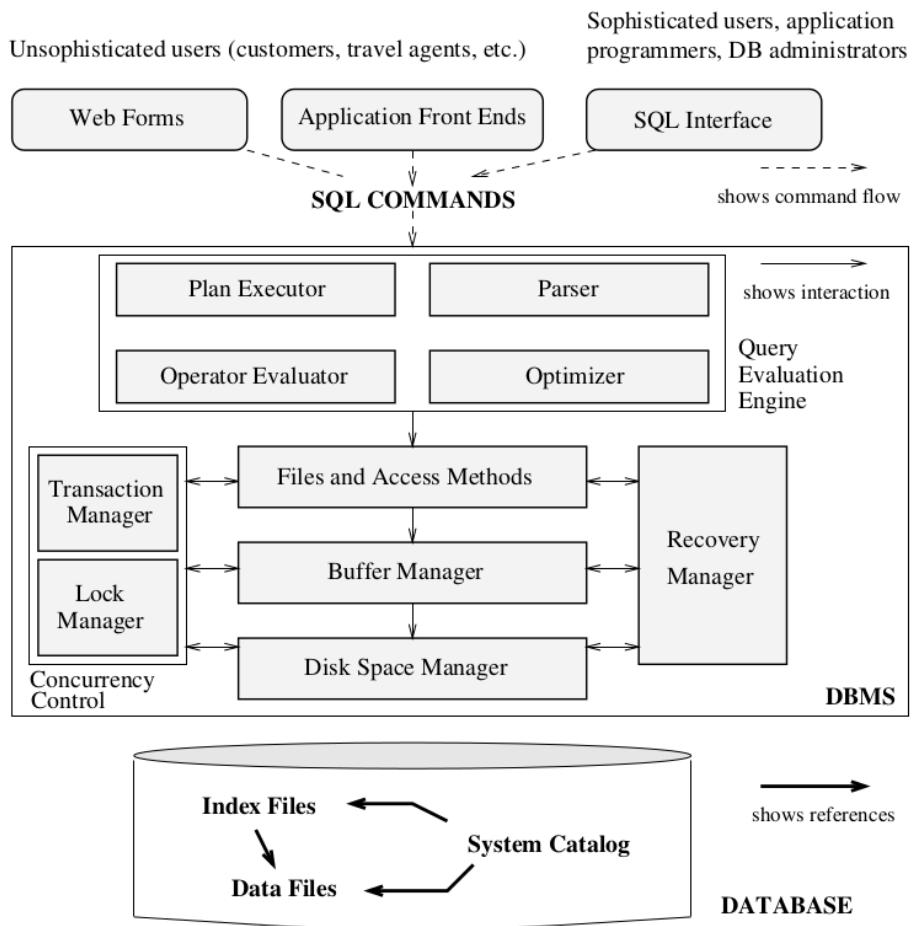


Figure 3.26.: Architecture of a DBMS (withdrawn from [59])

- **Concurrency control:** The Transaction Manager ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution of transactions. The Lock Manager keeps track of requests 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 [59]:

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 [59] (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 [59]:

- 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

### 3.11. RDBMS

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 [60]. 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. 3.27 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.

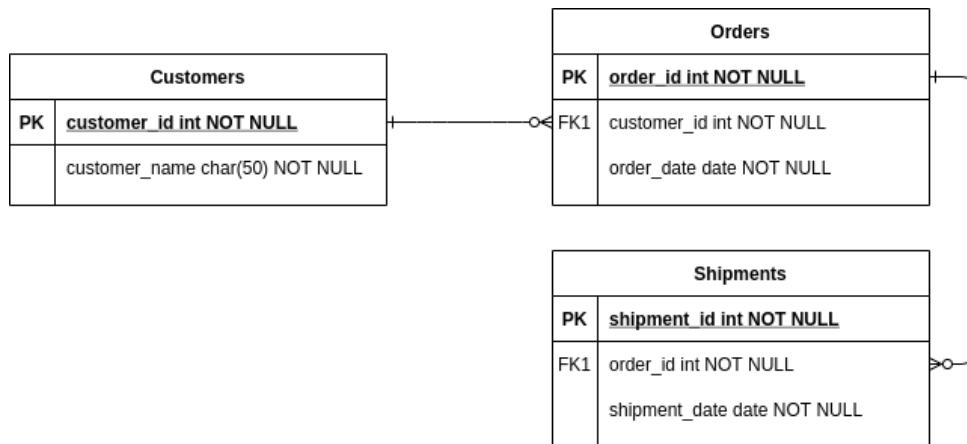


Figure 3.27.: Example of an ERD

### 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 [61]:

- **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** [62]: 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 syntax, it must be portable and conform to some basic standard. SQL complies well to these requirements [63].

SQL functions fit into two broad categories [63]:

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. 3.28).

2. It is a Data Manipulation Language (DML): SQL includes commands to insert, update, delete, and retrieve data within the database tables (see Fig. 3.29).

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 table's 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 3.28.: SQL data definition commands – withdrawn from [63]

### 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 [64]. The following interfaces are available: Java, Python, JavaScript, C++, C, C#, PHP, ODBC, NDB Cluster, MySQL Shell, and X DevAPI.

#### 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 [65]:

- 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.

### 3.11. RDBMS

---

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
<b>COMPARISON OPERATORS</b>	
=, <, >, <=, >=, <>	Used in conditional expressions
<b>LOGICAL OPERATORS</b>	
AND/OR/NOT	Used in conditional expressions
<b>SPECIAL OPERATORS</b>	
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
<b>AGGREGATE FUNCTIONS</b>	
COUNT	Used with SELECT to return mathematical summaries on columns
MIN	Returns the number of rows with non-null values for a given column
MAX	Returns the minimum attribute value found in a given column
SUM	Returns the maximum attribute value found in a given column
AVG	Returns the sum of all values for a given column
	Returns the average of all values for a given column

Figure 3.29.: SQL data manipulation commands – withdrawn from [63]

- 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 [66]. 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).

```

1  /* Standard C++ includes */
2 #include <stdlib.h>
3 #include <iostream>
4 /*
5     Include directly the different
6     headers from cppconn/ and mysql_driver.h + mysql_util.h
7     (and mysql_connection.h). This will reduce your build time!
8 */
9 #include "mysql_connection.h"
10
11 #include <cppconn/driver.h>
12 #include <cppconn/exception.h>
13 #include <cppconn/resultset.h>
14 #include <cppconn/statement.h>
15
16 using std::cout;
17 using std::endl;
18
19 int main(void)
20 {
21     cout << endl;
22     cout << "Running 'SELECT 'Hello World!' » AS _message '...'" << endl;
23
24     try {
25         sql::Driver *driver;
26         sql::Connection *con;
27         sql::Statement *stmt;
28         sql::ResultSet *res;
29
30         /* Create a connection */
31         driver = get_driver_instance();
32         con = driver->connect("tcp://127.0.0.1:3306", "root", "root");
33         /* Connect to the MySQL test database */
34         con->setSchema("test");
35
36         stmt = con->createStatement();
37         res = stmt->executeQuery("SELECT 'Hello World!' AS _message");
38         while (res->next()) {
39             cout << "\t... MySQL replies: ";
40             /* Access column data by alias or column name */
41             cout << res->getString("_message") << endl;
42             cout << "\t... MySQL says it again: ";
43         }
44     }
45 }
```

```

        /* Access column data by numeric offset , 1 is the first column */
44    cout << res->getString(1) << endl;
    }
46    delete res;
    delete stmt;
48    delete con;

50 } catch (sql::SQLException &e) {
    cout << "# ERR: SQLException in " << __FILE__;
52    cout << "(" << __FUNCTION__ << ") on line " >
        << __LINE__ << endl;
54    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 [66]

## 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**.

### 3.12.1. Different types of detection

There are several ways to do motion detection, through different sensors, such as [67]:

- 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 [67]. 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 [67].

- Microwave (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 [67].
- 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 [67].
- 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 [67].
  - Ultrasonic motion sensors: These sensors measure the reflections off of moving objects via pulses of ultrasonic waves [67].
  - Vibration motion sensors: This type of sensors detect small vibrations that people cause when they move through a room [67].
- 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 [67].
  - 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 [67].

#### 3.12.2. 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 to 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 other 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.

### **3.12.3. 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 [68].

As it can be seen in Fig. 3.30, 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 [69].

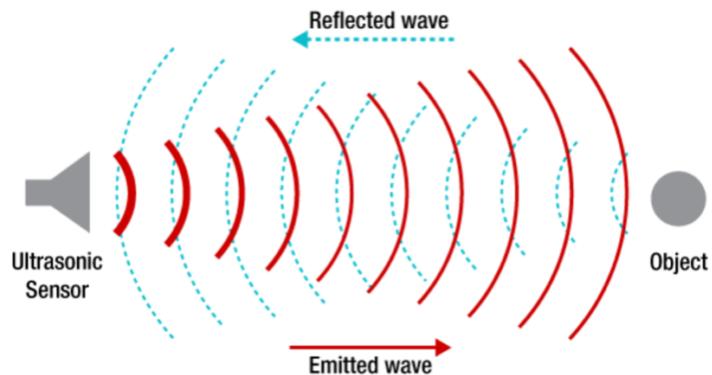


Figure 3.30.: Example of the behaviour of an ultrasonic sensor(withdrawn from [68])

## 3.13. Camera recording and codecs

In this project's local system (MDO-L), the Interaction Mode is composed by an interaction made through video between the machine and the user. To make possible that interaction through video, it is necessary to know how to record and show the frames captured by the camera using the Raspberry Pi.

The process of recording is just simply capture several frames in a specific period and then when all frames are joined together it creates a video. The more frames are captured per second, the more fluid the video becomes, that's why it is normal to talk about Frames per Second (FPS) when talking about video recording or playing.

### 3.13.1. Camera recording

It is possible to capture video from the raspberry using its camera module (V2) with the help of the OpenCV library. With a simple code in C++, it is possible to capture frames and show them in loop, in order to always refresh what the camera is recording. Listing 3.6 implements how to show what is being captured by the camera. Basically it is created a variable `frame` to read the frames of the camera and the `cap` refers to the camera (which in this case has 0 that represents the camera ID). Then, basically it is just needed to verify if the camera is opened and if so read the frame from the camera and show it. This last two steps are executed in a infinite loop until a key is pressed to stop the execution.

```

1 #include <stdio.h>
2 #include <iostream>
3 #include <opencv2/core.hpp>
4 #include <opencv2/highgui.hpp>
5 #include <opencv2/videoio/videoio_c.h>

6 using namespace cv;
7 using namespace std;

10 int main(int argc, char** argv)
11 {
12
13     Mat frame; //frame to capture
14     VideoCapture cap(0); //streaming from the camera
15         //in this case 0 is the ID of the camera
16
17     //verify if the camera opened
18     if (!cap.isOpened()){
19         cerr << "ERROR! Unable to open camera\n";
20         return -1;
21     }
22
23     while(true){
24
25         cap.read(frame);
26
27         imshow("Frame", frame);
28
29         if (waitKey(1) == 27){ //esc key
30             break;
31         }
32     }
33
34     cap.release();
35
36     return 0;
37 }
```

```

        }

        // Start acquisition
24    cout << "Start grabbing" << endl
        << "Press any key to terminate" << endl;

        // Infinite loop
28    for (;;)
    {
        // Read the frame from the camera
        cap.read(frame);

        // Avoid to show no frame
        if (frame.empty()) {
            cerr << "ERROR! Blank frame grabbed \n";
            break;
        }

        // Show the captured frame
        imshow("Live", frame);

        // If a key is pressed, exit the loop and stops de execution
42        if (waitKey (5) >= 0)
            break;
44    }
    return 0;
46}

```

Listing 3.6: Example of camera frames acquisition (withdrawn from [70])

### 3.13.2. Video files encoding (codecs) and formats

Video encoding formats, also called video file formats, are methods of optimizing digital video files for different platforms, programs, and devices. There are many different kinds of video encoding formats, but each is composed of two main parts: a **codec** and a **container** [71]. But firstly, it is necessary to know how video encoding works.

Video encoding is the process of turning uncompressed video input into a form that can be stored and played by a variety of devices and this involves two main processes: **compression** and **transcoding** [71].

- Compression: decreases the size of a video file so that it is more manageable. Without proper

### 3.13. Camera recording and codecs

---

compression, most files would be far too large to upload easily, load quickly, or play smoothly on users' devices.

- Transcoding: refers to the total audio and video conversion process from one video format to another. It ensures that a video file is compatible with the video player and/or platform it is using. Without transcoding, users would not be able to watch the video file at all.

On-demand streaming video is encoded so that it can be sent over the Internet and played on a variety of user devices. During live streaming, the video stream is segmented, compressed, and encoded in real time [71].

#### What is a codec?

A codec (coder/decoder) is a method for compressing and decompressing data so that it can be easily transported and received by different applications. Separate codecs are used to compress audio and video files, but they generally work in the same way [71]. Codecs encode files using either lossy compression or lossless compression. Lossy compression simplifies the data in a video file and only keeps the essential parts. This is why a video using lossy compression may look pixelated or "fuzzy" [71].

#### What is a container?

A container combines an encoded audio stream (audio codec), encoded video stream (video codec), and metadata in a single video file. The metadata tells the video player how to coordinate different audio and video codecs and may also provide additional elements, such as subtitles or alternate audio streams [71]. Each container supports a different range of video codecs. Some containers only work with a single type of codec and video player, which drastically limits playback options. Other containers are compatible with many types of video codecs and players [71].

#### Most common types of video formats

There are many types of video encoding formats and they are not compatible with the same platforms, browsers and devices. These are the most common video formats [71]:

- MP4: is a video file format created by the Motion Picture Expert Group. It compresses audio and video separately, which allows MP4 files to retain relatively high video quality after compression. Most browsers and iOS/Android devices are compatible with MP4 files.
- MOV: is a video file format created by Apple. Although it can run on both Mac OS and Windows OS, it is only compatible with QuickTime video players. It preserves video quality, but does not offer as

much file compression as other common video formats, such as MP4.

- AVI: is a video file format created by Microsoft. It is one of the oldest video file container specifications. AVI works with a number of different codecs, which can affect how well it is supported by different operating systems and browsers. It prioritizes video quality over compression, meaning that video files are larger and better quality overall.
- FLV: is a video file format created by Adobe Flash. A clear advantage of FLV is its ability to compress video files without severe loss of video quality. However, it is far less compatible across devices and OSes than other file formats: Though it is supported by most browsers and Android devices, it cannot be used to play any video files on iOS devices like iPhones or iPads. Browsers have dropped support for Adobe Flash because it is considered to be insecure, and Adobe no longer supports Flash.
- WebM: is a video file format developed by Google. It is a subset of the open-standard Matroska Video Container (MKV) format, which is highly adaptive to most video and audio codecs and compatible with a wide range of platforms and devices. WebM is a web-friendly, open-source alternative to MP4 that maintains high video quality after compression.

#### 3.13.3. Video players for Raspberry Pi

It is necessary to have a video player in the Raspberry Pi, in order to play the video that are encoded. The more extendable the player is, the more easier and practical it is to use. There are two main video players that can be used on Raspberry [72]:

- VLC: is one of the best video players that one can install on the Raspberry Pi. It will play almost any video and audio format that one throws at it. This means that it's not needed to spend time finding particular codecs to get videos or music playing on Raspberry Pi.
- OMXPlayer: is a video player for the Raspberry Pi that can run completely from the terminal. This video player has been heavily optimized for the Raspberry Pi's hardware and was designed originally as a testbed for the Kodi media player. While OMXPlayer does not have as a wide range of support as VLC, it still has its uses for those who don't want a full-blown video player.

In conclusion, for this project the best video player that can be used is the **OMXPlayer** for its capability of running completely in the terminal, which can be a very useful feature in several cases such as testing through the Admin Remote Client the operation of the MDO-L.

### 3.13.4. Playing video in C++

Although there's already a solution to play videos through terminal, it is also necessary to play videos in the UI and for that case it is mandatory to have a library that can do video playback.

One library that is very useful to do playback is the OpenCV library, having the advantage that this is already used in other subsystems of the project. This library has also the advantage of working compatibly with the chosen ui (explained later on section 3.17).

#### Example of video playback

Here's an example on how to play a video using this library in C++:

Firstly, the definition of the `Player()` class that handles the video playback. Listing 3.7 shows the `Player()` class that needs the `frame` to do the acquisition of the frames of the video, the `frameRate` to know the video frame rate and the `img` that is used to show the frame.

```

1 #ifndef PLAYER_H
2 #define PLAYER_H
3
4 #include <QMutex>
5 #include <QThread>
6 #include <QImage>
7
8 #include <opencv2/core/core.hpp>
9 #include <opencv2/imgproc/imgproc.hpp>
10 #include <opencv2/highgui/highgui.hpp>
11
12 using namespace cv;
13
14 class Player : public QThread
15 {
16     Q_OBJECT
17
18     private :
19         bool stop;
20         QMutex mutex;
21         QWaitCondition condition;
22         Mat frame;
23         int frameRate;
24         VideoCapture capture;
25         Mat RGBframe;
26         QImage img;
27
28     signals :
29         // Signal to output frame to be displayed
30         void processedImage(const QImage &image);
31
32     protected :
33         void run();
34         void msleep(int ms);
35 }
```

```

28 public :
29     // Constructor
30     Player( QObject * parent = 0 );
31     // Destructor
32     ~Player();
33     // Load a video from memory
34     bool loadVideo( string filename );
35     // Play the video
36     void Play();
37     // Stop the video
38     void Stop();
39     // check if the player has been stopped
40     bool isStopped() const;
41 };
42 #endif // VIDEOPLAYER_H

```

Listing 3.7: Example of video playback in OpenCV using C++ - Player class (withdrawn from [73])

Then, listing 3.8 shows the implementation of the main member functions. Firstly, the `loadVideo` function loads the video and reads the framerate of the video. Then, the `run` function reads the frame from the video, attributes it to the `img` to be then showed on the widget and finally waits the specific time to show the next frame in order to respect the video's frame rate, all this execution occurs while the video doesn't stop. Lastly, the `updatePlayerUI` puts the image on the widget using the functions that Qt already gives.

```

1     bool Player::loadVideo( string filename ) {
2         // Load the file and get its framerate
3         capture.open( filename );
4         if ( capture.isOpened() )
5         {
6             frameRate = (int) capture.get( CV_CAP_PROP_FPS );
7             return true;
8         }
9         else
10            return false;
11     }
12
13     void Player::run()
14     {
15         // Delay because of the framerate
16         int delay = (1000 / frameRate);
17         while (!stop) {
18             // Stop play if doesn't read a frame

```

### 3.14. Image filtering

---

```
16         if (!capture.read(frame))
17     {
18         stop = true;
19     }
20
21     // Attribute the image object all the properties from the video frame captured
22     if (frame.channels() == 3)
23     {
24         cv::cvtColor(frame, RGBframe, CV_BGR2RGB);
25         img = QImage((const unsigned char*)(RGBframe.data),
26                       RGBframe.cols, RGBframe.rows, QImage::Format_RGB888);
27     }
28     else
29     {
30         img = QImage((const unsigned char*)(frame.data),
31                       frame.cols, frame.rows, QImage::Format_Indexed8);
32     }
33
34     // show the frame
35     emit processedImage(img);
36
37     // wait the specific time to respect the framerate of the video
38     this->msleep(delay);
39 }
40
41 void MainWindow::updatePlayerUI(QImage img)
42 {
43     if (!img.isNull())
44     {
45         // puts image on the widget (in this case on a label)
46         ui->label->setAlignment(Qt::AlignCenter);
47         ui->label->setPixmap(QPixmap::fromImage(img).scaled(ui->label->size()
48                                         Qt::KeepAspectRatio, Qt::FastTransformation));
49     }
50 }
```

Listing 3.8: Example of video playback in OpenCV using C++ - Player class implementation (withdrawn from [73])

## 3.14. Image filtering

Previously on subsection 3.10.2 it was mentioned how to detect faces using computer vision. After that, it is necessary to use filters and apply them in the user's face in order to have the Snapchat alike filters, for

example like in Fig. 3.31.

Out[15]: <matplotlib.image.AxesImage at 0x1e147482848>

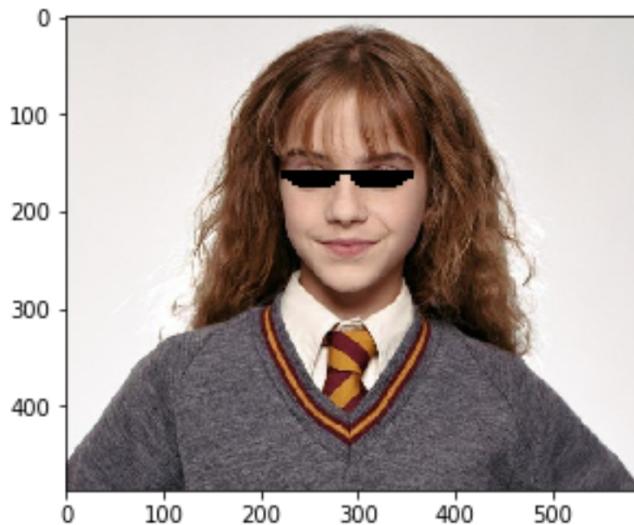


Figure 3.31.: Example of Snapchat like filter(withdrawn from [74])

### 3.14.1. Using filters with OpenCV

Using OpenCV it is also possible to implement that kind of Snapchat like filters in a easy way. In Listing 3.9 is an example on how to implement that type of filters using python. Basically, it are detected all the faces in the image and it is made the scale of the image filters depending on the size of the face, this makes the filter look more real in the user's face. The filter is only an overlayed image, in this case are three images, a mustache, a cigar and glasses. These three images are then scaled and putted on the specific point of the face too look like the user has a mustache, sunglasses and a cigar.

```

1 import numpy as np
2 import cv2

4 face_cascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

6 specs_ori = cv2.imread('glass.png', -1)
cigar_ori = cv2.imread('cigar.png', -1)
8 mus_ori = cv2.imread('mustache.png', -1)

10 # Camera Init
cap = cv2.VideoCapture(0)

```

### 3.14. Image filtering

---

```
12 cap.set(cv2.CAP_PROP_FPS, 30)

13
14
15

16     def transparentOverlay(src, overlay, pos=(0, 0), scale=1):
17         overlay = cv2.resize(overlay, (0, 0), fx=scale, fy=scale)
18         h, w, _ = overlay.shape # Size of foreground
19         rows, cols, _ = src.shape # Size of background Image
20         y, x = pos[0], pos[1] # Position of foreground/overlay image
21
22         for i in range(h):
23             for j in range(w):
24                 if x + i >= rows or y + j >= cols:
25                     continue
26                 alpha = float(overlay[i][j][3] / 255.0) # read the alpha channel
27                 src[x + i][y + j] = alpha * overlay[i][j][:3] + (1 - alpha) * src[x + i][y + j]
28
29         return src
30
31
32 while 1:
33     ret, img = cap.read()
34     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
35     faces = face_cascade.detectMultiScale(img, 1.2, 5, 0, (120, 120), (350, 350))
36
37     for (x, y, w, h) in faces:
38         if h > 0 and w > 0:
39             glass_symin = int(y + 1.5 * h / 5)
40             glass_symax = int(y + 2.5 * h / 5)
41             sh_glass = glass_symax - glass_symin
42
43             cigar_symin = int(y + 4 * h / 6)
44             cigar_symax = int(y + 5.5 * h / 6)
45             sh_cigar = cigar_symax - cigar_symin
46
47             mus_symin = int(y + 3.5 * h / 6)
48             mus_symax = int(y + 5 * h / 6)
49             sh_mus = mus_symax - mus_symin
50
51             face_glass_roi_color = img[glass_symin:glass_symax, x:x + w]
52             face_cigar_roi_color = img[cigar_symin:cigar_symax, x:x + w]
53             face_mus_roi_color = img[mus_symin:mus_symax, x:x + w]
54
55             specs = cv2.resize(specs_ori, (w, sh_glass), interpolation=cv2.INTER_CUBIC)
56             cigar = cv2.resize(cigar_ori, (w, sh_cigar), interpolation=cv2.INTER_CUBIC)
57             mustache = cv2.resize(mus_ori, (w, sh_mus), interpolation=cv2.INTER_CUBIC)
```

```
56     transparentOverlay(face_glass_roi_color, specs)
57     transparentOverlay(face_cigar_roi_color, cigar, (int(w / 2), int(sh_cigar / 2)))
58     #transparentOverlay(face_mus_roi_color, mustache)

59
60     cv2.imshow('Thug Life', img)
61     key = cv2.waitKey(1) & 0xFF
62     if key == ord("q"):
63         break

64
65     k = cv2.waitKey(30) & 0xff
66     if k == 27:
67         cv2.imwrite('img.jpg', img)
68     break

69 cap.release()
70 cv2.destroyAllWindows()
```

Listing 3.9: Example on how to implement filters in faces

Now, it is only necessary to translate this type of approach to C++ language, which is not that hard to do because there is already a knowledge on how to detect faces using this library.

## 3.15. GIF generation

The Graphics Interchange Format (GIF) is a bitmap image format that was developed by a team at the CompuServe in 1987 [75]. It has since come into widespread usage on the World Wide Web due to its wide support and portability between applications and operating systems.

The main features of GIF are [76]:

1. Up to 256 colors using 1 to 8 bits per pixel: allowing a single image to reference its own palette chosen from the 24-bit color space. These palette limitations make GIF less suitable for reproducing color photographs, but well-suited for simpler images such as graphics or logos with solid areas of color.
2. Multiple images per file: supporting animation.

GIF images are compressed using the Lempel–Ziv–Welch (LZW) lossless data compression technique to reduce the file size without degrading the visual quality. It stores multi-byte integers with the Least Significant Bit (LSB) first (little-endian) [76].

Conceptually, a GIF file describes a fixed-sized graphical area (the ‘logical screen’) populated with zero or more ‘images’, which can be animated, resulting in an animated GIF [75].

In the present work, one is interested specifically in the animation capability of the GIF format, as it is widely popular in social media platforms where brands can target their audience.

### **3.15.1. C/C++ libraries and APIs**

In this section, the most important C/C++ libraries and APIs are outlined.

**ImageMagick** is a free and open-source cross-platform software suite for displaying, creating, converting, modifying, and editing raster images, supporting over 200 image file formats, including PNG, JPEG, GIF, WebP, HEIC, SVG, PDF, DPX, EXR and TIFF. ImageMagick can resize, flip, mirror, rotate, distort, shear and transform images, adjust image colors, apply various special effects, or draw text, lines, polygons, ellipses and Bézier curves [77]. It has APIs for C/C++, namely:

1. **MagickWand – C** [78]: it is the recommended interface between the C programming language and the ImageMagick image processing libraries. Unlike the MagickCore C API, MagickWand uses only a few opaque types. Accessors are available to set or get important wand properties.
2. **MagickCore – C** [79]: it is a low-level interface between the C programming language and the ImageMagick image processing libraries and is recommended for wizard-level programmers only. Unlike the MagickWand C API which uses only a few opaque types and accessors, with MagickCore you almost exclusively access the structure members directly.
3. **Magick++ – C++** [80]: is the object-oriented C++ API to the ImageMagick image-processing library. Magick++ supports an object model which is inspired by PerlMagick. Images support implicit reference counting so that copy constructors and assignment incur almost no cost. The cost of actually copying an image (if necessary) is done just before modification and this copy is managed automatically by Magick++. De-referenced copies are automagically deleted. The image objects support value (rather than pointer) semantics so it is trivial to support multiple generations of an image in memory at one time.

Magick++ provides integrated support for the Standard Template Library (STL) so that the powerful containers available (e.g. deque, vector, list, and map) can be used to write programs similar to those possible with PERL & PerlMagick. STL-compatible template versions of ImageMagick’s list-style operations are provided so that operations may be performed on multiple images stored in STL containers.

The **ImageMagick** C API is complex and the data structures are currently not documented. **Magick++** provides access to most of the features available from the C API but in a simple object-oriented and well-

documented framework [81]. Also, Magick++ supports manipulation and conversion of OpenCV data structures — which are used to store frames from the camera and to process them — thus making it the chosen solution.

## 3.16. Social media sharing APIs

Social media platforms are great contact points to target customers and to increase brand awareness, which is highly desirable for digital marketing. These platforms provide a set of functionalities to external agents interact with them in a programmatic way, enabling automation of tasks like content sharing, scheduling, search, etc. These functionalities are exposed by APIs, providing a custom and well-defined interface that can be explored by developers to build custom applications that leverages on them.

There are several social media platforms, but here one focuses on the most popular ones, namely [82, 83]:

1. Facebook: Facebook is a social networking platform that allows users to communicate using messages, photos, comments, videos, news, and other interactive content.
  - API features: Facebook provides various APIs and Software Development Kits (SDKs) that allow developers to access its data and extend the capabilities of their applications. The Facebook Graph API is an Hypertext Transfer Protocol (HTTP)-based API that provides the main way of accessing the platform's data. With the API, one can query data, post images, access pages, create new stories, and carry out other tasks. Furthermore, the Facebook Marketing API allows one to create applications for automatically marketing one's products and services on the platform.
  - Popularity: At the end of 2018, Facebook was boasting of more than 2.2 billion monthly active users, making it the most popular social media platform in the world.
  - Price: The Facebook APIs — Graph API — are provided for free.
  - Ease of use: Apart from its detailed documentation, Facebook has an active developer community with members who are always willing to assist each other make the most of them. The API is large, documentation decent and leans towards PHP, and requires the developer to create a video or screencast to be approved.
  - Registration process: the developer needs to register its app, and if for commercial purposes, register its business. This includes verifying his/her legal entity and address.
2. Instagram: Instagram is a Facebook-owned social networking platform that lets users share photos and videos.

### 3.16. Social media sharing APIs

---

- API features: Facebook offers many APIs to allow developers to create tools that enhance users' experience on the Instagram platform. With the APIs, one can enable users to share their favorite stories and daily highlights from one's application to Instagram. Furthermore, there is the Instagram Graph API that allows developers to access the data of businesses operating Instagram accounts. With the Graph API, one can conveniently manage and publish media objects, discover other businesses, track mentions, analyze valuable metrics, moderate comments, and search hashtags.
  - Popularity: At the end of 2018, Instagram had more than 1 billion monthly active users.
  - Price: The APIs are offered for free.
  - Ease of use: The Instagram APIs are easy to use. Facebook has done good work in providing detailed documentation to assist developers in easily implementing the APIs into their applications.
  - Registration process: subject to the same terms of Facebook registration process. Additionally, the API allows the developer to pull data, but it can not post via the API unless he's/she's a 'Partner'. The partner program started in 2018 and isn't open to new partners unless Instagram asks one to join.
3. Twitter: Twitter is a popular social media service that allows users to find the latest world events and interact with other users using various types of messaging content (called tweets). Twitter can be accessed via its website interface, applications installed on mobile devices, or a short message service (SMS).
- API features: Twitter provides various API endpoints for completing various tasks. For example, one can use the Search API to retrieve historical tweets, the Account Activity API to access account activities, the Direct Message API to send direct messages, Ads API to create advertisement campaigns, and Embed API to insert tweets on one's web application.
  - Popularity: Twitter is a very popular social media networking service that can assist in enhancing the engagement of one's application. At the end of 2018, it had more than 335 million monthly active users.
  - Price: Twitter provides its APIs for free. However, if one want a high level of access and reliability, one'll need to contact them for paid API versions.
  - Ease of use: The Twitter APIs are very easy to use. Twitter provides comprehensive documentation to assist one in flawlessly integrating any of its APIs into one's specific use case. Some API wrappers are also available in several programming languages. The new Twitter

developer site no longer allows localhost in their approved callback URLs, so one'll need to use **ngrok** or an equivalent tunnel to test locally.

- Registration process: it requires the application to be registered as a new project (using Twitter's Developer interface) and the developer can get a permanent access token. Much easier than for the other two platforms.

The social media APIs are specific to each platform. Thus, developing a custom interface for each of these platforms is a very time consuming task. Instead, one will focus only on one social media platform, and its associated API. The chosen platform is **Twitter** as it's the less extent and more easy to implement and its registration process is the least cumbersome.

#### 3.16.1. Twitter API

The Twitter API can be used to programmatically retrieve and analyze Twitter data, as well as build for the conversation on Twitter. Over the years, the Twitter API has grown by adding more levels of access for developers and academic researchers to be able to scale their access to enhance and research the public conversation. Recently, the Twitter API v2 has been released, including a modern foundation, new and advanced features, and quick on-boarding to Essential access [84].

To get access to the Twitter API, the developer needs to [85]:

1. Sign up for a developer account: After signing-up, one will create a Project and an associated developer App during the on-boarding process, which will provide a set of credentials that will be used to authenticate all requests to the API.
2. Save the Application's keys and tokens and keep them secure: After completing step 1, one will be able to find or generate the following credentials within one's developer App:
  - API Key and Secret: Essentially the username and password for one's App. One will use these to authenticate requests that require OAuth 1.0a User Context, or to generate other tokens such as user Access Tokens or an app-only Bearer Token.
  - A set of user Access Tokens: In general, Access Tokens represent the user that one is making the request on behalf of. The ones that one can generate via the developer portal represent the user that owns the App. One will use these to authenticate requests that require OAuth 1.0a User Context. If one would like to make requests on behalf of another user, one will need to use the 3-legged OAuth flow for them to be authorized.
  - Bearer Token: One will use this token when making a request to an endpoint that requires OAuth 2.0 Bearer Token.

### 3.16. Social media sharing APIs

---

3. Make the first request: after completing the first two steps, one can use the API to interact with the Twitter platform.
4. Apply for additional access: With Essential access, one is only able to make requests to the Twitter API v2 endpoints, and not the v1.1 or enterprise endpoints. One is limited to 500K Tweets/month, and unable to take advantage of certain developer portal functionality such as teams and access to additional App environments.

If one wants to access the standard v1.1, premium v1.1, or enterprise endpoints, or if one wants to take advantage of an increased Tweet cap and developer portal functionality, one needs to apply for Elevated or Academic Research access.

It is important to note that the Twitter API is a Representation State Transfer (REST) (a.k.a. RESTful) API, which means that when a client request is made through it, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, is delivered in one of several formats via HTTP: Javascript Object Notation (JSON), HTML, XLT (Excel Templates), Python, PHP, or plain text. The Twitter API uses JSON – the most generally popular file format to use, because, despite its name, it's language-agnostic, as well as readable by both humans and machines [86].

REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways, making REST APIs faster and more lightweight, with increased scalability – perfect for IOT applications and mobile app development [86].

Another important note about RESTful APIs: headers and parameters are also important in the HTTP methods of a RESTful API HTTP request, as they contain important identifier information as to the request's metadata, authorization, uniform resource identifier (URI), caching, cookies, and more. There are request headers and response headers, each with their own HTTP connection information and status codes [86].

For an API to be considered RESTful, it has to conform to the following criteria [86]:

- A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
- Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
- Cacheable data that streamlines client-server interactions.
- A uniform interface between components so that information is transferred in a standard form. This requires that:
  - resources requested are identifiable and separate from the representations sent to the client.
  - resources can be manipulated by the client via the representation they receive because the representation contains enough information to do so.
  - self-descriptive messages returned to the client have enough information to describe how the

client should process it.

- hypertext/hypermedia is available, meaning that after accessing a resource the client should be able to use hyperlinks to find all other currently available actions they can take.
- A layered system that organizes each type of server (those responsible for security, load-balancing, etc.) involved the retrieval of requested information into hierarchies, invisible to the client.
- Code-on-demand (optional): the ability to send executable code from the server to the client when requested, extending client functionality.

To interact with the Twitter platform, one can make a request and retrieve its response by following these steps [87]:

1. Select the end-point: several actions can be performed on the Twitter website or mobile application, varying its interface.
2. Select a tool to make the request: one can use command line tools, driver programs or libraries in several programming languages, or tools like Postman and Insomnia — visual tools to make requests to REST endpoints. For example, here is a cURL example for the user lookup endpoint. To use this request the **BEARER\_TOKEN** and **USERNAME** with Bearer Token and Twitter handle of the developer, and execute it in the command line.

```
curl "https://api.twitter.com/2/users/by/username/$USERNAME" -H  
"Authorization: Bearer $BEARER_TOKEN"
```

3. Review the response: Once a successful request has been made, a payload will be received with metadata related to the request.

- If you used an endpoint that utilizes a GET HTTP method, you will receive metadata related to the resource (Tweet, user, List, Space, etc) that you made the request to in JSON format. Review the different fields that returned and see if you can map the information that you requested to the content on Twitter.
- If you used an endpoint that utilizes a POST, PUT, or DELETE HTTP method, you performed an action on Twitter. Go to Twitter.com or the mobile app and see if you can track down that action.

4. Adjust the request using parameters: Each endpoint has a different set of parameters that can be used to alter the request. For example, additional metadata fields can be requested when using GET endpoints with the fields and expansions parameters.

## Manage Tweets example

Creating and deleting Tweets using the Twitter API is essential for engaging with the public conversation. There are two available methods to manage Tweets [88]:

1. POST: it allows one to post polls, quote Tweets, Tweet with reply settings, Tweet with geo, Tweet with media and tag users, and Tweet to Super Followers, in addition to other features. It can be further customized using parameters. There is a user rate limit of 200 requests per 15 minutes for the POST method.
2. DELETE: allows to delete a specific Tweet. It has a rate limit of 50 requests per 15 minutes.

Since one is making requests on behalf of a user with all manage Tweets endpoints, one must authenticate with OAuth 1.0a User Context and use the Access Tokens associated with a user that has authorized one's App. The Access Tokens can be generated using the 3-legged OAuth flow.

As an example let's focus on the **post tweet** feature. For this purpose, one needs to [89]:

1. Select a tool or library to make the request: Postman, Imsonia, driver programs or libraries in several programming languages. In this case, one will consider cURL and the command line.
2. Authenticate the request: To make a successful request to this endpoint, one will need to use OAuth 1.0a User Context. To do this, the following keys and tokens must be added to the shell environment by exporting the following variables:
  - consumer\_key with your API Key
  - consumer\_secret with your API Key Secret
  - access\_token with your Access Token
  - token\_secret with your Access Token Secret
3. Configure the request with parameters: one can inspect the POST /2/tweets API call to understand its usage and configuration [90]. Some relevant parameters are: **text** – a string containing the text of the Tweet; **media** – a JSON object containing the media information being attached to the tweet. Here, one will use just the first.
4. Make the request: using cURL one has:

```
curl -X POST https://api.twitter.com/2/tweets -H  
"Authorization: OAuth \$OAUTH_SIGNATURE" -H  
"Content-type: application/json" -d '{"text": "Hello World!"}'
```

5. Analyze the response: an example response can be the following JSON object, containing the **id** and the **text** of the newly created tweet.

{

```
"data": {  
    "id": "1445880548472328192",  
    "text": "Hello world!"  
}  
}
```

## C++ libraries and APIs

The Twitter API is a RESTful API, which, in practical terms, means that HTTP headers can be used to send (POST) and retrieve data (GET) from the Twitter platform.

Thus, one only requires an wrapper around these HTTP ‘methods’ to interface Twitter. One such tool, as aforementioned, it’s **cURL** — client Uniform Resource Locator (URL), a command-line utility for transferring data with URLs [91]. It is a free, open-source tool developed in C++, first released in 1997. **cURL** offers ‘**libcurl**’, a library, and ‘**curl**’, a command-line tool, and is often used to retrieve data from Twitter [92].

Several C++ APIs are available for Twitter, although not officially recommended by Twitter:

1. **kQOAuth** by Johan Paul — a Qt based OAuth Library;
2. **libOAuth** by Robin Gareus — a collection of POSIX-C functions implementing OAuth;
3. **QTweetLib** by Toni Jovanoski — a Qt based Twitter API library;
4. **Twitcurl** by Mahesh — a Twitter API library;

There are two Qt based libraries, thus making it highly-dependent on the Qt platform, a collection of POSIX-C functions implementing only the authentication mechanism, and a pure C++ API library for Twitter. Hence, from the above two options arise for building the Local System’s interface for Twitter:

1. **libcurl**: a free and easy-to-use client-side URL transfer library with a C API [93]. One could then write a C++ wrapper around it to follow Object Oriented Programming (OOP) paradigm or use the one of the recommended wrappers — **curlpp**, **curlcpp**, or **C++ Requests** — and then write a wrapper to handle Twitter requests [94];
2. **Twitcurl** [95]: use a pure C++ API to interface Twitter, providing a straightforward mechanism to handle requests. It uses **cURL** for handling HTTP requests and responses and it works well on any OS that supports **cURL**. It supports:
  - v1.1 Twitter REST APIs: timeline, status, user, direct message, friendship, social graph, account, favorite, block, saved search and trend methods;
  - OAuth: authentication methods for Twitter

twitcurl returns JSON responses from [twitter.com](http://twitter.com) as it is. Thus, a C++ JSON parser is required to parse the responses.

Thus, for practicality reasons, `twitcurl` is the preferred option.

## 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 [96]:

- 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’ [97]. 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 [98].

#### 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 [99].

- 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 [99].
- 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 [99].

Compared to callbacks, signals and slots are slightly slower because of the increased flexibility they provide, although the difference for real applications is insignificant [99].

## Usage Example

In Fig.3.32 is an example on how Qt can be used and what it can generate:

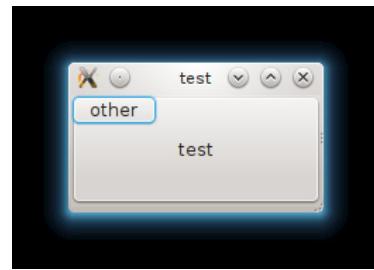


Figure 3.32.: Usage example of Qt(withdrawn from [100])

The code that generates this UI is the following:

```

1 #include <QApplication>
2 #include <QPushButton>

3
4
5 int main(int argc, char **argv)
6 {
7     // Application instantiation
8     QApplication app (argc, argv);

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

12    // Show button
13    button1.show();

14    // Execution in loop of the application,
15    // waiting for its events to trigger the functions
16
17    return app.exec();

```

19 }

Listing 3.10: 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. 3.33.

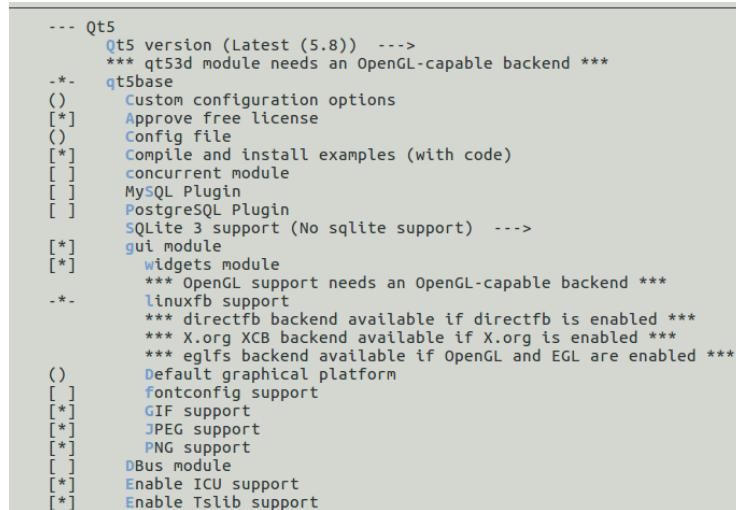


Figure 3.33.: Selection of qt package in buildroot(withdrawn from [101])

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

Sometimes during the operation of all the system, it is necessary to transfer file between machines, for example, when a brand wants to upload video advertisements to its rent, or even when the admin wants to download that ad to verify its reliability.

### **3.18.1. Protocols Overview**

There are various file transfer protocols to use, with different features and types of security and reliability. Throughout this various protocols, these are the most common ones [102]:

- FTP: it is a popular file transfer method that has been around for decades. FTP exchanges data using two separate channels known as the **command channel** to authenticate the user, and the **data channel** to transfer the files. With FTP, both channels are **unencrypted**, leaving any data sent over these channels vulnerable to being taken advantage of. However, it does require an authenticated username and password for access.
- FTPS: it is a secure file transfer protocol that allows you to transfer files securely with trading partners, customers, and users. The transfers can be authenticated through FTPS-supported methods like client certificates, server certificates, and passwords.
- SFTP: it is a secure FTP protocol and a great alternative to unsecure FTP tools or manual scripts. SFTP exchanges data over an SSH connection and provides organizations with a high level of protection for file transfers shared between their systems, trading partners, employees, and the cloud.
- SCP: it is a network protocol that supports file transfers between hosts on a computer network. It's somewhat similar to FTP, however, SCP supports encryption and authentication features.
- HTTP: it is the foundation of data communication. It defines the format of messages through which web browsers and web servers communicate and defines how a web browser should response to a web request. HTTP uses TCP as an underlying transport and is a stateless protocol. This means each command is executed independently and no session information is retained by the receiver.
- HTTPS: it is the secure version of HTTP where communications are encrypted by TLS or SSL.

### **3.18.2. Which protocol is more efficient?**

In this case, **security** is one of the main goals, which means that all data must be sent in the best secure way possible. Thus, all file transfers must have **authentications** and **encryption**. In conclusion, the better choice to this system is to chose the **HTTPS** protocol for several reasons:

- The communications between subsystems are made through TCP/IP, this is also used in this protocol, which is an advantage;
- The data is transferred only with authentications (request and responses) which makes it more secure;
- all communication is encrypted, which makes it even more secure.

### **3.18.3. Example of how to transfer files**

Here are examples on how to configure a connection for file transfers and also how to download a file from a remote server using HTTPS.

#### **Configuring HTTP Connection Characteristics for File Transfers**

The following task is used to customize the connection characteristics for your network to specify a username and password, connection preferences, a remote proxy server, and the source interface to be used. These are the summary steps (that are then explained on Fig. 3.34 and Fig. 3.35) [103]:

1. enable;
2. configure terminal;
3. ip http client connection forceclose | idletimeout <seconds> | timeout <seconds>;
4. ip http client username <username>;
5. ip http client password <password>;
6. ip http client proxy-server <proxy-name> | <ip-address [proxy-port <port-number>];
7. ip http client source-interface <interface-id>;
8. do copy running-config startup-config;
9. end.

#### **Downloading a File from a Remote Server Using HTTP or HTTPS**

Perform this task to download a file from a remote HTTP server using HTTP or HTTPS. The copy command helps you to copy any file from a source to a destination. These are the summary steps (that are then explained on Fig. 3.36 and Fig. 3.37) [103]:

1. enable;
2. Do one of the following:
  - copy [/erase] [/noverify] http://<remote-source-url>local-destination-url>;
  - copy https://<remote-source-url> local-destination-url>.

### **3.18.4. File Transfer Protocol APIs**

There are several APIs that allow to develop applications in order to transfer files between two distinct points using file transfer protocols. The most common is `libcurl`, however, this is used in C language and it is necessary to use a wrapper to follow OOP paradigm, as it was previously said on subsection 3.16.1.

In this case, it will be used one of the wrappers previously mentioned `curlpp`.

### 3.18. File transfer protocols

---

	<b>Command or Action</b>	<b>Purpose</b>
<b>Step 1</b>	<b>enable</b>  <b>Example:</b> Router> enable	Enables privileged EXEC mode. • Enter your password if prompted.
<b>Step 2</b>	<b>configure terminal</b>  <b>Example:</b> Router# configure terminal	Enters global configuration mode.
<b>Step 3</b>	<b>ip http client connection {forceclose   idletimeoutseconds   timeoutseconds}</b>  <b>Example:</b> Router(config)# ip http client connection timeout 15	Configures characteristics for HTTP client connections to a remote HTTP server for all file transfers: • <b>forceclose</b> --Disables the default persistent connection. • <b>idle timeout seconds</b> --Sets the period of time allowed for an idle connection, in a range from 1 to 60 seconds. Default timeout is 30 seconds. • <b>timeout seconds</b> --Sets the maximum time the HTTP client waits for a connection, in a range from 1 to 60 seconds. Default is 10 seconds.
<b>Step 4</b>	<b>ip http client username username</b>  <b>Example:</b> Router(config)# ip http client username user1	Specifies the username to be used for HTTP client connections that require user authentication. <b>Note</b> You can also specify the username on the CLI when you issue the <b>copy</b> command, in which case the username entered overrides the username entered with this command. See the "Downloading a File from a Remote Server Using HTTP or HTTPS: Example" section for an example.

Figure 3.34.: Configuring HTTP Connection - 1(withdrawn from [103])

In Listing 3.11 is an example on how does an HTTP POST is done. Basically it is created an **easy object** that will handle all the POST request, inserting the URL, the header and the POST's field and size, then it is just needed to perform the request.

```

1 #include <cstdlib>
2 #include <cerrno>

3 #include <curlpp/cURLpp.hpp>
4 #include <curlpp/Easy.hpp>
5 #include <curlpp/Options.hpp>
6 #include <curlpp/Exception.hpp>

7 int main(int argc, char *argv[])
8 {
9     if(argc < 2) {
10         std::cerr << "Example 11: Wrong number of arguments" << std::endl
11         << "Example 11: Usage: example12 url"
12         << std::endl;
13     }
14     return EXIT_FAILURE;
15 }
```

### 3.18. File transfer protocols

	<b>Command or Action</b>	<b>Purpose</b>
<b>Step 5</b>	<b>ip http client password <i>password</i></b>  <b>Example:</b> Router(config)# ip http client password letmein	Specifies the password to be used for HTTP client connections that require user authentication.  <b>Note</b> You can also specify the password on the CLI when you issue the <b>copy</b> command, in which case the password entered overrides the password entered with this command. See the "Downloading a File from a Remote Server Using HTTP or HTTPS: Example" section for an example.
<b>Step 6</b>	<b>ip http client proxy-server {proxy-name   ip-address} [proxy-port<i>port-number</i>]</b>  <b>Example:</b> Router(config)# ip http client proxy-server edge2 proxy-port 29	Configures the HTTP client to connect to a remote proxy server for HTTP file system client connections.  • The optional <b>proxy-port<i>port-number</i></b> keyword and argument specify the proxy port number on the remote proxy server.
<b>Step 7</b>	<b>ip http client source-interface <i>interface-id</i></b>  <b>Example:</b> Router(config)# ip http client source-interface Ethernet 0/1	Specifies the interface for the source address in all HTTP client connections.
<b>Step 8</b>	<b>do copy running-config startup-config</b>  <b>Example:</b> Router(config)# do copy running-config startup-config	(Optional) Saves the running configuration as the startup configuration file.  • The <b>do</b> command allows you to execute privileged EXEC mode commands from global configuration mode.
<b>Step 9</b>	<b>end</b>  <b>Example:</b> Router(config)# end  <b>Example:</b> Router#	Ends your configuration session and returns the CLI to user EXEC mode.

Figure 3.35.: Configuring HTTP Connection - 2(withdrawn from [103])

```

char * url = argv[1];

try {
21    curlpp::Cleanup cleaner;
    curlpp::Easy request;

    request.setOpt(new curlpp::options::Url(url));
25    request.setOpt(new curlpp::options::Verbose(true));

    std::list<std::string> header;
    header.push_back("Content-Type: application/octet-stream");

    request.setOpt(new curlpp::options::HttpHeader(header));

    request.setOpt(new curlpp::options::PostFields("abcd"));
33    request.setOpt(new curlpp::options::PostFieldSize(5));

35    request.perform();

```

### 3.18. File transfer protocols

	<b>Command or Action</b>	<b>Purpose</b>
<b>Step 1</b> enable	<b>Example:</b> Router> enable	Enables privileged EXEC mode. <ul style="list-style-type: none"><li>• Enter your password if prompted.</li></ul>
<b>Step 2</b> Do one of the following: <ul style="list-style-type: none"><li>• <b>copy [/erase] [/noverify] http://remote-source-url local-destination-url</b></li><li>• <b>copy https:// remote-source-url local-destination-url</b></li></ul> <b>Example:</b> Router# copy http://user1:mypassword@209.165.202.129:8080/image_files/c7200-i-mx flash:c7200-i-mx	Copies a file from a remote web server to a local file system using HTTP or HTTPS. <ul style="list-style-type: none"><li>• <b>/erase</b> --Erases the local destination file system before copying. This option is provided on Class B file system platforms with limited memory to allow an easy way to clear local flash memory space.</li><li>• <b>/noverify</b> --If the file being copied is an image file, this keyword disables the automatic image verification that occurs after an image is copied.</li><li>• The <i>remote-source-url</i> argument is the location URL (or alias) from which to get the file to be copied, in standard Cisco IOS file system HTTP syntax as follows: <b>http:// [[username:password]@] {hostname   host-ip} [/filepath]/filename</b></li></ul>	

Figure 3.36.: Downloading a File using HTTP - 1(withdrawn from [103])

	<b>Command or Action</b>	<b>Purpose</b>
		<b>Note</b> The optional <i>username</i> and <i>password</i> arguments can be used to log in to an HTTP server that requires user authentication, in place of configuring the <b>iphttpclientusername</b> and <b>iphttpclientpassword</b> global configuration commands to specify these authentication strings. <ul style="list-style-type: none"><li>• The <i>local-destination-url</i> is the location URL (or alias) to put the copied file, in standard Cisco IOS file system syntax as follows: <b>filesystem : [/filepath][/filename]</b></li></ul> <b>Note</b> For more information on URL syntax when you use the <b>copy</b> command, see the "Additional References" section.

Figure 3.37.: Downloading a File using HTTP - 2(withdrawn from [103])

```

}
37 catch ( curlpp::LogicError & e ) {
    std::cout << e.what() << std::endl;
39 }
catch ( curlpp::RuntimeError & e ) {

```

```
41     std :: cout << e.what() << std :: endl;
42 }
43
44 return EXIT_SUCCESS;
45 }
```

Listing 3.11: POST example using <curlpp.h>

### Third-party applications

To make the file transfer more easier, it will be used a third-party application named `transfer.sh`. This application has easy-to-use commands in order to transfer files with a URL, there is unlimited upload, it can encrypt files and it is also possible to limit the amount of downloads and days available of downloading [104].

With a simple command like `curl -upload-file <path-to-the-file> https://transfer.sh/<name-that-you-want>` the file is uploaded to the site and it generates a URL to transfer that file using the command `curl <generated-url> -o <name-you-want>`. And with this simple steps it is possible to easily transfer files between two nodes.

Using the library `curlpp` with the `transfer.sh` API, it is possible to create a program to transfer files between nodes automatically when requested from the user, without the user needs to decorate or do such commands.

# 4. Design

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

## 4.1. Hardware specification

The first step for system design is the HW specification. This can be pictured as a block diagram, this block diagram was already shown and mentioned in section [2.3.1](#) on Fig. [2.2](#).

### 4.1.1. Architecture

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

As it can be seen in Fig. [4.1](#), the Raspberry Pi is the main controller and it is powered by the Power Supply through USB-C. The Power Supply also supplies via Micro-USB the LCD Display control board and the Fragrance Diffusion Actuator. The LCD Display board connects to its board through 50-pin Transistor-Transistor Logic (TTL) and the board connects to the Raspberry Pi through High-Definition Multimedia Interface (HDMI). The Speakers also connect to the LCD Display board through JST PH2.0. Beyond being powered up, the Fragrance Diffusion Actuator also connects to the Raspberry Pi through the General Purpose Input/Output (GPIO). What also connects to the Raspberry Pi through GPIO are the Motion Detection sensors. Lastly, the Camera connects through Camera Serial Interface (CSI) to the Raspberry Pi.

### 4.1.2. Main Controller

The main controller was also previously mentioned because it makes part of one of the requirements of this project: use the Raspberry Pi 4B (Fig. [4.2](#)). This SoC has several of specifications [[105](#)]:

#### 4.1. Hardware specification

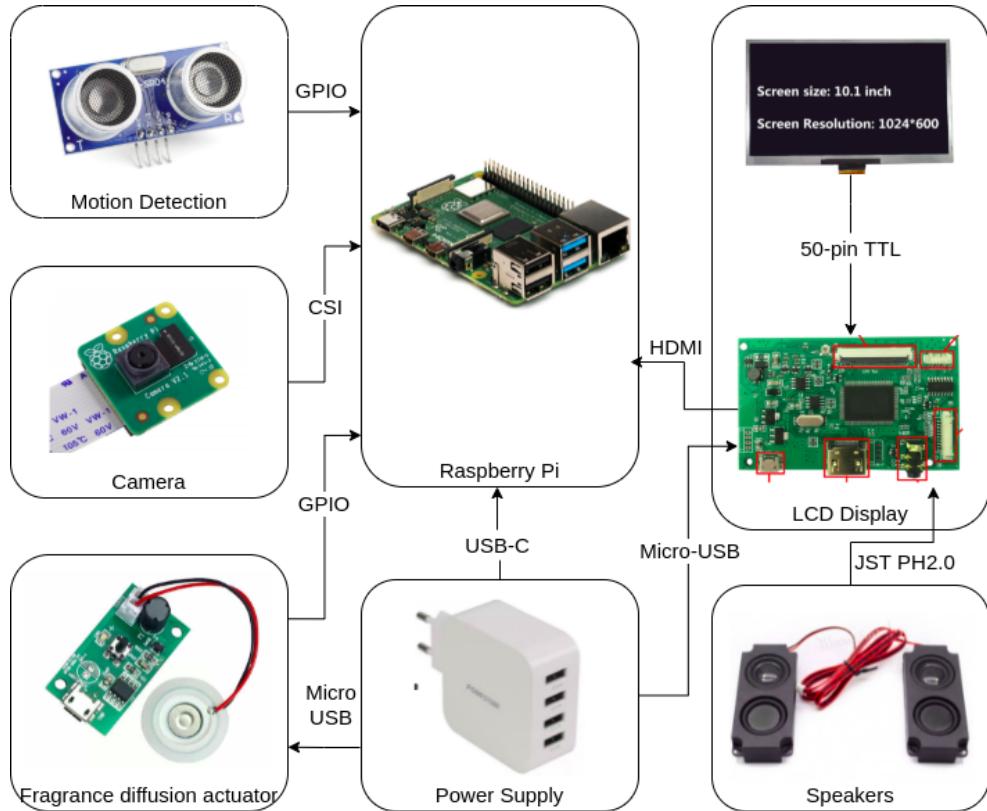


Figure 4.1.: HW architecture Block Diagram

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

#### 4.1. Hardware specification

---

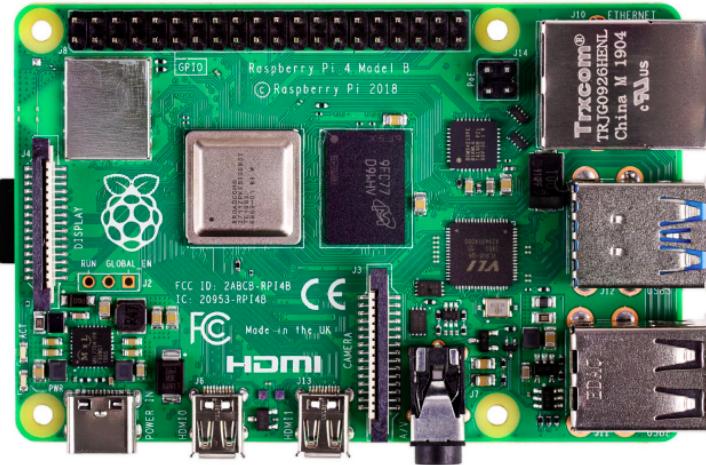


Figure 4.2.: Raspberry Pi model 4B

#### SD Card

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

#### 4.1.3. Motion Detection

For the motion detection, it was already mentioned in section 3.12 that the best option is to use an ultrasonic sensor. Thus, the sensor that has been chosen is the HC-SR04 Ultrasonic Sensor. This sensor has the following specifications [106]:

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

#### Sensor Pinout

In Fig. 4.3 is described the sensor pinout and each pin works as follows [106]:

#### 4.1. Hardware specification

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

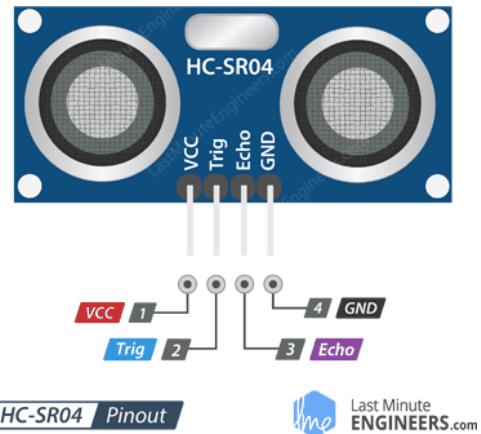


Figure 4.3.: HC-SR04 Pinout (withdrawn from [106])

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

#### 4.1.4. Fragrance Diffusion Actuator

The chosen fragrance diffusion actuator is in Fig. 4.4 and has the following specifications [107]:

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



Figure 4.4.: Fragrance module (withdrawn from [107])

#### 4.1.5. Camera

The camera to use in this project needs to be compatible with the board in use, in this case, the Raspberry Pi. Thus the camera module that is used is the **Raspberry Pi Camera Module V2** (Fig. 4.5).

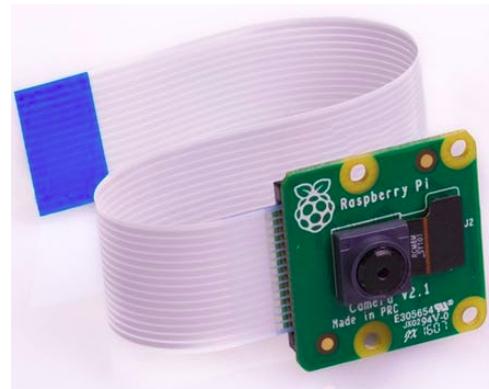


Figure 4.5.: Camera module (withdrawn from [108])

This camera module has a **Sony IMX219 8-megapixel sensor** and can be used to take high-definition video, as well as stills photographs. It supports 1080p30, 720p60 and VGA90 video modes, as well as still capture. It attaches via a 15 centimeters ribbon cable to the CSI port on the Raspberry Pi. The camera works with all models of Raspberry Pi 1, 2, 3 and 4. It can be accessed through the MMAL and V4L APIs, and there are numerous third-party libraries built for it, including the Picamera Python library. [108].

### 4.1.6. LCD Display

In Fig. 4.6 is the display that is used in this project. One advantage on this display is that it has audio drivers, which means that it is only necessary to plug a speaker and the board can handle the rest. It is also important to refer that the display isn't touch because there is no need to it and also this was the chosen one because it was the bigger and best on market considering quality and price.

As it can be seen, the display has 10.1 inches and it is supplied with 5V DC and with a current of 2 A via a micro USB port. Fig. 4.7 show all the interfaces that the board module of the display provides. That was also one more reason for the choice of this display: it has an **HDMI interface** to connect to the Raspberry, the **50Pin TTL Screen Interface** that will connect to the display and two options to plug audio - the **Speaker Interface** and the **3.5mm audio interface**.



Figure 4.6.: Display (withdrawn from [109])

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

### 4.1.7. Speakers

When playing video ads, it is not only necessary a display, but also a speaker to playback the sound of the ads. As it can be seen in Fig. 4.7, the screen board has two different interfaces of audio: the speaker interface and the audio interface. For this project it will be used speakers that uses the speaker interface, because that type of speakers with that interface are passive speakers and don't need a DC power supply, which is an advantage [110].

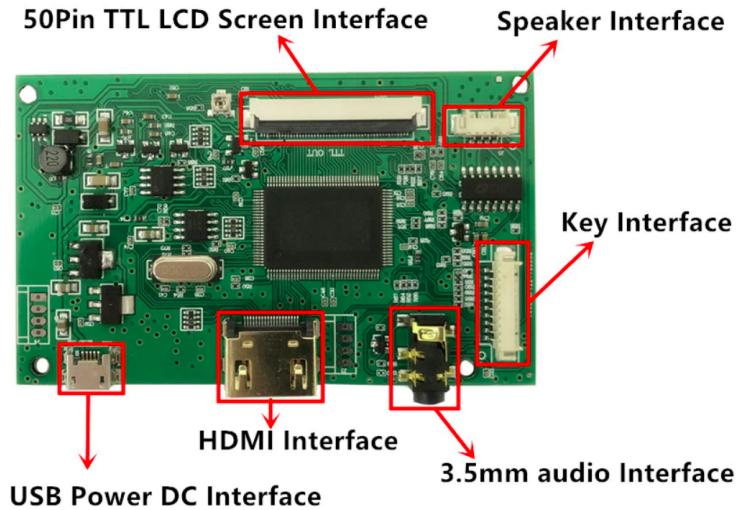


Figure 4.7.: Display Interfaces (withdrawn from [109])

Thus, the speakers that will be used have an impedance of 8 Ohm and a power of 5 Watts and are displayed on Fig 4.8.



Figure 4.8.: Speakers (withdrawn from [111])

#### 4.1.8. Power Supply

The MDO-L will be a plugged in system, so it will be needed a plugged in power supply to supply all the components of the system. In total, the power consumption will not overtake 20 Watts and all the supplies necessary are plugged by USB (Raspberry Pi, Fragrance Diffusion Actuator, and screen). So, it will be used the power supply in Fig. 4.9, that has 4 outputs of 5 V DC and 2.4 A DC each.

#### 4.1. Hardware specification

---



Figure 4.9.: Power Supply (withdrawn from [112])

#### 4.1.9. On/Off button

In this context, an On/Off button can be useful to power On/Off the MDO-L. However, this feature is not that necessary, so, this prototype will be only powered through the power supply previously talked in section 4.1.8 and will be always on until the power supply be unplugged, powering off the machine.

#### 4.1.10. Total HW cost

The total HW cost can now be precisely calculated, once that all the hardware is now specified on table 4.1, yielding about 175 EUR.

Table 4.1.: Total spending on Hardware

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

## 4.2. Hardware interfaces definition

After specifying the HW, it is important to define its interfaces. Firstly, it is necessary to have basic know-how of the main board's pinout. In Fig. 4.10 is represented the pinout of the Raspberry Pi.

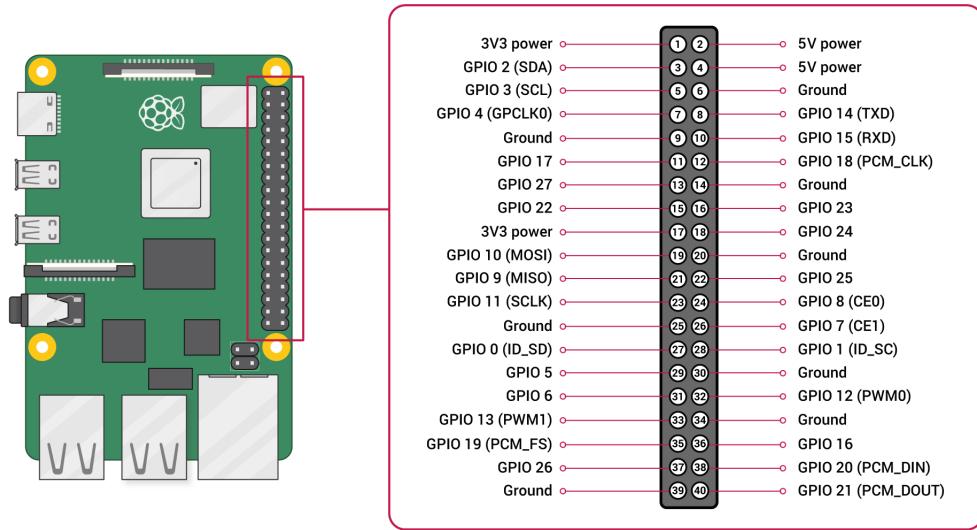


Figure 4.10.: Power Supply (withdrawn from [113])

### 4.2.1. Peripherals Mapping

After have a basic know-how of the Raspberry's pinout, it is now possible to map all the HW components. As it can be seen in Fig. 4.11, only two types of peripherals need to be mapped in the Raspberry: the ultrasonic sensor and the fragrance diffusion actuator.

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

Lastly, the fragrance diffusion actuator. Although this one is powered up by Micro-USB, it is necessary to activate or deactivate the diffusion and this is only possible with a **MOSFET**, activating its gate. The gate

## 4.2. Hardware interfaces definition

is connected to the pin 36 (GPIO 16) and this is the pin responsible to activate or deactivate the fragrance diffusion.

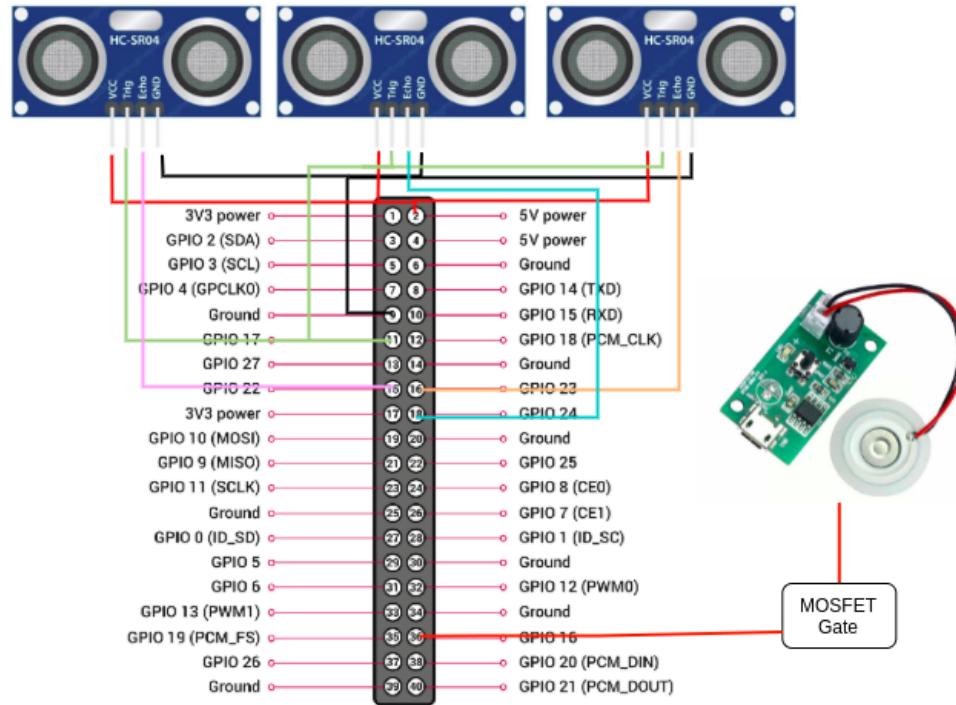


Figure 4.11.: Peripheral Mapping

To turn things more clear, Table 4.2 shows all the pin mapping in a more clear way.

### 4.2.2. Test Cases

In order to verify if all the hardware is in good conditions, are made some test cases to each component. In Table 4.3 are displayed all the test cases to be done in all pieces of HW. Basically, all tests are functional tests and need to be done to know if some component may be damaged or also could not be in the error range provided by the manufacturer.

### 4.2.3. PCB Design

As it can be seen in Fig: 4.11, it is a little bit complex and messy to look and conceive a circuit with all those connections, and since many connections are in common, it is an advantage to design a PCB to make things more simpler.

## 4.2. Hardware interfaces definition

Table 4.2.: Pin Mapping

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

For the design of this PCB it was used the software **PADS**, developed by Mentor Graphics. In Fig. 4.12 is the schematic of the connections to the PCB, this part was made on **PADS Logic** and basically is joining everything that is common to the sensors to then connect to the raspberry Pi, then there's a pin for every Trigger of every sensor and finally a pin for the gate of the MOSFET and two pins for its supply.

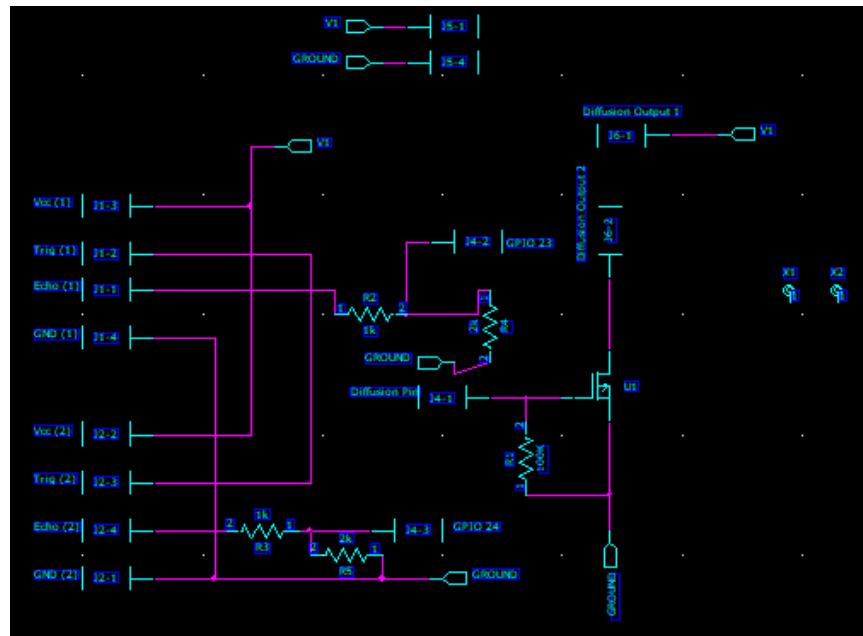


Figure 4.12.: PCB schematic

When the logic is done, it is moved to the layout on **PADS Layout** where all the components are placed

#### 4.2. Hardware interfaces definition

---

Table 4.3.: Hardware Test Cases

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

#### 4.3. Mechanical structure

---

and connected through tracks. In Fig 4.13 is the final layout of the PCB. Note that there are two tracks that are more wide, this is because they are the tracks that supply the MOSFET and this one need more current, which means wider tracks.

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

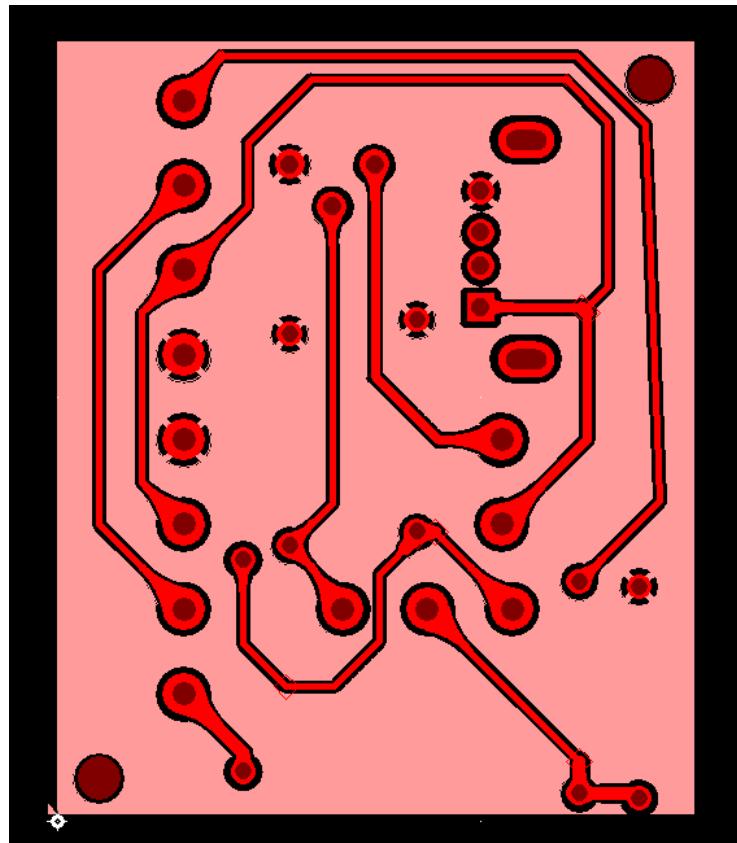


Figure 4.13.: PCB layout

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

## 4.3. Mechanical structure

In this section the mechanical structure design is discussed. The design considerations included low weight, easy assembly and rapid manufacturing, providing a means to fast iteration, low cost, and flexible. For the aforementioned reasons, the manufacturing process chosen was laser cutting of MDF boards.

#### 4.3. Mechanical structure

---

Fig. 4.14 illustrates the 3D model of the enclosure. It contains seven panels, six for the exterior and one middle layer, where the majority of the HW components are meant to be assembled. The front panel contains an opening for Liquid Crystal Display (LCD) display mounting, camera and ultrasonic sensors to detect the user.

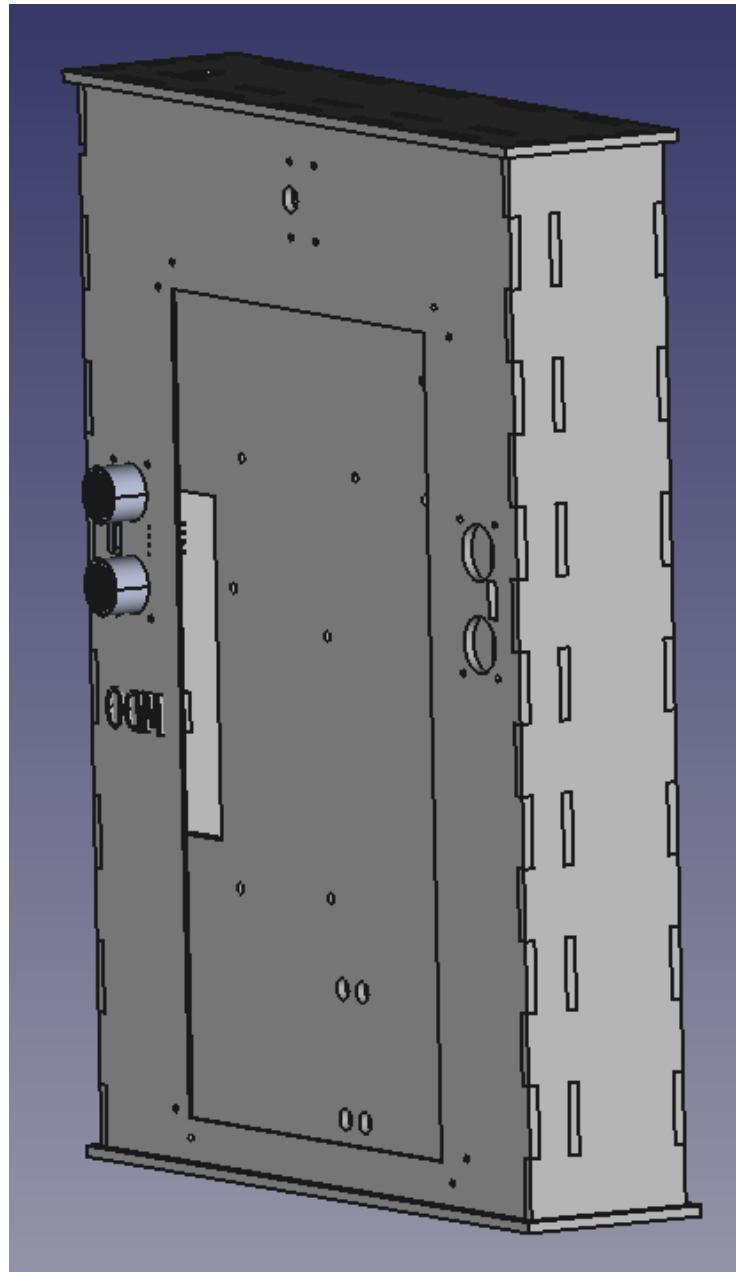


Figure 4.14.: Mechanical design: Enclosure perspective

Fig 4.15 presents the fixation panel for the majority of the HW, namely the LCD controller, Raspberry Pi,

#### 4.3. Mechanical structure

---

custom PCB, fragrance diffuser, and speakers.

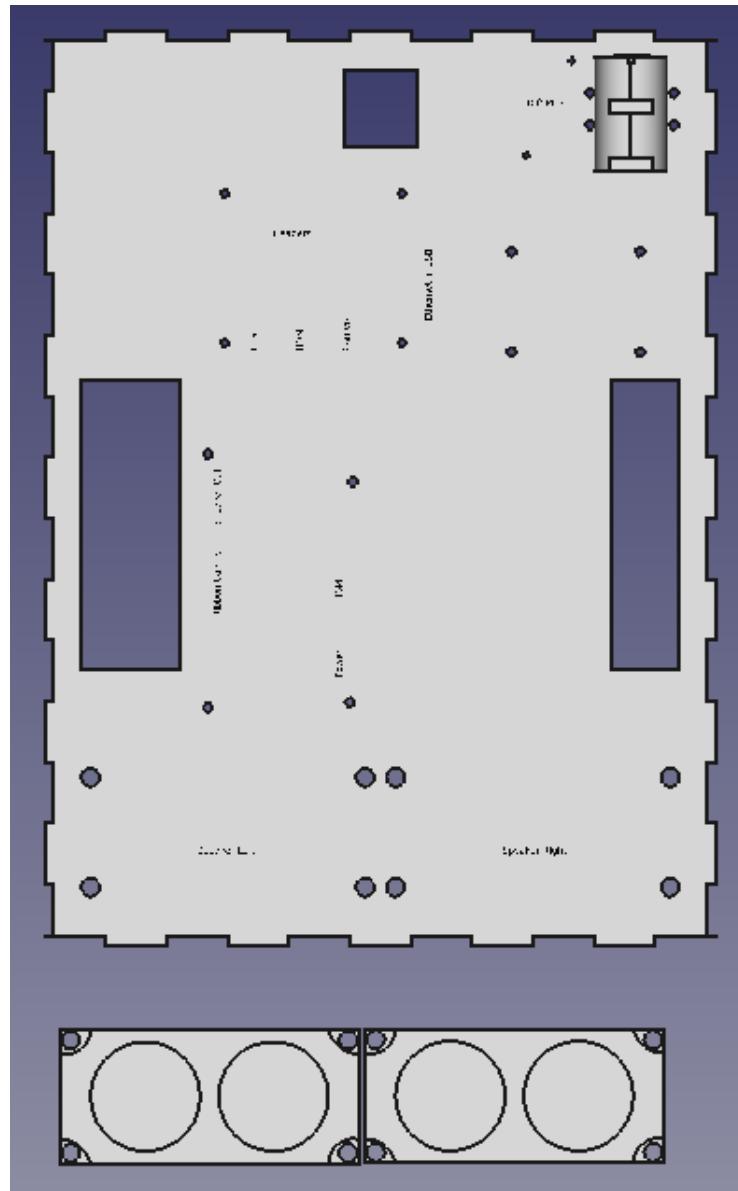


Figure 4.15.: Mechanical design: Hardware fixation panel

Fig 4.16 presents the top panel of the enclosure, with a hole for the fragrance diffusion. It showcases the flexible mechanical assembly method, with grooves and protrusions to provide mechanical coupling.

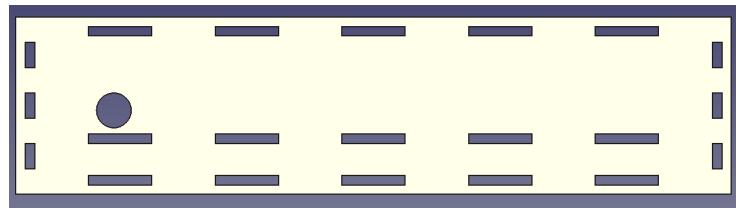


Figure 4.16.: Mechanical design: Top panel

## 4.4. Software specification

Next, the SW responsible for system operation is specified for all subsystems — MDO Remote Client (MDO-RC), MDO Remote Server (MDO-RS), and MDO Local System (MDO-L). All these subsystems are event-driven (asynchronous), and they can be more easily specified using state-machine diagrams, previously illustrated in the [analysis phase](#) (Section 2.3.2). Also in the [analysis phase](#), the use case diagrams helped to identify the main features required for the system and the respective sequence diagrams helped to clarify the intervening objects and the interaction among them.

In this section, the analysis phase information is used to derive the SW design specification, i.e., the set of SW artifacts that translate the gathered information into an implementable form.

Thus, firstly, the SW architecture is devised using components diagrams for all subsystems, alongside with the deployment specification, mapping the SW components to the HW nodes. The database is designed using the ER model and the ERD. Then, the data formats for communication between subsystems are devised, determining a well-established protocol for communication.

Next, the static architecture of the system — classes diagram — is devised and the threads are specified, alongside with its priorities, as all subsystems operate multiple tasks concurrently. The flowcharts are then devised for all subsystems, oriented towards events and threads.

The test cases for each subsystem are listed, defining its operation and the expected result. The COTS SW and the third-party libraries are identified and a mapping between class topics and the foreseeable implementation is presented for clarification. Finally, the SW tools are listed.

### 4.4.1. Software architecture

The system's SW architecture was devised using [UML component diagrams](#) for [Remote Client](#) (Fig. 4.17), [Remote Server](#) (Fig. 4.18), and [Local System](#) (Fig. 4.19). Each component diagram illustrates all SW components for the system in analysis and the interaction between them, and its interfaces with external subsystems.

## Remote Client

Fig. 4.17 depicts the Remote Client SW architecture, encapsulated in the package MDO-RC: AppManager, and is comprised of the following artifacts:

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

## Remote Server

Fig. 4.18 depicts the Remote Server SW architecture, encapsulated in the package MDO-RS: AppManager. It interacts with the Remote Client (package MDO-RC: AppManager), with the Local System (package MDO-L: AppManager), and with the DB server (MDO-RS: DB Server). The database management is done using client-server architecture, with MDO-RS: AppManager containing the DB client and MDO-RS: DB Server the server.

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

- Command-Line Interface package: contains the CLI, the CLI Engine, and the CLI Parser. It provides the server external interface for clients to perform requests, capturing the events which drive the Remote Server's logic.
- Comm Manager package: manages incoming and outgoing connections to the Remote Client (package MDO-RC: App Manager), and each Local System (package MDO-L: AppManager) it needs to interact. It periodically checks all connections statuses. All connections consist of TCP/IP sockets.
- DB Client package: manages the queries and the associated responses by building or parsing them,

#### **4.4. Software specification**

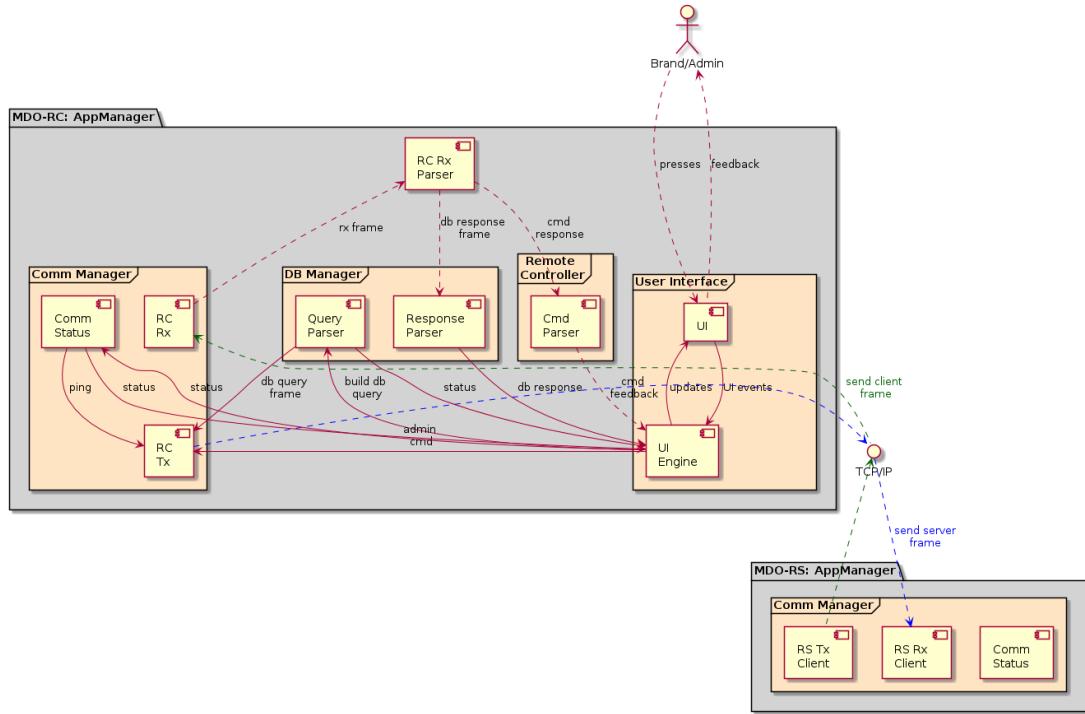


Figure 4.17.: SW architecture: component diagram – Remote Client

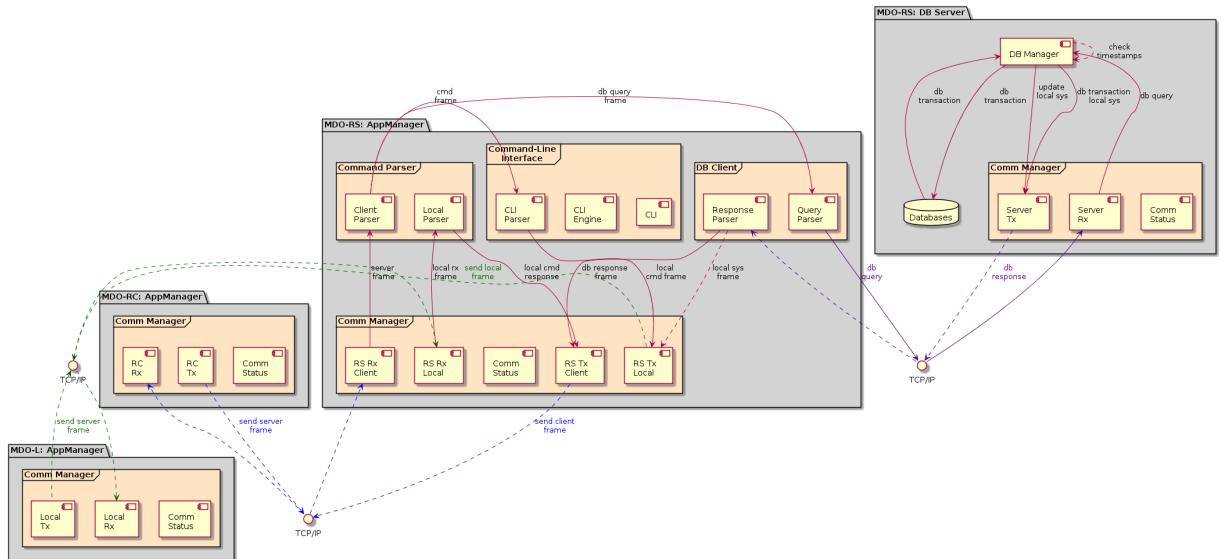


Figure 4.18.: SW architecture: component diagram – Remote Server

respectively. It performs the requests for DB server with the solicited queries.

- Command Parser package: contains the Client Parser, and the Local Parser to parse and handle frames received from the Remote Client or Local System, respectively, forwarding it for appropriate

dispatching.

- TCP/IP sockets: incoming/outgoing connection nodes, through which incoming or outgoing traffic flows for the Remote Client, the Local System and the DB Server.

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

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

## Local System

Fig. 4.19 depicts the Local System SW architecture, encapsulated in the package MDO-L: AppManager.

It interacts with:

- Remote Server (package MDO-RS: AppManager): to retrieve updates on its operation or Admin commands
- Twitter (via its REST APIs): to share posts on it
- transfer.sh – an URL proxy server: to ease file transfer between the Remote Server and the Local System.

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

- User Interface package: contains the UI, the UI Engine, the Gesture Recognition Engine, and the User Detection package. The UI provides user feedback and UI Engine captures the events that drive the Local System's logic. When a user approaches the Local System, the Ultrasonic sensor device driver captures this event and passes it to the user space where the Ultrasonic sensor daemon logs it, which, in turn, signals this event to the UI Engine. When the User performs gestures, the Gesture Recognition Engine requires a service to the Gesture recognition component (Computer Vision Framework) to recognize the gesture. Also, when the User is detected, the UI Engine requests the Normal mode manager to stop running, and requests Face detection to track people's faces in the camera.
- Comm Manager package: manages incoming and outgoing connections to the Remote Server (package MDO-RS: App Manager), and the internet for each web service it needs to interact (Twitter and

#### **4.4. Software specification**

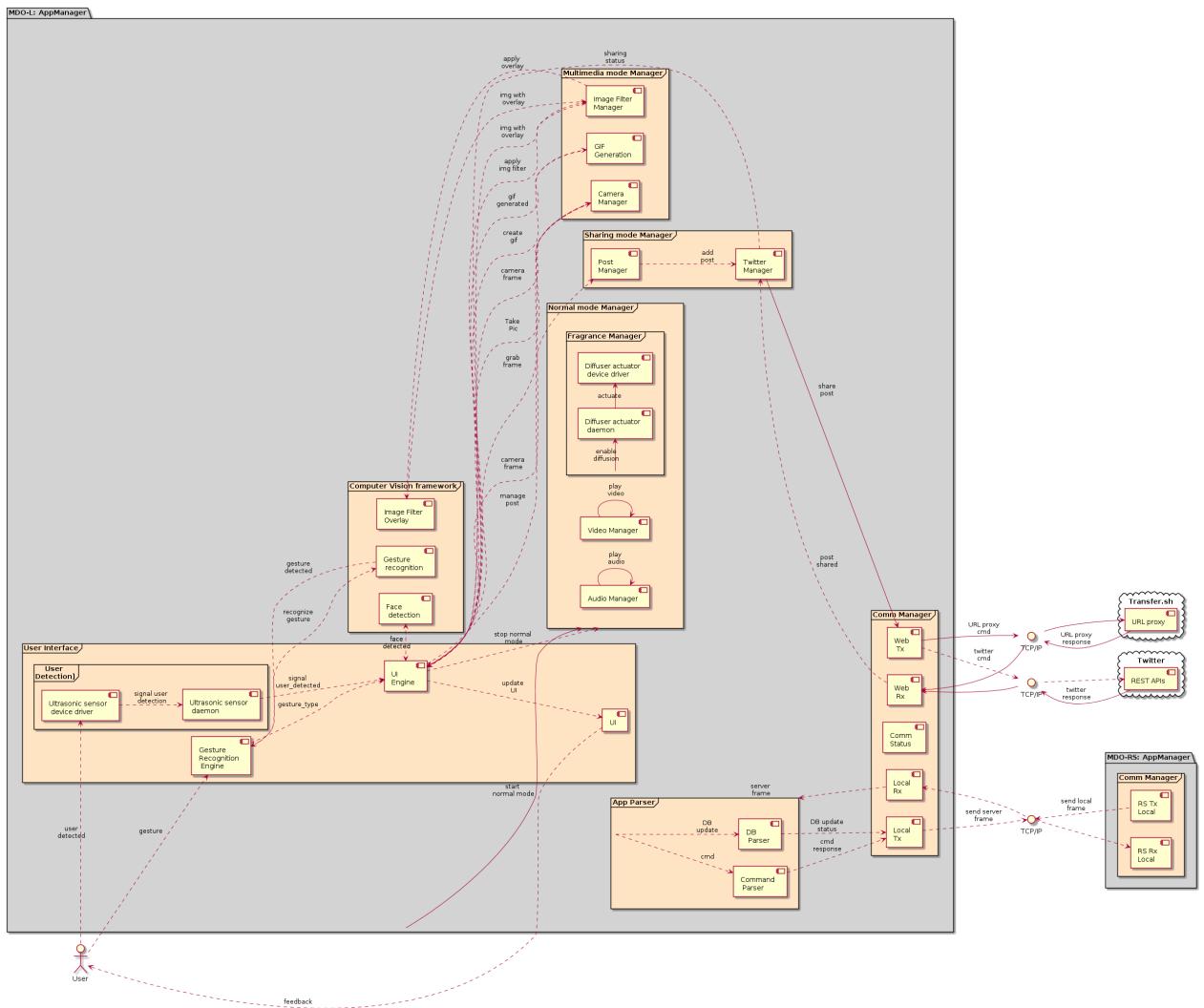


Figure 4.19.: SW architecture: component diagram – Local system

`transfer.sh`). It periodically checks all connections statuses. All connections consist of TCP/IP sockets.

- Computer Vision framework package: manages the computer vision related tasks, namely, gesture recognition, face detection, and image filter overlay.
  - Multimedia Mode Manager package: manages the multimedia mode related tasks, namely, image filtering, GIF generation, and camera management.
  - Sharing Mode Manager package: manages the sharing mode related tasks, namely, post management, and social media management, in this case, **Twitter**.
  - Normal Mode Manager package: manages the normal mode related tasks, namely, fragrance dif-

fusion, video and audio outputs. The fragrance diffusion is requested to the device driver using a daemon to bridge user-space and kernel-space.

- App Parser package: manages the parsing for database queries and requested commands.
- TCP/IP sockets: incoming/outgoing connection nodes, through which incoming or outgoing traffic flows for the **Remote Server**, and the web services for **Twitter** and the **URL proxy server**.

### 4.4.2. Deployment specification

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

Fig. 4.20 depicts the deployment diagram for the MDO system. In blue are represented the computational nodes of interest, namely:

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

#### 4.4. Software specification

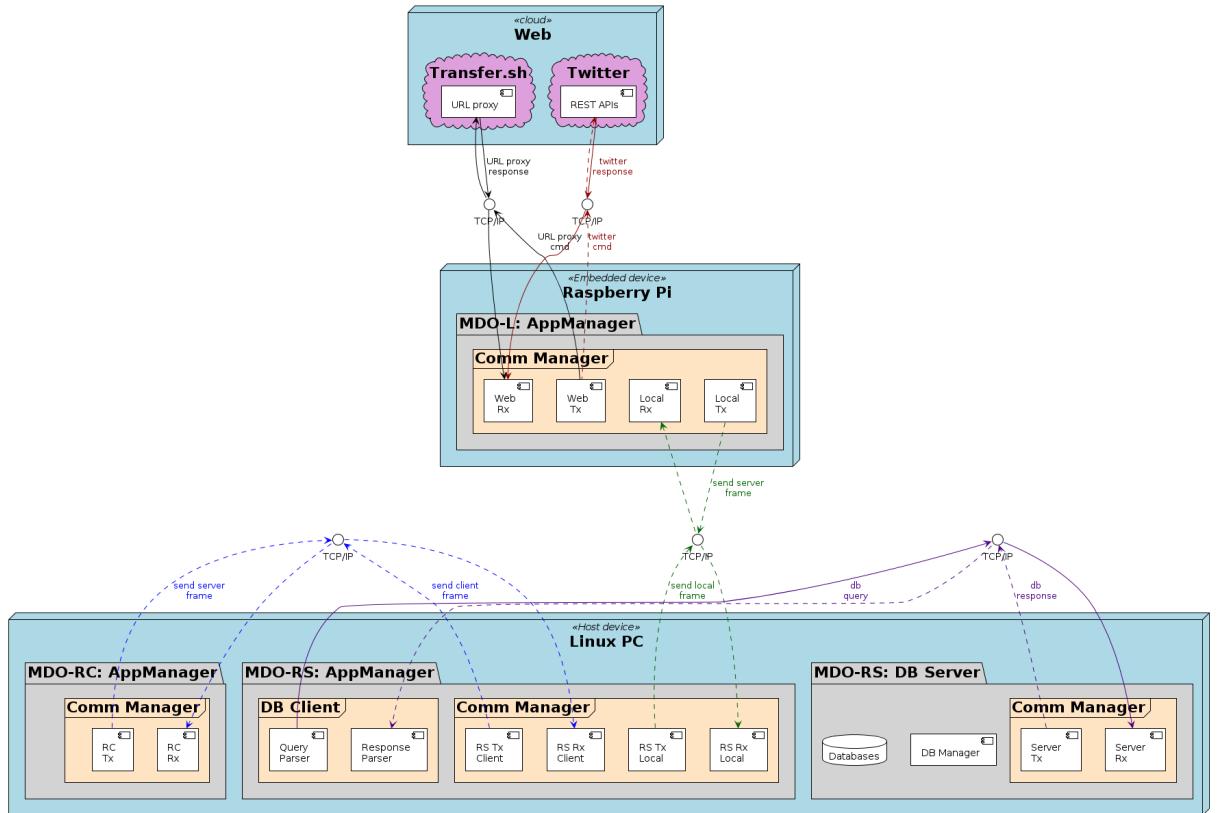


Figure 4.20.: Deployment diagram

#### 4.4.3. Database design

As aforementioned in Section 3.11.6, the conceptual design of a database consists of a high-level description of the data to be stored in the database and its constraints, which is usually carried out using the ER model. Thus, in Section 3.11.7 the key concepts of the ER model are presented, which can be reviewed to assist the comprehension of the subsequent database design produced.

Fig. 4.21 depicts the ERD for the conceptual design of the MDO system's database, with the primary keys shown in red and the foreign keys in blue, comprising the relevant entities and its relationships, namely:

- **User:** models an User in the application (Brand or Admin). It has a role, a name, an email, and a pass (which is encrypted before being stored). An User manages 0 or many UserStations and owns 0 or many Ads.
- **UserStations:** it unfolds the relationship many-to-many between an User and a Station. An User manages 0 or many UserStations, and a UserStations contains one or many Stations. It imports the User.id as a foreign key to enable referencing an User.
- **Station:** it models a physical MDO station. It contains an id, a name, a location and an IP address.

It imports the `UserStations.id` as a foreign key to enable referencing an `UserStations`.

- **TimeTable**: it models a weekly timetable that every `Station` contains. It contains one or many `TimeSlots`. It imports the `Station.id` as a foreign key to enable referencing a `Station`.
- **TimeSlot**: it models a slot of time available for ads reproduction. It contains the duration (in minutes), the cost, and a flag signaling if it is rented or not. It imports the `TimeTable.id` as a foreign key to enable referencing a `TimeTable`.
- **Ad**: it models an advertisement. It imports the `User.id` as a foreign key to reference an `User`, a `Fragrance.id` to reference the associated `Fragrance`, and a `TimeSlot.id` to reference a `TimeSlot`. The `Fragrance` is optional, but, as the foreign key must not contain a null value, a default value of `0` is used to signal that no fragrance is used.
- **MediaFile**: it models a media file that must be attached to the `Ad`. It contains an id, file specifications (filename, filesize, and filetype), mdata for storing the file and an optional description. It imports the `Ad.id` as a foreign key to enable referencing an `Ad`.
- **Fragrance**: it models a `Fragrance`. It is added to the database by the `Admin` and it can be selected by the `Brand` from the `FragranceList` available for each `Station`. It contains an id, a name, an intensity to define the actuation time, a maximum capacity (`vol_ml_max`) and a current capacity (`vol_ml_level`), and an optional description. It imports the `FragranceList.id` as a foreign key to reference the `FragranceList` available for each `Station`.
- **FragranceList**: it represents the list of fragrances available for each `Station`. It imports the `Station.id` as a foreign key to reference its associated `Station`.

#### 4.4.4. Data formats

After devising the SW architecture the data flow across the several subsystems became clearer, with data frames being conveyed through TCP/IP sockets. Thus, it is important to define the data formats for internal representation and communication between the subsystems, as illustrated in Fig. 4.22. In gray are depicted the several subsystems, in white the data frames and in blue the legend.

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

#### 4.4. Software specification

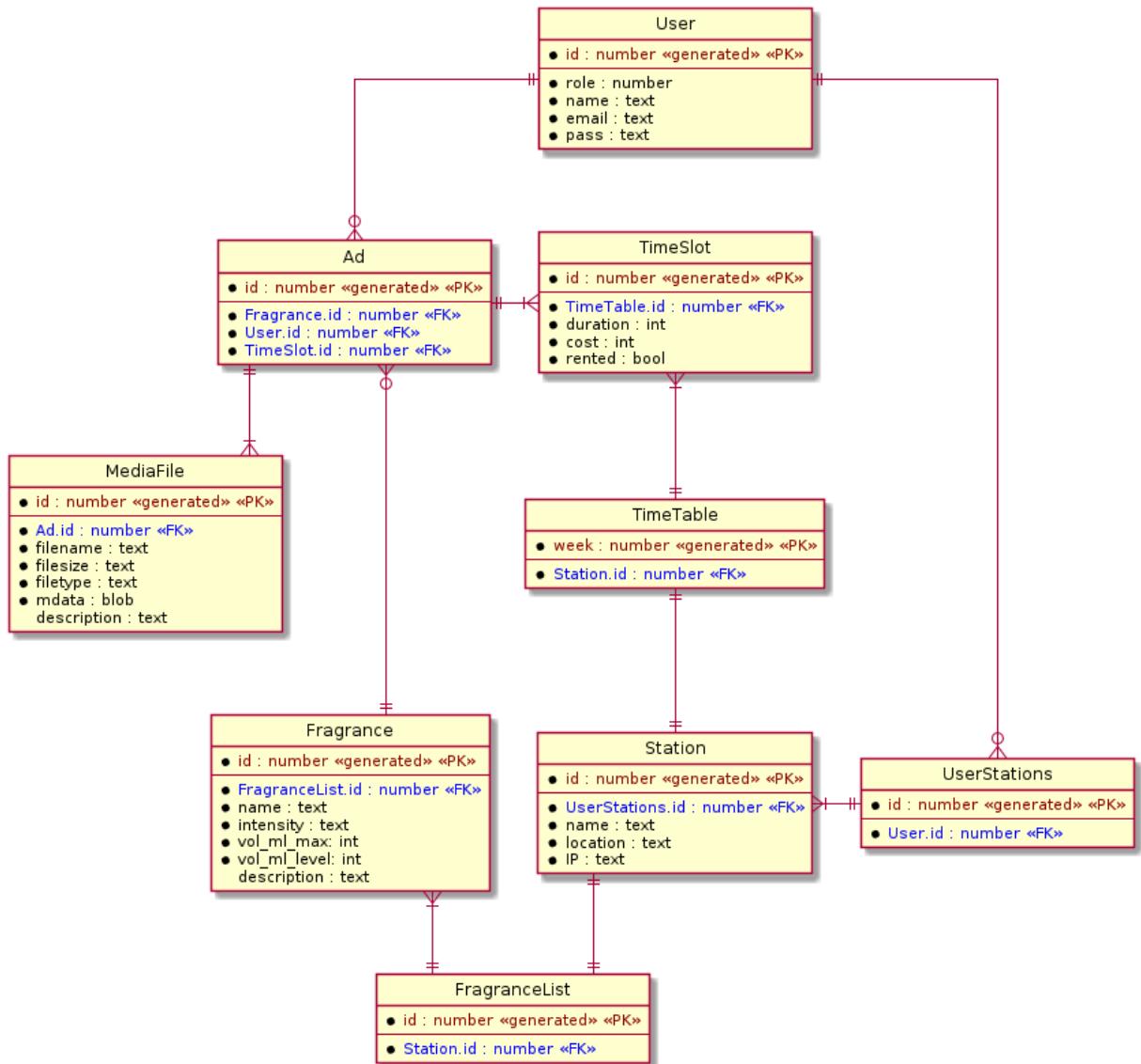


Figure 4.21.: MDO Entity-Relationship Diagram: conceptual design

system – of type **int** to prevent collisions with normal messages sent.

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

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

#### 4.4. Software specification

System.

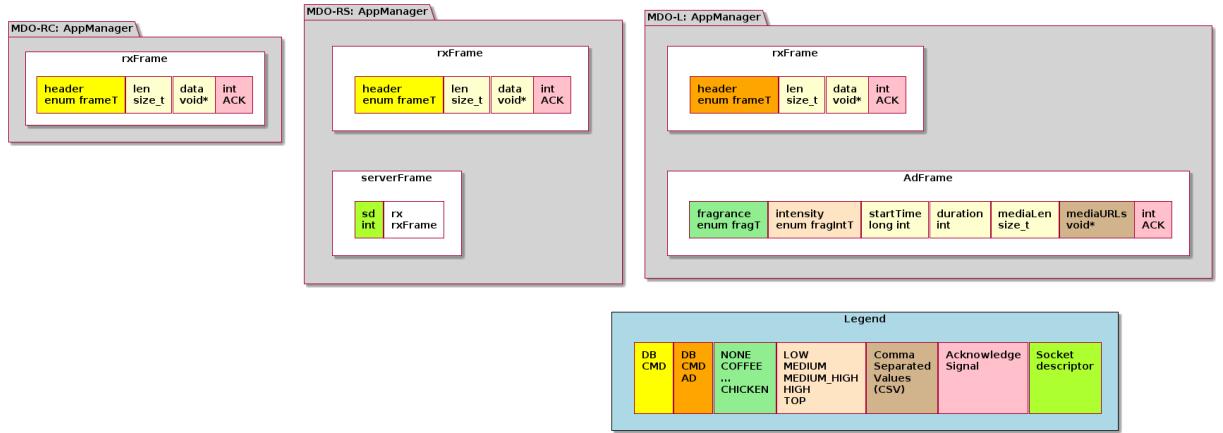


Figure 4.22.: Data formats

#### 4.4.5. Static architecture – Class diagrams

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

##### Remote Client

Fig. 4.23 depicts the class diagram for the Remote Client subsystem. The **UIApp** and **UIWindow** handle all the events made by the user with the UI. The **CommManager** handles the communication between this system and the **Remote Server**, which has an enumerator to classify the connection status (**ConnStatus**) and also the class **DBManager** handles all the databases through its functions and enumerators **DB\_OPER** and **DB\_OBJ**. Then, there's the normal classes that are mandatory to have: **User** and **Admin** that have a relation (because an Admin is a User), **Station** with its **TimeTable**, **Date** and also **Ad** to handle the ads.

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

#### **4.4. Software specification**

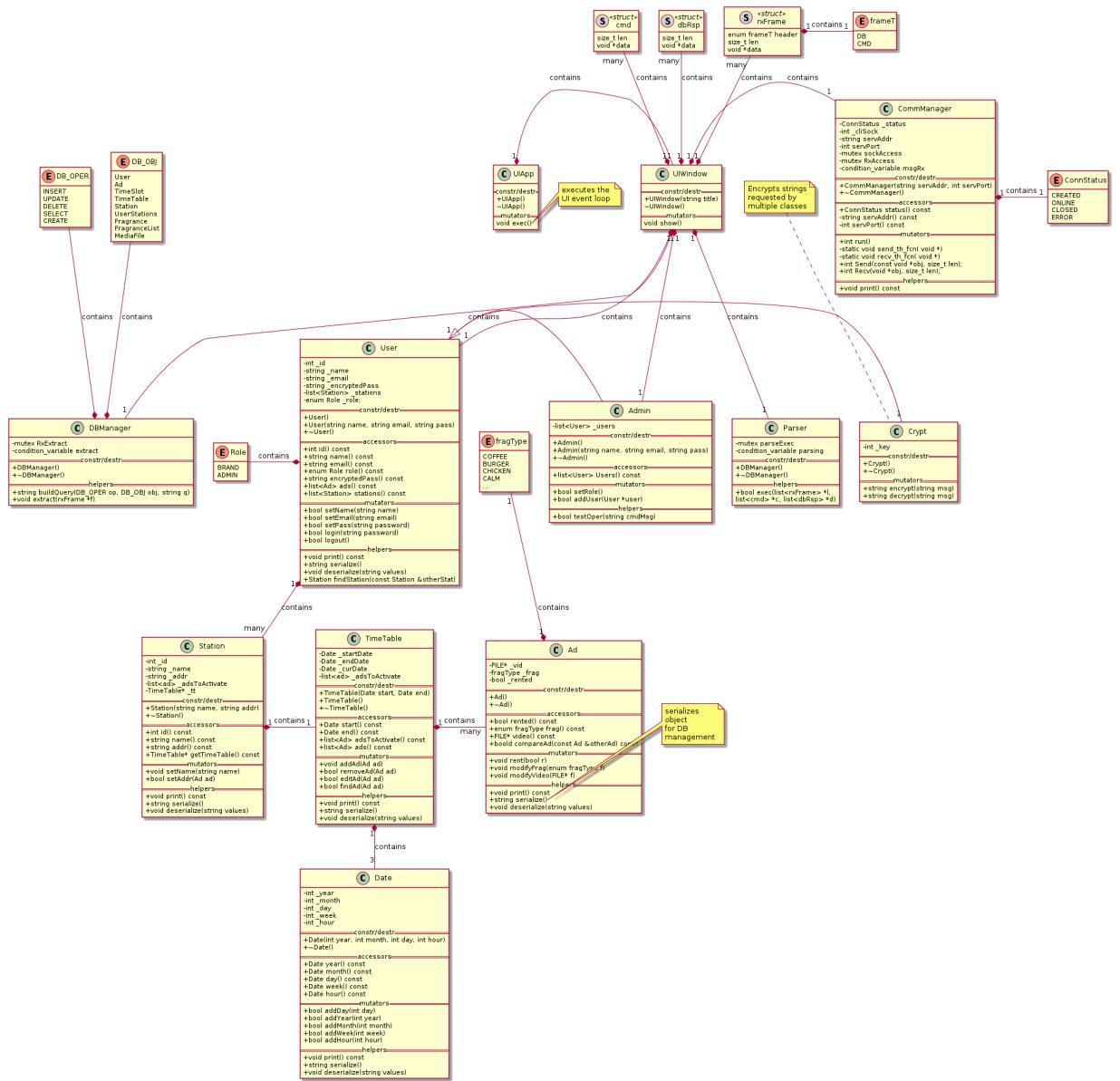


Figure 4.23.: Class diagram: Remote Client

# Remote Server

Fig. 4.24 depicts the class diagram for the Remote Server subsystem. The CLI is the Command Line Interface and handles almost everything in this subsystem. It has several members, the more important ones are the send and receive vectors that have the type of data serverFrame containing the rxFrame and the socket to which the server is connected, this facilitates the work when sending and receiving because there is already the knowledge of the socket to or from the message is being received or sent. Allied to

#### **4.4. Software specification**

this, there's all the conditional variables and mutexes that maintain the stability of all variables and avoid collisions between classes, communications and so on.

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

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

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

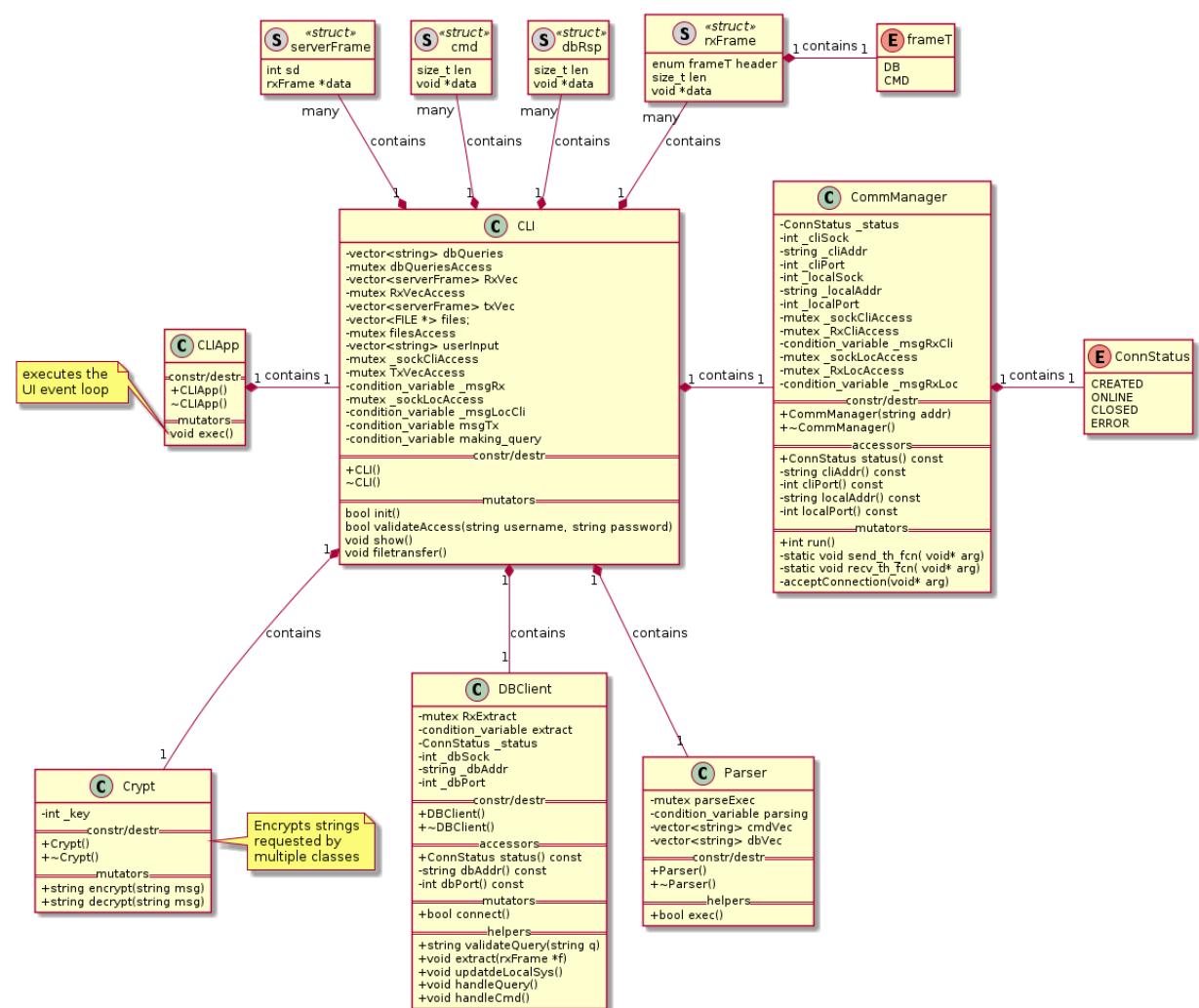


Figure 4.24.: Class diagram: Remote Server

## Local System

Fig. 4.25 and Fig. 4.26 depicts the class diagram for the Local System.

In Fig. 4.25 is illustrated a partial view of the class diagram of the Local System, containing the UIApp which executes the UI event loop. As the majority of the events are triggered by the User it is reasonable to compose the UI object of all the other ones, and the relevant data structures.

The CommManager handles all communications to the RemoteServer, the UserDetectionEngine handles the user detection and the GestureRecognitionEngine handles the recognition of user gestures. A Parser abstract class is derived to obtain specialized parsers for DB, commands and Twitter. Twitter is derived from SocialMedia abstract class to handle Post objects and share them on Twitter.

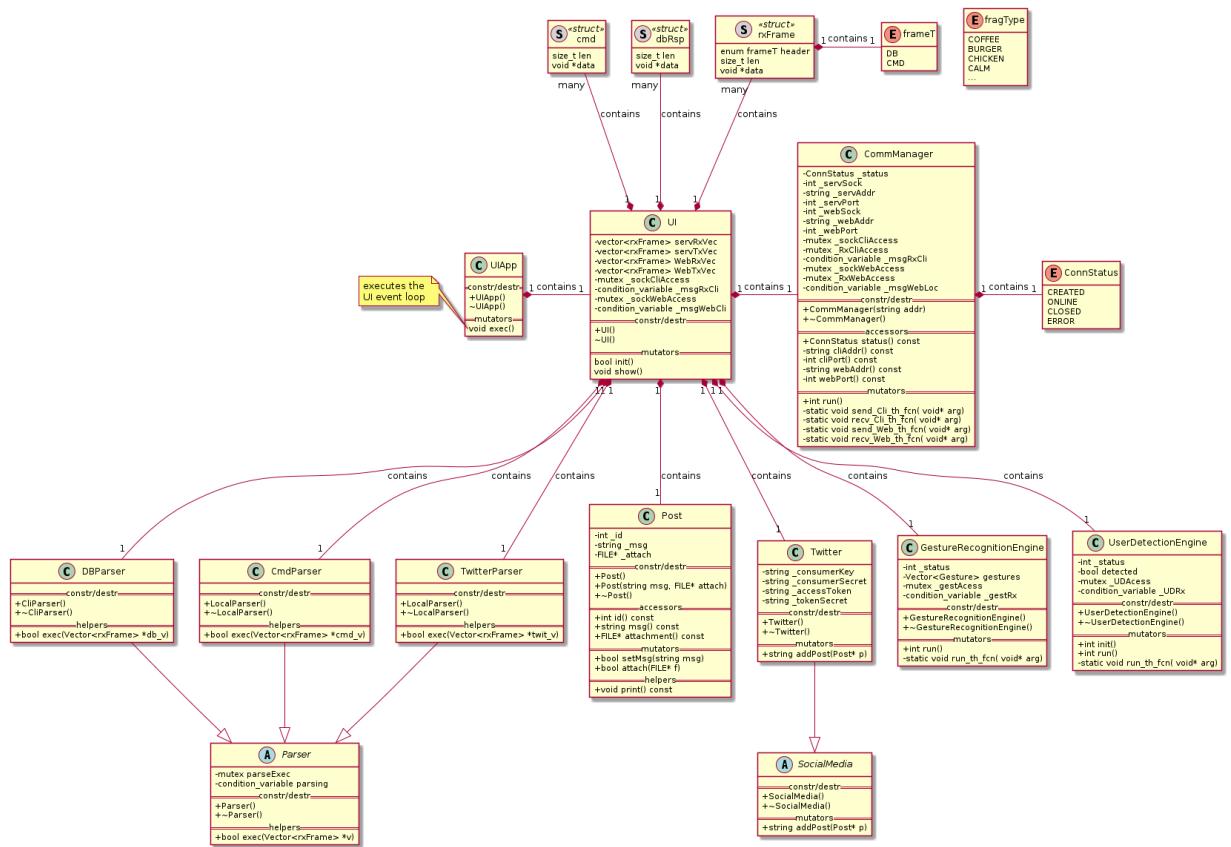


Figure 4.25.: Class diagram: Local System – part 1/2

In Fig. 4.26 is illustrated the second part of the class diagram of the Local System.

As it can be seen, there are some classes that handle some features of the station, such as GifGenerator, VideoManager, AudioManager, ImageFilterEngine, CameraEngine and FragranceManager. All these classes handles all modes of the local system and, obviously they all interact with the UI.

#### **4.4. Software specification**

There is also the class **Ad** that handles the ads and has the structure **adFrame** containing some information that are relevant to download the associated media files and play the ad on the station.

There are several threads in various classes that will be explained next in section 4.4.6.

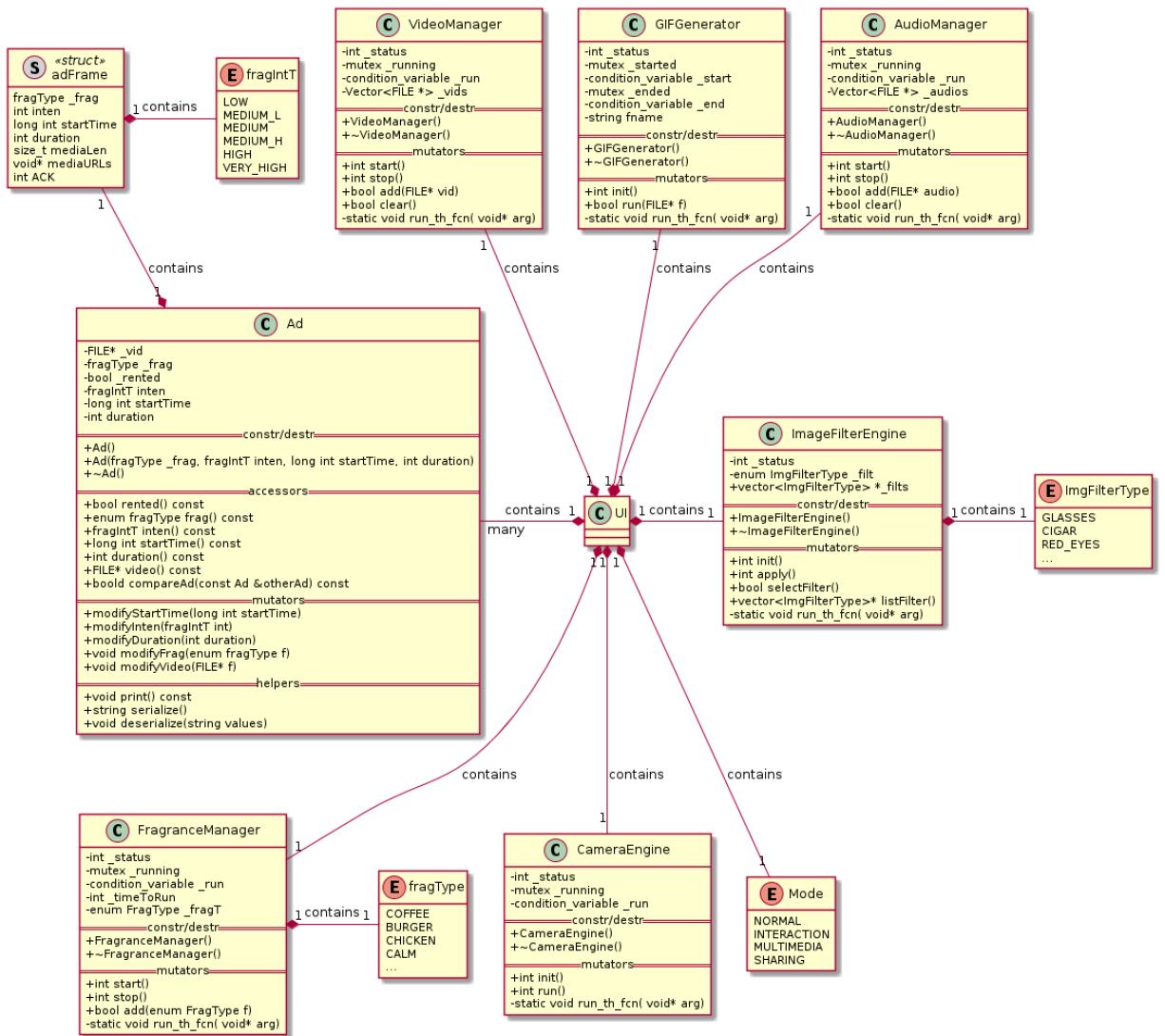


Figure 4.26.: Class diagram: Local System – part 2/2

#### **4.4.6. Threads specification**

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

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

## Remote Server

Fig. 4.27 illustrates the thread specification and the associated priorities for the Remote Server. There are four logical groups:

1. CLI: the command line interface has a thread to transfer files because it can be necessary to transfer them to both subsystems. This thread has low priority because it is not a critical feature that need to happen as soon as possible and, depending on the file, it can take a while until its transfer, so, it can have a lower priority and take its time to transfer.
2. Parser: the parser has one thread that is its execution. This thread has high priority because it is mandatory to parse and execute the commands that are received, as soon as they are received.
3. Comm: the communication manager has three threads that are important: the receive, the send, and the accept connection. The one with most priority is the receive thread, because it is necessary to receive the messages to then parse and execute them. Then, the other two threads have medium priority because receiving and parsing messages is more important than this.
4. DB: the database management has three threads that have high priority, that's because it is important to always have the databases updated. These three threads make queries to the databases, but they are controlled one by another in order to avoid the risk to make queries at the same time or some queries to be lost. There's another thread that handles the commands sent to the server or between the two subsystems

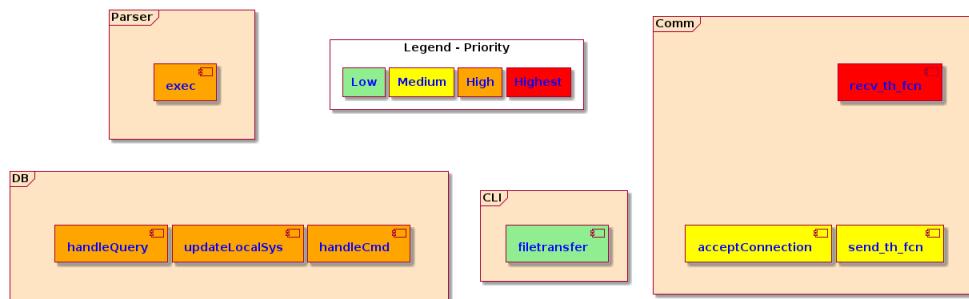


Figure 4.27.: Thread specification: Remote Server

## **Local System**

Fig. 4.28 illustrates the thread specification and the associated priorities for the Local System. There are five logical groups:

1. **User Interface**: the **UserDetection** detects when a user is in range and is the top priority thread, as it will determine the device's mode, stopping normal mode and going into interaction mode.  
The **FrameGrabber** and **UI** threads are high priority. The first grabs frames from the camera and stores it in a frame buffer which is available to the **UI**, **ImgFilterOverlay**, and **GestureRecognition** threads. The second is responsible for providing user feedback on the LCD display.  
The **ImgFilterOverlay**, and the **GestureRecognition**, and the threads are medium priority. The first is responsible for detecting faces on the camera frame buffer and applying the selected image filter overlay, and the second for recognizing **User** gestures.  
It should be noted that only the **UserDetection** and **UI** threads should be permanently running, while the others are only running while an **User** is detected.
2. **Comm**: these threads are responsible for communications handling. **LocalRx** is the top priority thread, as it needs to check if an relevant command or database update has arrived. It stores the incoming data frames into a First-In, First-Out (FIFO) buffer.  
The **LocalTx** is a high priority threads, responsible for transmitting outgoing data frames to the **RemoteServer**.  
The **TwitterShare** and **FileTransfer** are low priority tasks, as they don't impact the user experience or device operation significantly. The first is responsible for sharing posts on Twitter. The second handles file transferring between **RemoteServer** and the **LocalSystem**.
3. **App**: these threads handle app specific tasks. The **AppParser** is a high priority thread, consuming incoming data frames and dispatches it for additional processing of commands – **CmdHandler** – or of **Ads** requiring file transfer – **FileTransfer**.  
The **CmdHandler** is a medium priority task responsible for responding to commands sent to the Local System.
4. **Normal mode**: these threads are only running when normal mode is on. **AudioMan** and **VidMan** are high priority tasks, and they are responsible for Ad's audio and video management, respectively.  
**FragMan** is a medium priority task, as variations in the fragrance diffusion are less significant for user experience. It is responsible for diffusing the fragrance.
5. **Multimedia mode**: the **GifGenerator** is a low priority thread, responsible for generating GIFs. It should only run after a user request was acknowledged.

#### 4.4. Software specification

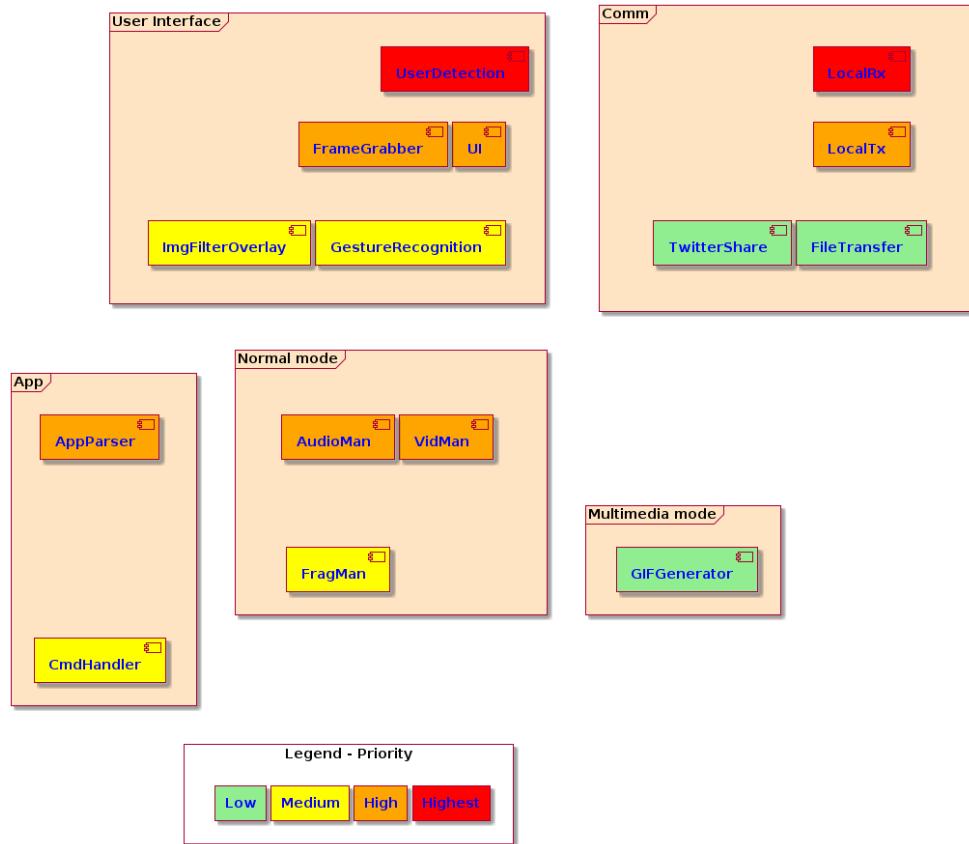


Figure 4.28.: Thread specification: Local System

#### 4.4.7. Flowcharts

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

#### Remote Client

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

In Fig. 4.29 is the flowchart of the Login feature. After the login button being pressed, it is necessary to verify if all fields are filled. If so, the password is encrypted in order to compare with the passwords from the database (that are all encrypted). It is made a query to the users database to select a count and the role of all users that have the same username and password. If the count is equal to one, then the user

#### 4.4. Software specification

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

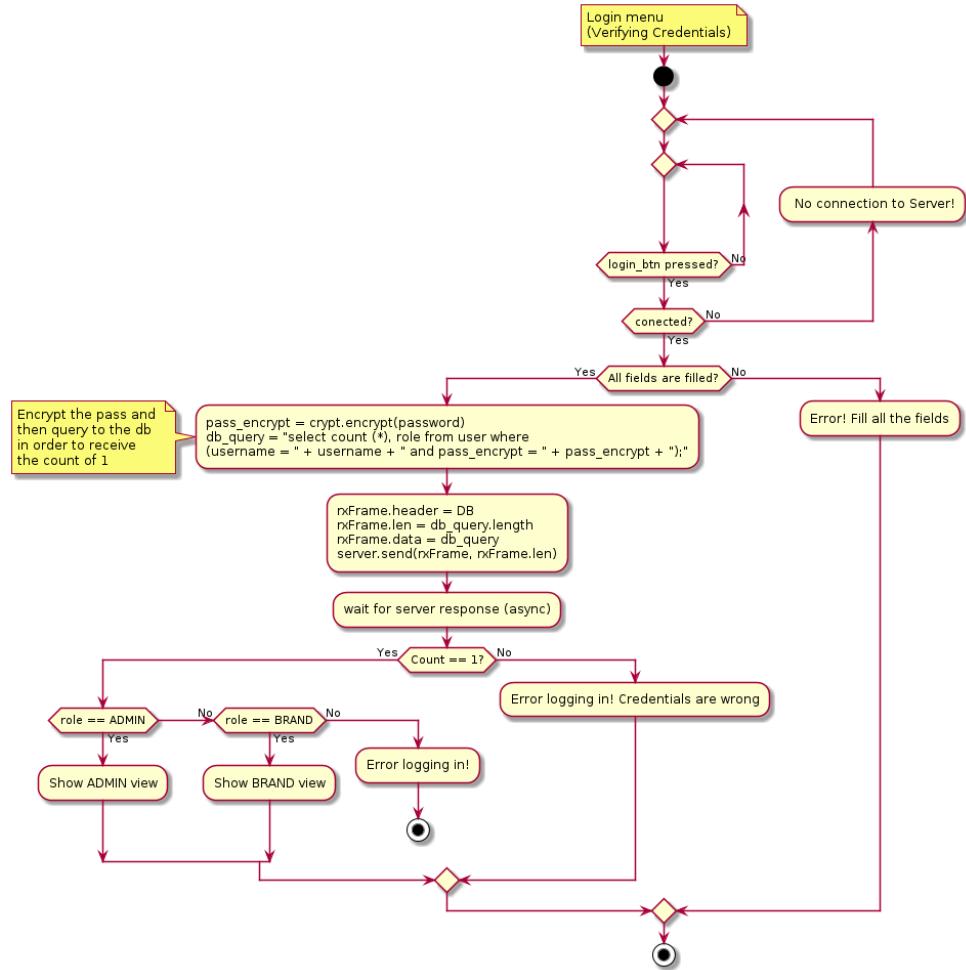


Figure 4.29.: Flowchart: Remote Client – Login

In Fig. 4.30 is depicted the flowchart for registering a user. Basically, after the button being pressed and all the fields are filled, an user object is created. To know if the user can be created, it is made a query to the Users database to know if there is already an user with that username or that password. If not, then it is made another query to add that user to the users database and the registration is complete.

In Fig. 4.31 is depicted a flowchart of the delete user feature. Basically, the user is deleted using its id, and for that it is made a query to the users database to delete the row with that user id. After that query, the delete process is completed

In Fig. 4.32 is the flowchart of how it behaves the rent ad feature. Basically an ad object is created and added to the timetable object. After that, it is made a query to the time table database to add the ad to

#### 4.4. Software specification

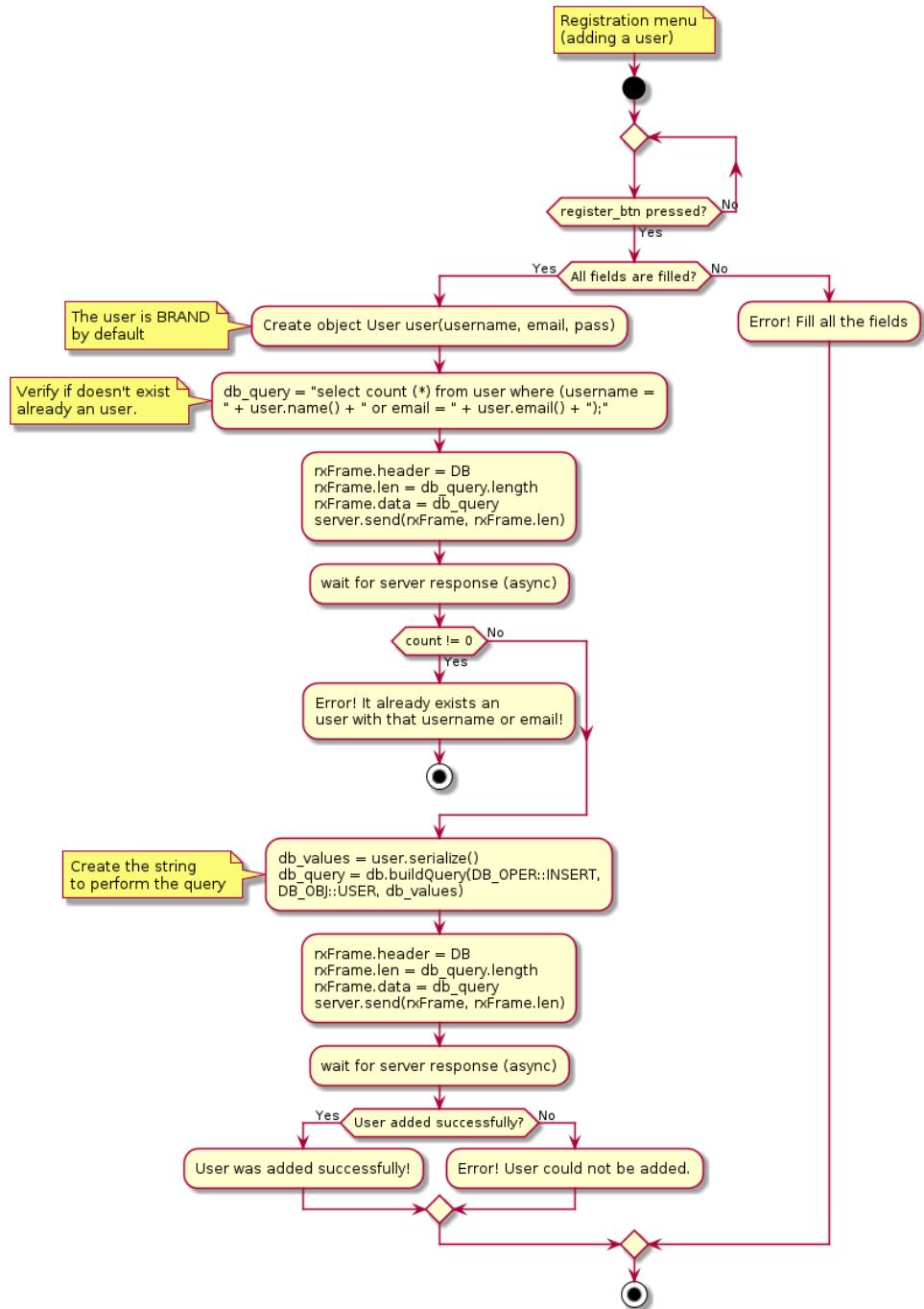


Figure 4.30.: Flowchart: Remote Client – Register

the specific time slots. Then, if everything occurs as expected, the Ad is then added to the database by a query. If that query don't occur as expected, then the timetable previously added needs to be removed from the database.

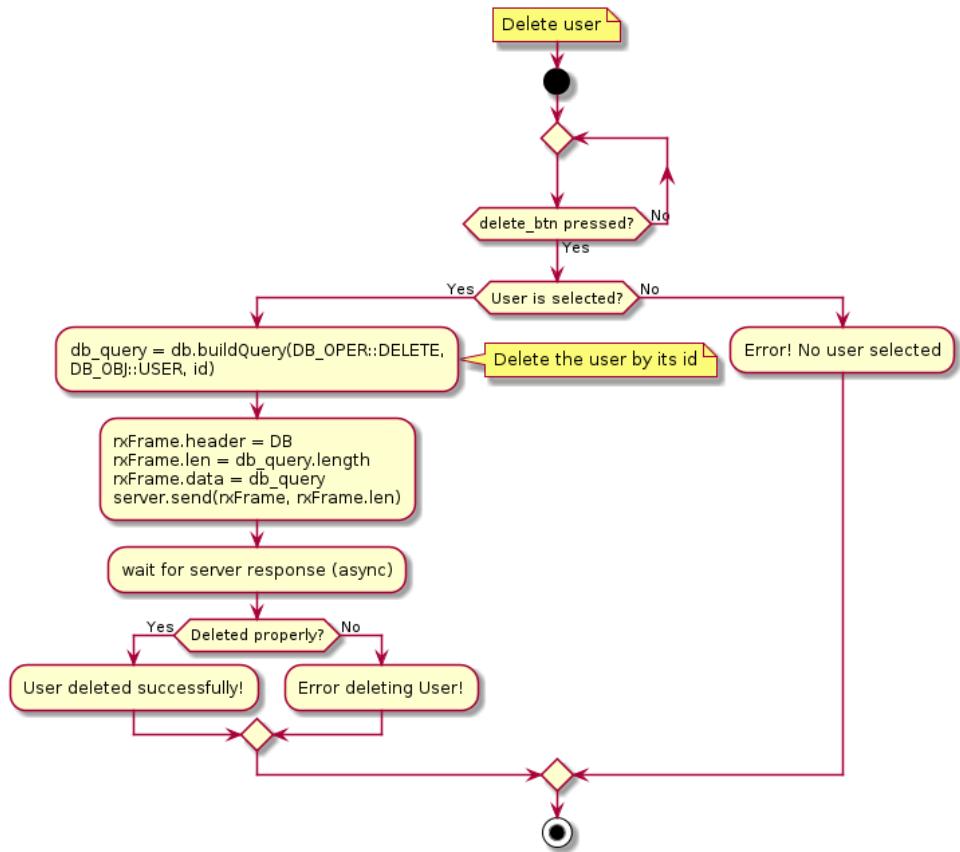


Figure 4.31.: Flowchart: Remote Client – Delete User

## Remote Server

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

Fig. 4.33 shows how the receive thread is implemented. Basically this thread waits to receive data from a socket and after that, add that received command to the vector RxVec. In this situation it is mandatory to use mutexes to avoid accessing RxVec at the same time in different threads. Also it is used a semaphore to handle the number of threads that can use this vector. The send thread is particularly the same thing, with the difference that it is necessary to know if there is something to send and send it. The information of the socket is already in the txVec and it is easy to send the information to the specific side.

Fig. 4.34 shows how the parser is executed. Basically this thread verifies if it exists messages received to read, if so, they are popped from the FIFO and it is dropped the semaphore that was being used. Then,

#### 4.4. Software specification

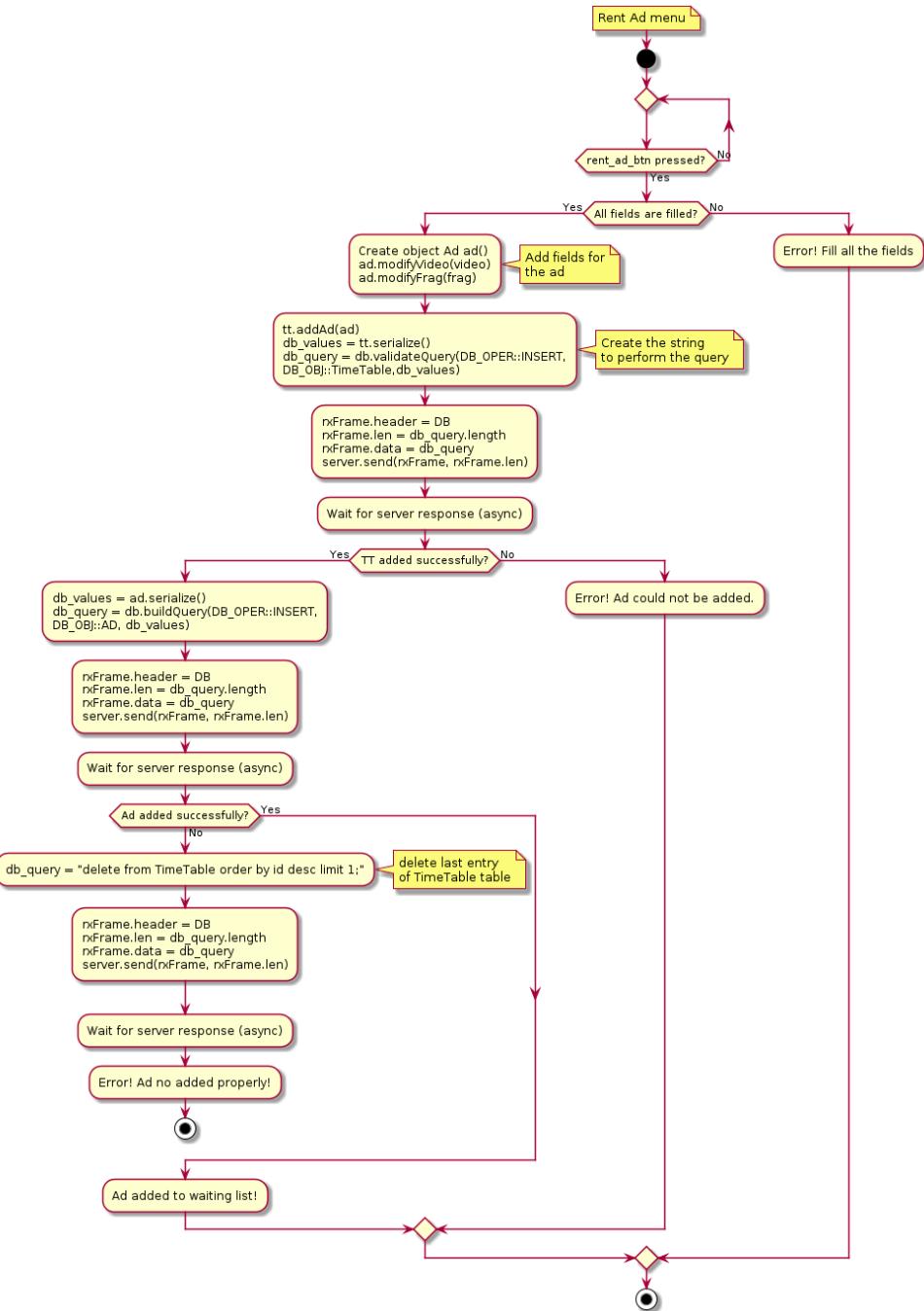


Figure 4.32.: Flowchart: Remote Client – Rent Ad

by the structure `rxFrame` it is verified if the data is a command or a database query. If it is a command, it is signalized that it is a command and it is added to the vector, if it is a database query, it does the same but for the signal of the database handle and the `db` vector. It is always mandatory to avoid collisions of data,

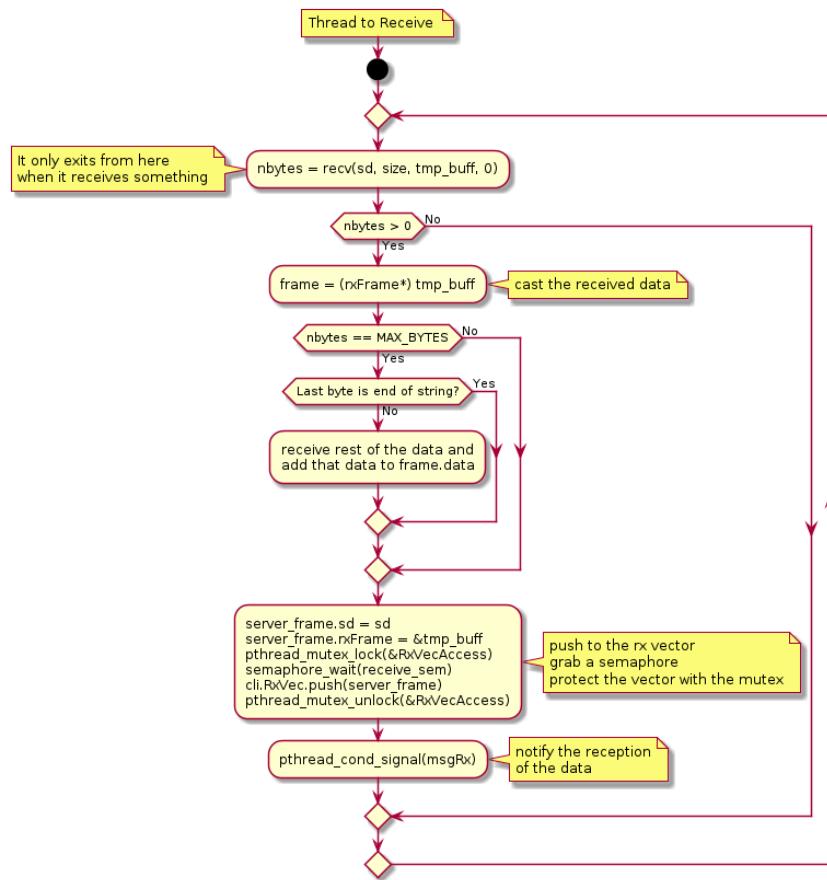


Figure 4.33.: Flowchart: Remote Server – Receive Thread

using for that the mutexes to lock the usage of the vectors.

Fig. 4.36 shows how the handleCmd thread is processed. Firstly, it is necessary to pop a command from the cmdVec and then execute it. It is necessary to know if the command is to execute in the remote server or to send to the local system. If it is to send to the local system, it is needed to make a query to the database to know the ip of the local system and then send to the machine.

Fig. 4.35 shows how the handleQuery thread is processed. Firstly, it is necessary to retrieve a query from the vector dbQueries (using the mutex to save the vector from other usage) and then make the query if no one is querying the database. After the query, it is received and stored the response to then send to the other side. So, the response is saved on the TxVec that already has the destination socket.

Fig. 4.37 illustrates how the update local system works. Basically, when some field from the database is changed and it is necessary to inform the local system to change its behavior, the database sends a trigger that is received from this thread. After that, the thread needs to make the query to the explicit database that made the trigger to ask for all the information that changed to send to the station. After receiving the

#### 4.4. Software specification

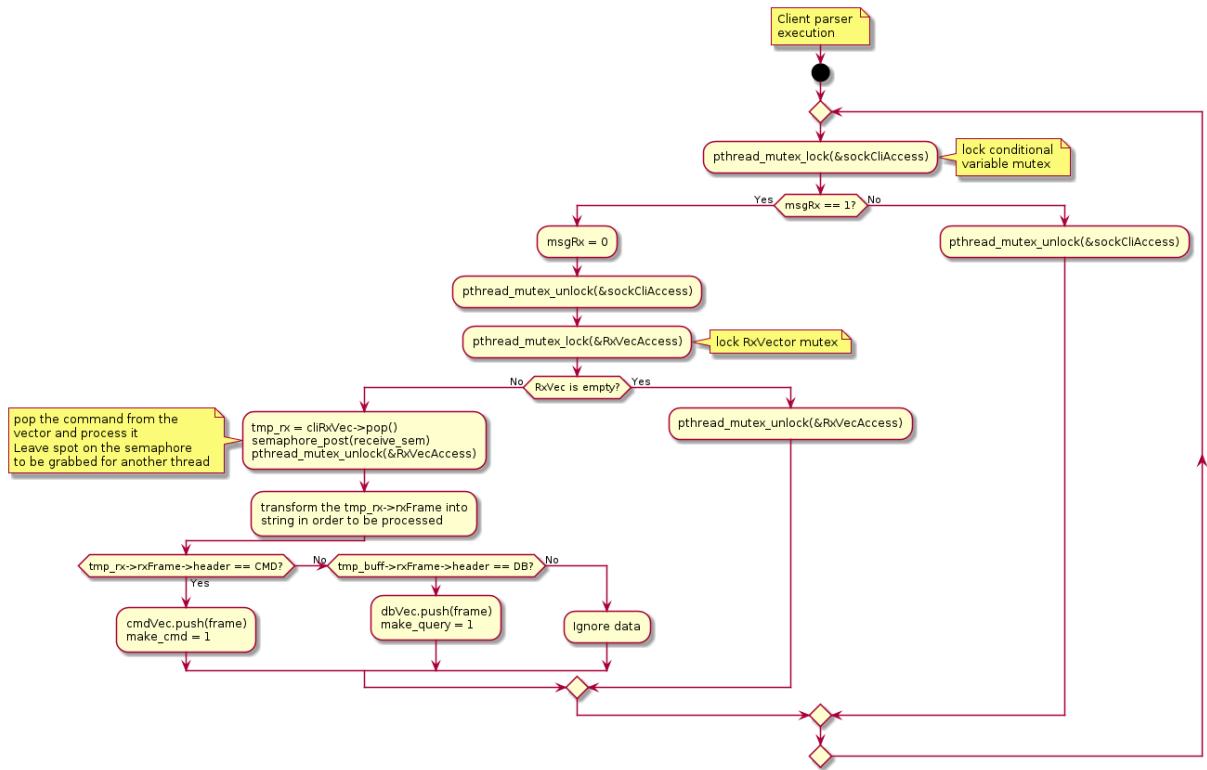


Figure 4.34.: Flowchart: Remote Server – Parser

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

### Local System

Fig. 4.38 through Fig. 4.48 present the flowcharts corresponding to the Local System threads specified in Section 4.4.6.

#### UserDetection thread

Fig. 4.38 depicts the flowchart for the UserDetection thread. It is executed periodically to check if an User was detected, consuming user detection events from a message queue that the ultrasonic sensor daemon must update.

If an User was detected, then the current value is added to a sliding window to determine if its a valid user detection. If the user was detected, this event is signaled.

#### 4.4. Software specification

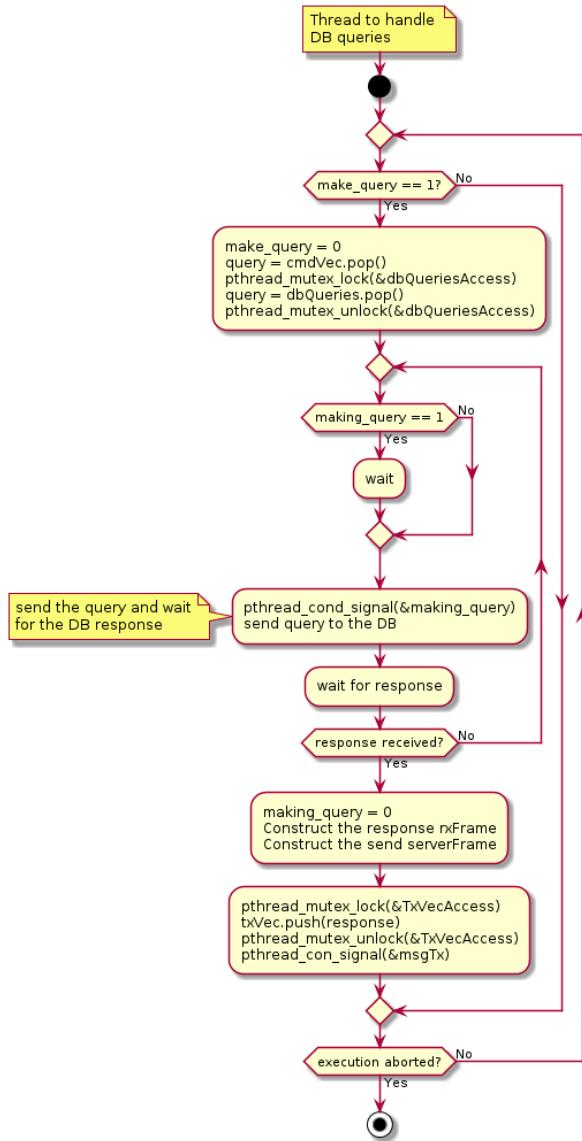


Figure 4.35.: Flowchart: Remote Server – handleQuery

#### FrameGrabber thread

Fig. 4.39 depicts the flowchart for the FrameGrabber thread. It is executed periodically, accordingly to the frame rate defined, to grab frames from camera and store them. It produces camera frames that are stored into a shared FIFO and it uses a semaphore to handle multiple consumer threads access.

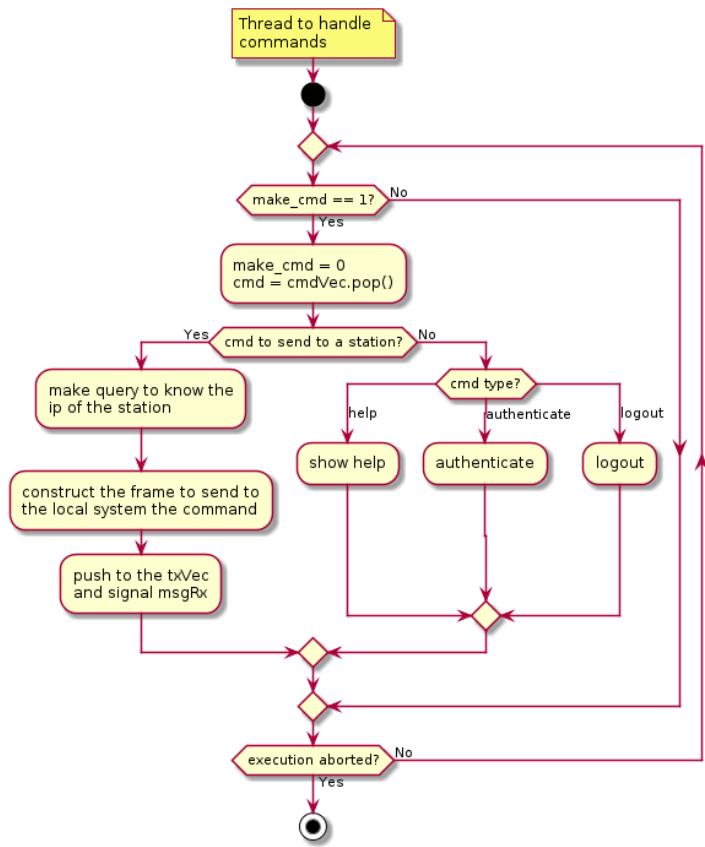


Figure 4.36.: Flowchart: Remote Server — handleCmd

### LocalRx thread

Fig. 4.40 depicts the flowchart for the LocalRx thread. It is listening on the socket used to communicate with the Remote Server for incoming data packets. When data arrives, it is validated by checking for the ACK signal. If the frame is correct and terminated it is pushed to a shared FIFO and this event is signaled for the consuming thread – AppParser.

### LocalTx thread

Fig. 4.41 depicts the flowchart for the LocalTx thread. It waits for data to be transmitted to become available, pops it from the shared FIFO and sends it to the Remote Server.

### AppParser thread

Fig. 4.42 depicts the flowchart for the AppParser thread. It waits for received data to become available, and then processes all frames in that shared FIFO. It has two main flows, depending on the frame header. If it is a command the data is casted to a cmdFrame, pushed to a shared buffer and this event is signaled

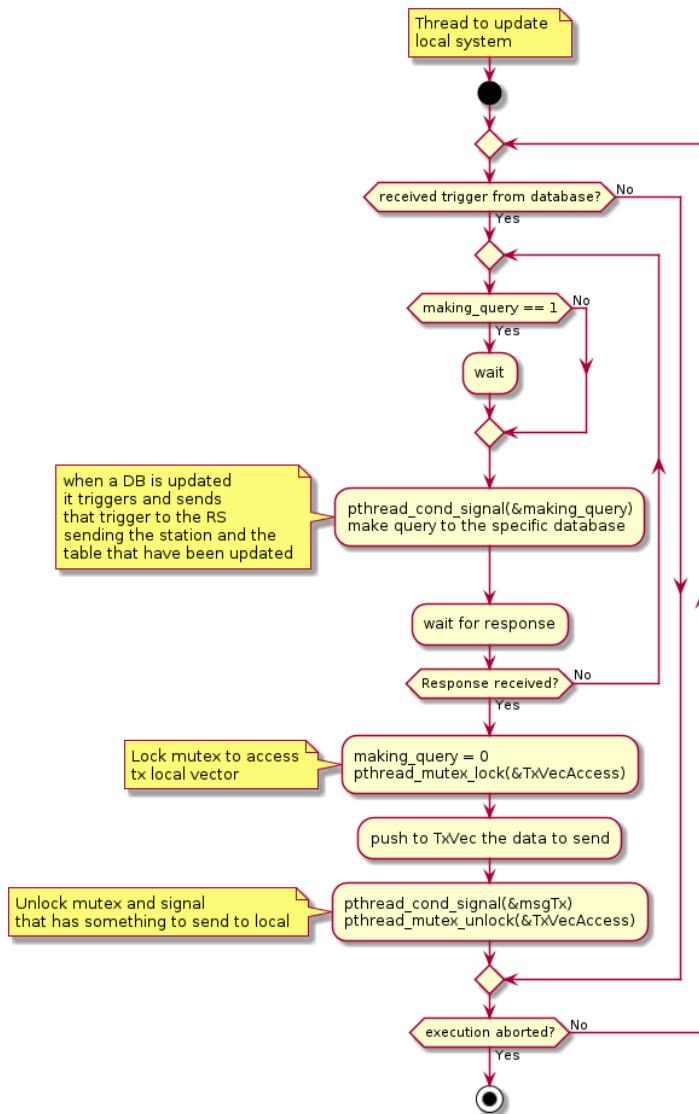


Figure 4.37.: Flowchart: Remote Server – updateLocalSys

for the CmdHandler thread to process these data. If it is an Ad, the data is casted to a adFrame, pushed to a shared buffer and this event is signaled to the FileTransfer thread so it can download media files.

### CmdHandler thread

Fig. 4.43 depicts the flowchart for the CmdHandler thread. It waits for commands to be available, and then processes all commands in that shared FIFO.

#### 4.4. Software specification

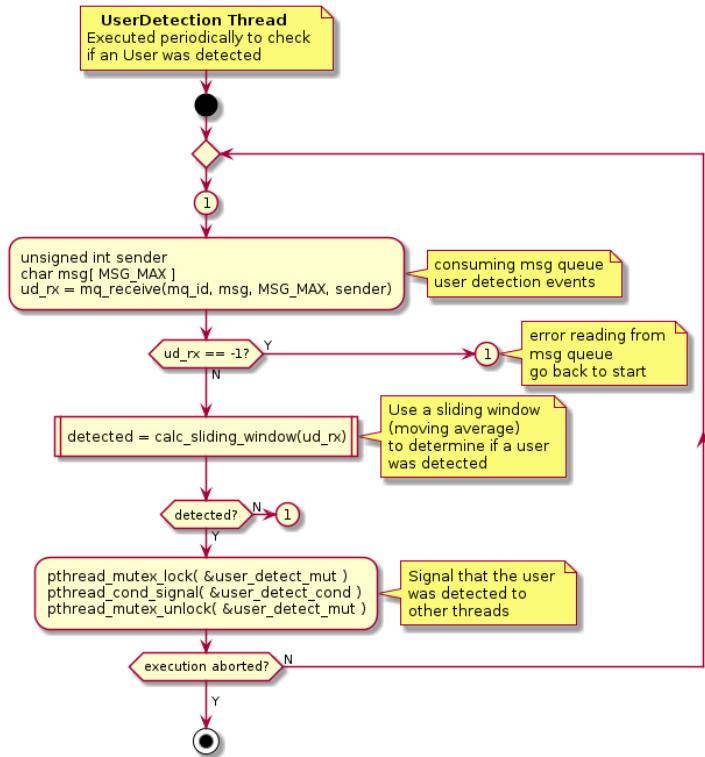


Figure 4.38.: Flowchart: Local System – UserDetect thread

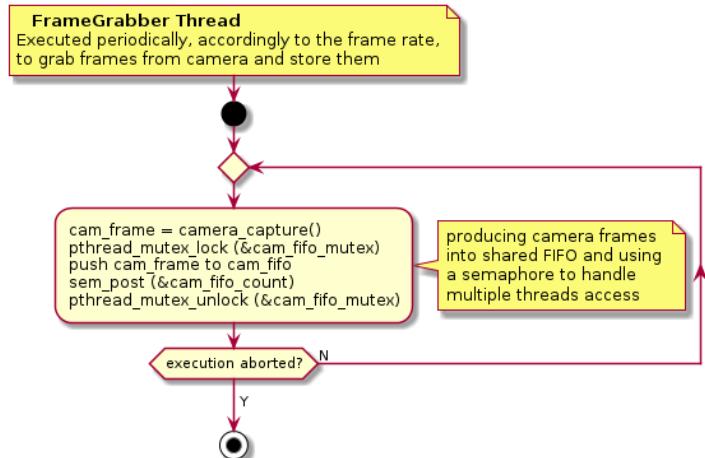


Figure 4.39.: Flowchart: Local System – FrameGrabber thread

#### TwitterShare thread

Fig. 4.44 depicts the flowchart for the TwitterShare thread. It waits for a post to be available, and then shares it on Twitter – using curl and HTTP POST method under the hood. The share status is then updated and a signal is emitted to the waiting thread – UI – to provide appropriate feedback to the User.

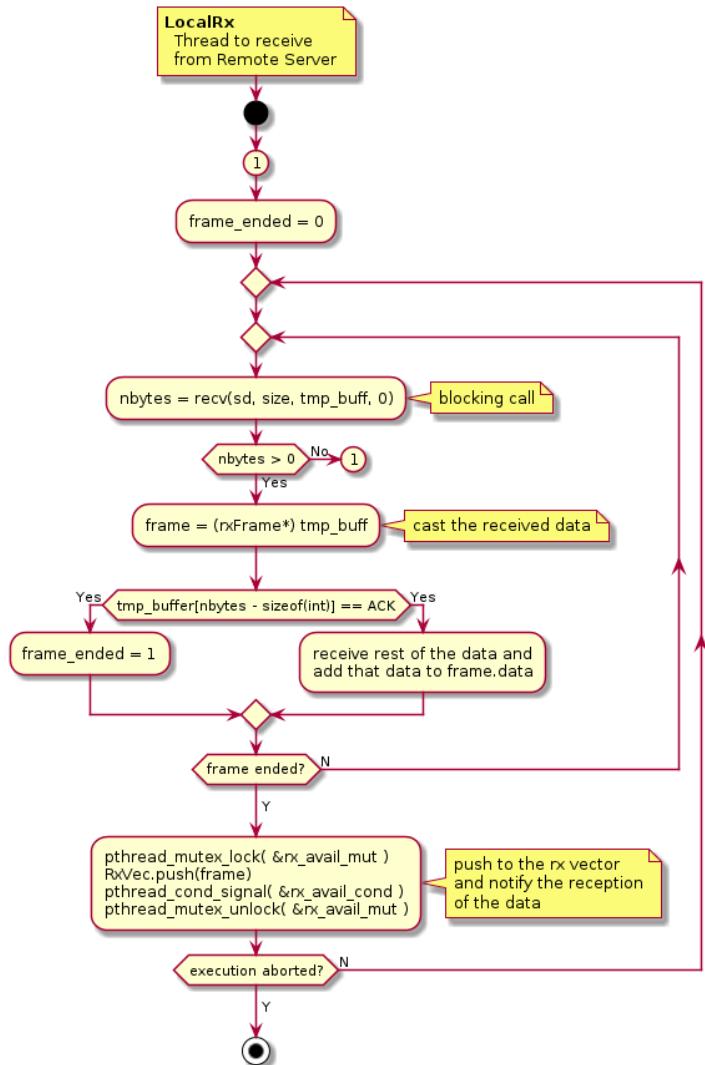


Figure 4.40.: Flowchart: Local System — LocalRx thread

### FileTransfer thread

Fig. 4.45 depicts the flowchart for the FileTransfer thread. It waits for Ad frame data to be available and then pops it. It constructs a new Ad object. It then retrieves the Comma Separated Values (CSV) string mediaURLs and tokenizes it, using the comma as the delimiter token, to extract all URLs. While there are URLs available, the file is downloaded from the URL proxy server and the filename is stored in a FIFO for later reference.

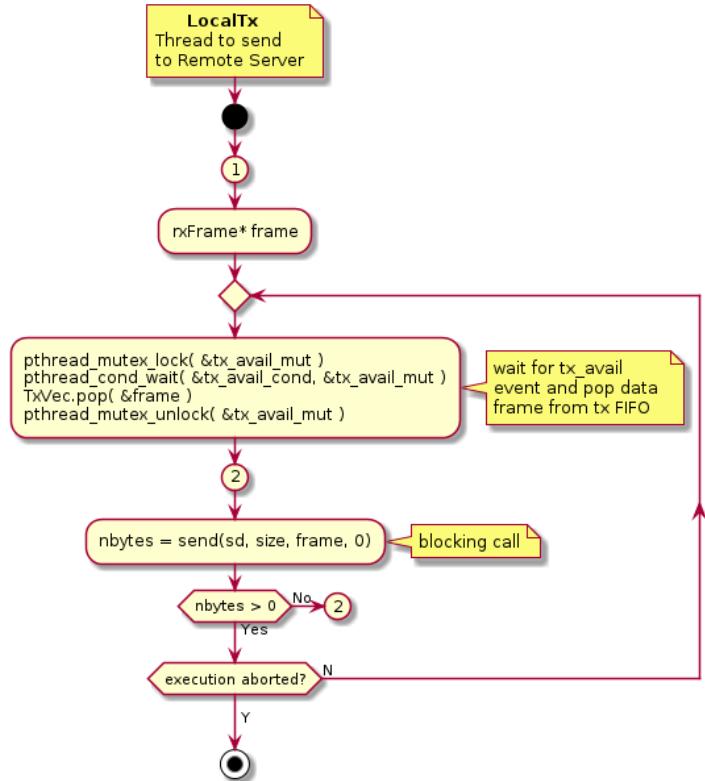


Figure 4.41.: Flowchart: Local System — LocalTx thread

### VidMan thread

Fig. 4.46 depicts the flowchart for the VidMan thread. It checks for the current mode, and if it is the normal one, it retrieves the current Ad to be played, if any. If there is an Ad to be played a counter and a timer are initialized. While an User is not detected and the timer is not elapsed, the videos are fetched from a buffer and reproduced.

### FragMan thread

Fig. 4.47 depicts the flowchart for the FragMan thread. It checks for the current mode, and if it is the normal one, it retrieves the current Ad to be played, if any. If there is an Ad to be played the on and off durations are calculated, based on the Ad intensity, the timer `tim_on` is initialized and the fragrance diffusion is enabled. While an User is not detected for the duration of the Ad, the fragrance is diffused in intervals of `tim_on` and `tim_off`.

#### 4.4. Software specification

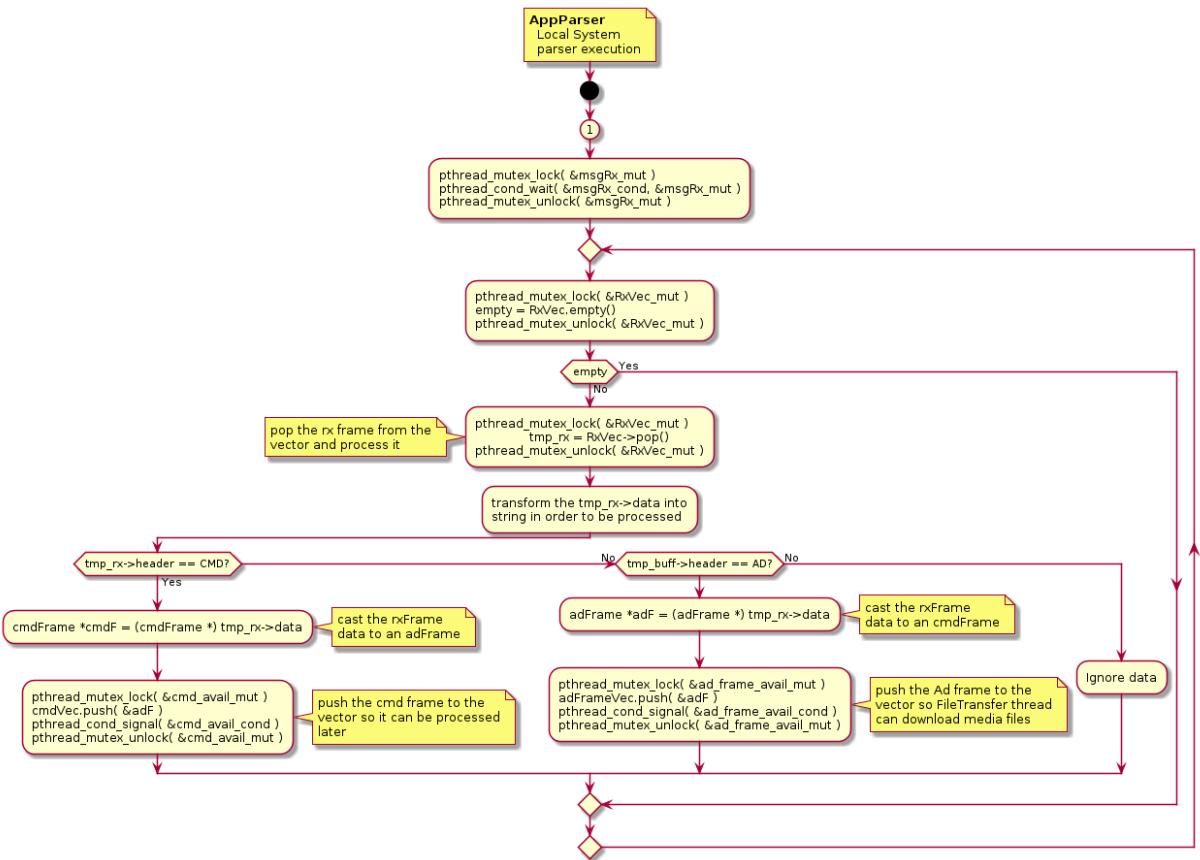


Figure 4.42.: Flowchart: Local System – AppParser thread

#### GIFGenerator thread

Fig. 4.48 depicts the flowchart for the GIFGenerator thread. It waits for an event signaled by the UI thread to start and then starts a timer. While the timer is not elapsed, the camera frame is popped from the shared FIFO and stored in an internal buffer. When the timer elapses, the GIF is stored in disk and this event is signaled to the waiting thread — the UI thread — so it can inform the user.

#### 4.4.8. Test cases

In order to verify if all the software is in good conditions, are made some functional test cases to each subsystem to see if the results combine with the expected ones.

#### **4.4. Software specification**

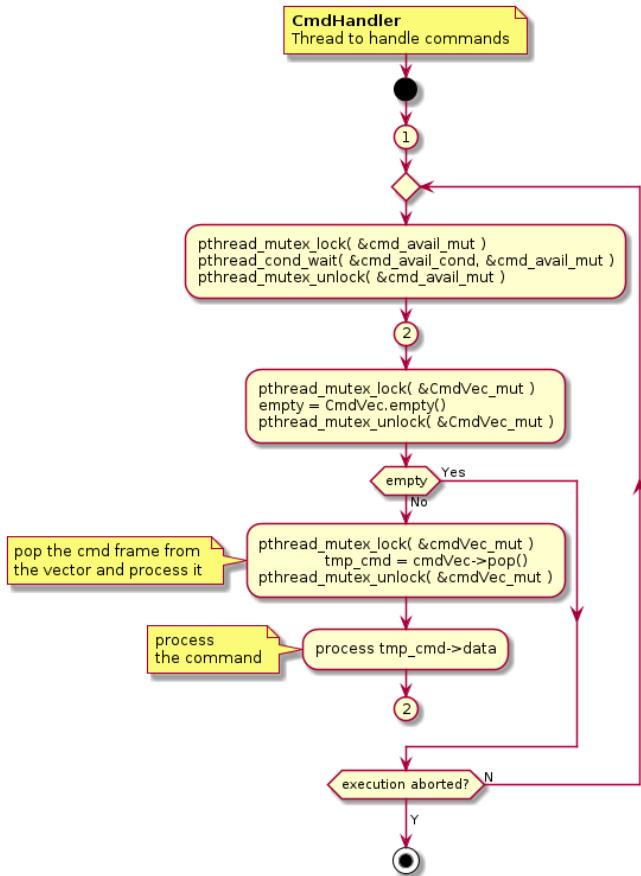


Figure 4.43.: Flowchart: Local System – CmdHandler thread

# Remote Client

In table 4.4 is depicted the test cases for the Remote Client. These test cases are mostly made through the UI verifying if this is responding as it should. Examples can be seen in the table, such as Register, Login, Logout and so on.

## Remote Server

In table 4.5 are illustrated the test cases for the Remote Server. These tests are divided in two parts: the part that belongs to the server parsing and execution commands and the part that belongs to send commands to the stations in order to test their operation. These last kind of tests can also entry on the Local System test cases, because it is needed a feedback from the MDO-L.

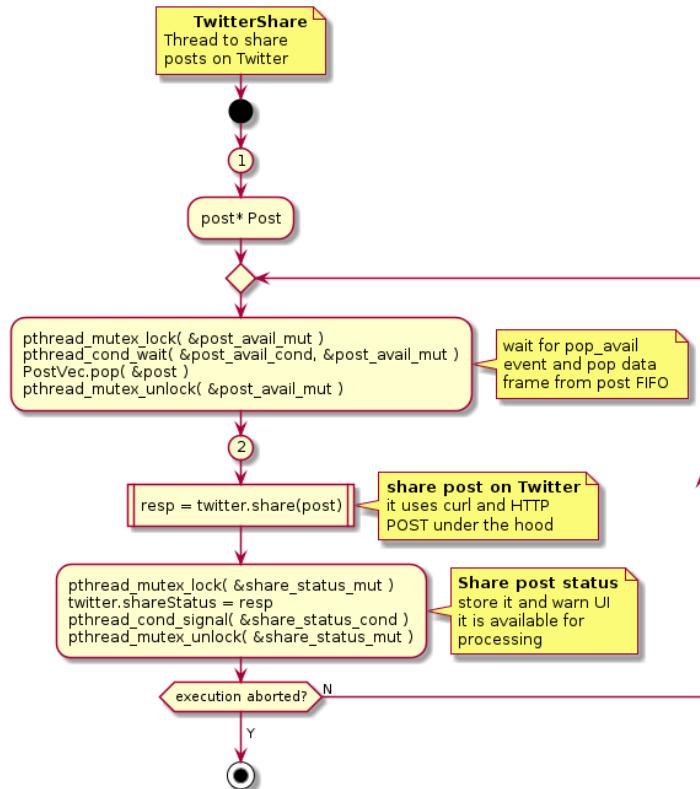


Figure 4.44.: Flowchart: Local System – TwitterShare thread

## Local System

In table 4.6 are depicted the test cases for the Local System. These test are functional and have as purpose to know if the machine is working well, if the ads are being played, if the face is being well detected and so on. Basically, it is a test operation to all functionalities of the machine.

### 4.4.9. COTS & third-party libraries

For this project there are used several tools and libraries that are open-source and helpful to develop all the subsystems.

The tools are:

1. [Plantuml](#): draw UML diagrams
2. [Make](#): and makefiles
3. [Doxygen](#): source-code documentation
4. [Qt](#): UI Framework
5. [Buildroot](#)

#### 4.4. Software specification

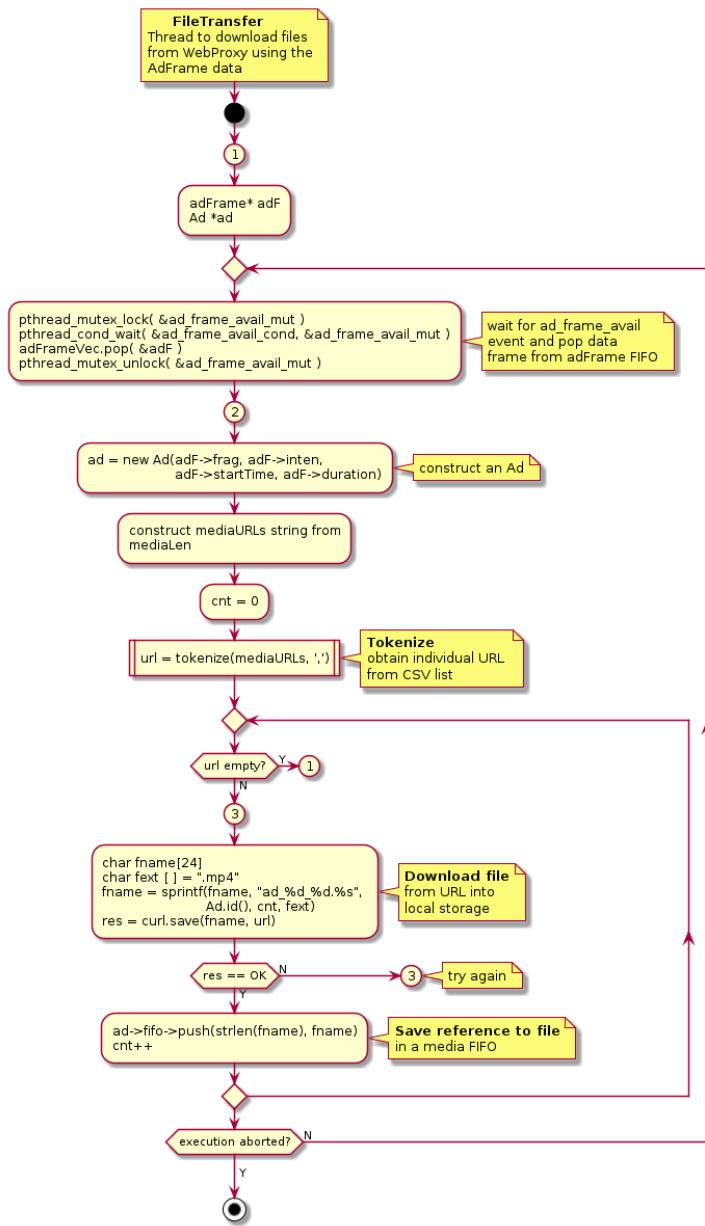


Figure 4.45.: Flowchart: Local System – FileTransfer thread

## 6. Linux

The third-party libraries used are:

1. OpenCV: computer vision
2. Magick++: GIF generation
3. MySQL: DB management
4. Twitter REST APIs

#### 4.4. Software specification

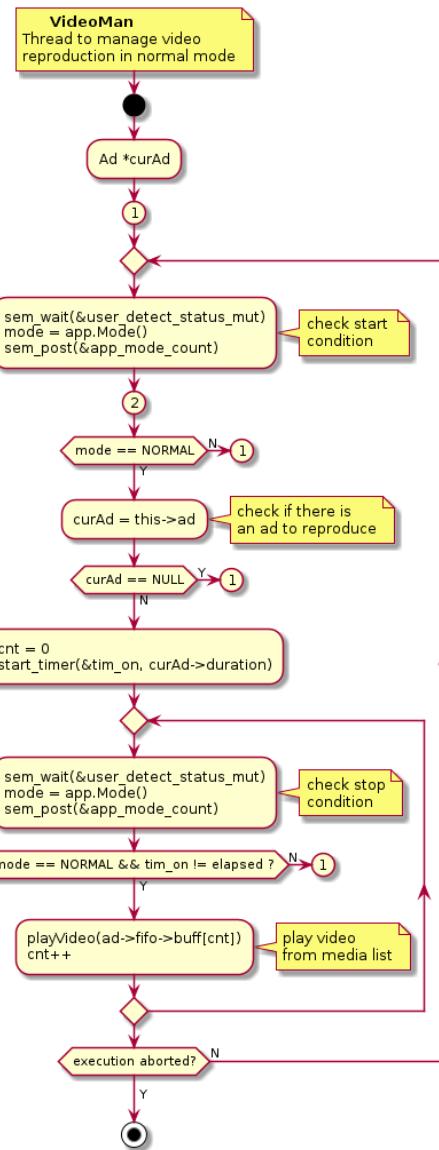


Figure 4.46.: Flowchart: Local System – VidMan thread

5. Curlpp + Transfer.sh: file transfer
6. Qt: UI framework

#### 4.4. Software specification

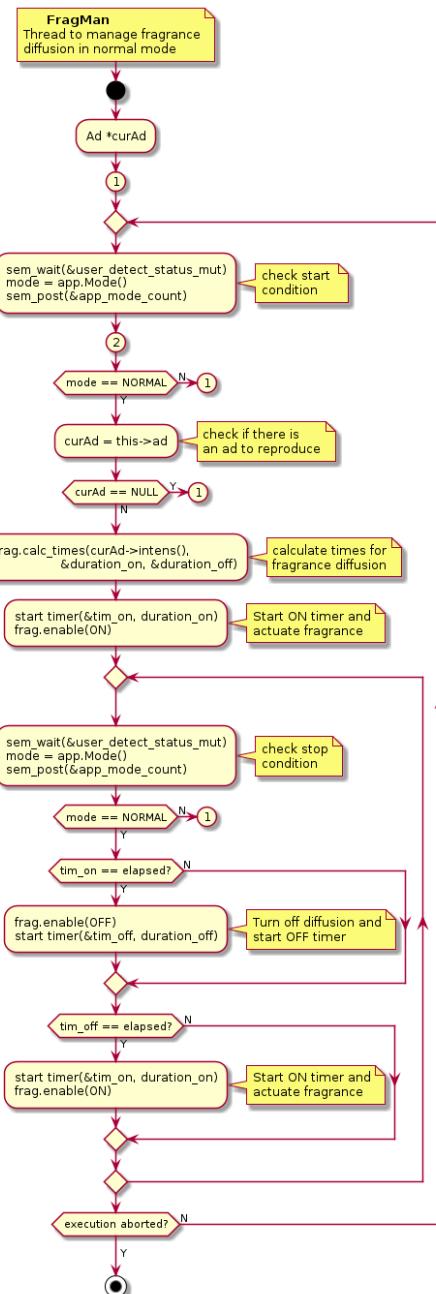


Figure 4.47.: Flowchart: Local System – FragMan thread

#### 4.4. Software specification

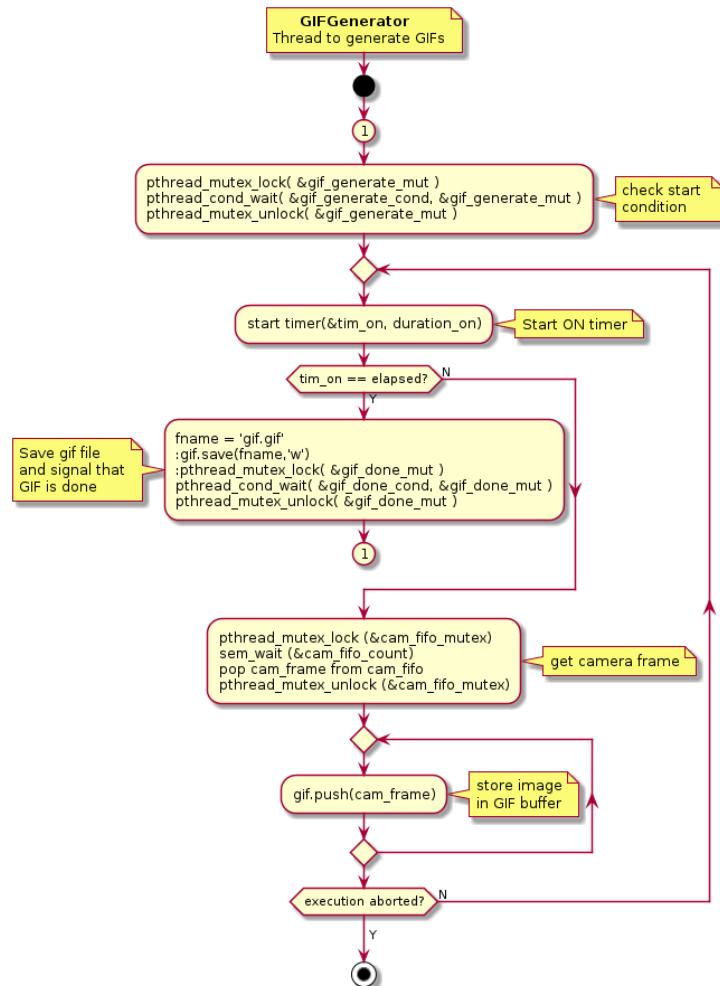


Figure 4.48.: Flowchart: Local System – GIFGenerator thread

#### 4.4. Software specification

---

Table 4.4.: Remote Client Test Cases

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

#### 4.4. Software specification

---

Table 4.5.: Remote Server Test Cases

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

#### 4.4. Software specification

---

Table 4.6.: Local System Test Cases

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

# 5. Implementation

In the implementation phase, the solution developed in the various domains is implemented into the target platforms, accordingly to the system design specification. In this chapter is presented the implementation for the various domains and subsystems identified, namely the **Remote Client**, **Remote Server** and **Local System**.

## 5.1. Hardware

The implementation of the hardware consists on mount everything that was bought and created and finish with the final prototype. This final prototype consists in a box with all the hardware inside of it, as it can be seen on fig 5.1. This also means that inside that box is the designed PCB (fig 5.2).

## 5.2. Remote Client

The MDO-RC is a Desktop application that is developed in QT. This application handles all the job that is referred to interaction between this last and a **brand** - to rent an ad or see its statistics - or an **admin** - to manage all users, stations and ads.

### 5.2.1. Classes

For a more easier developing, the layout and implementation of this application was divided by 3 distinct QWidget classes:

1. MainWindow class – to handle the login and register use cases;
2. BrandWindow class – to handle everything related to the brand usage;
3. AdminWindow class – to handle everything related to the admin usage;

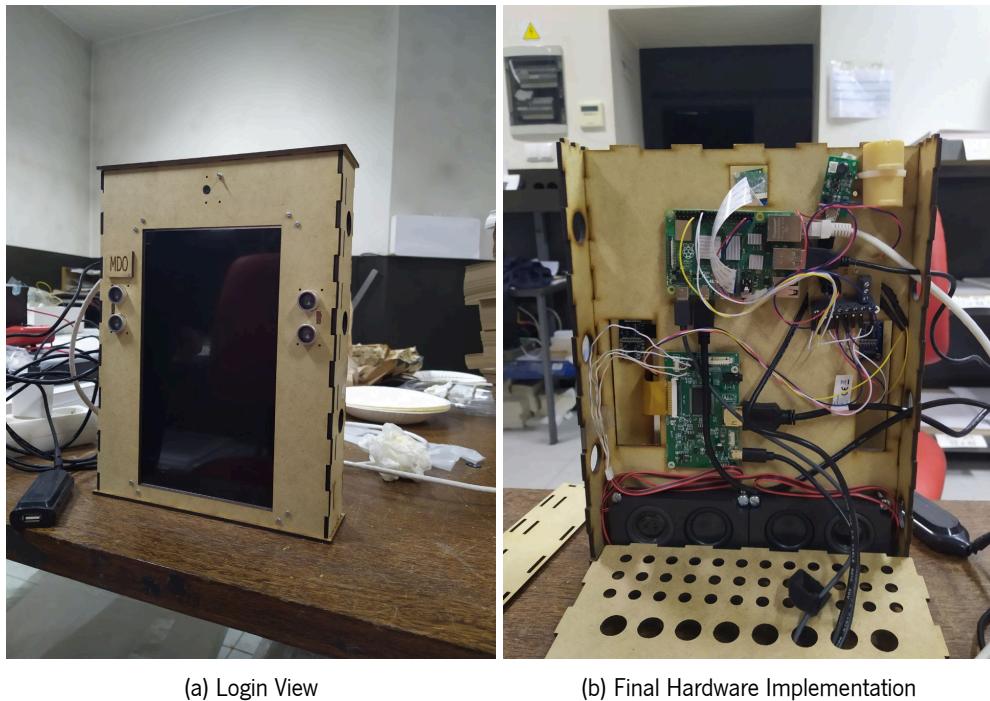


Figure 5.1.: MainWindow views

### MainWindow class

The **MainWindow** class is, as it suggests, the main class of all the program. This class handles all the transitions between views, such as Login View, Admin View, Brand View and Register View.

As it can be seen in the header file of this class (listing 5.1), there are two pointers to the other two classes, each one of them as a pointer, in order to be instantiated right on the constructor. The **Sess \*sess** that refers to the **MySQL session**, the variables associated with threads and the timer will be explained later on the next section, as well as some functions like thread functions.

```

1 #ifndef MAINWINDOW_H
2 #define MAINWINDOW_H

3 #include <QMainWindow>
4 #include <QStackedWidget>
5 #include "adminwindow.h"
6 #include "brandwindow.h"
7 #include <mysqlcpp/include/mysqlx/xdevapi.h>
8 #include <QTimer>
9 #include <pthread.h>

10 using namespace mysqlx;

```

## 5.2. Remote Client

---

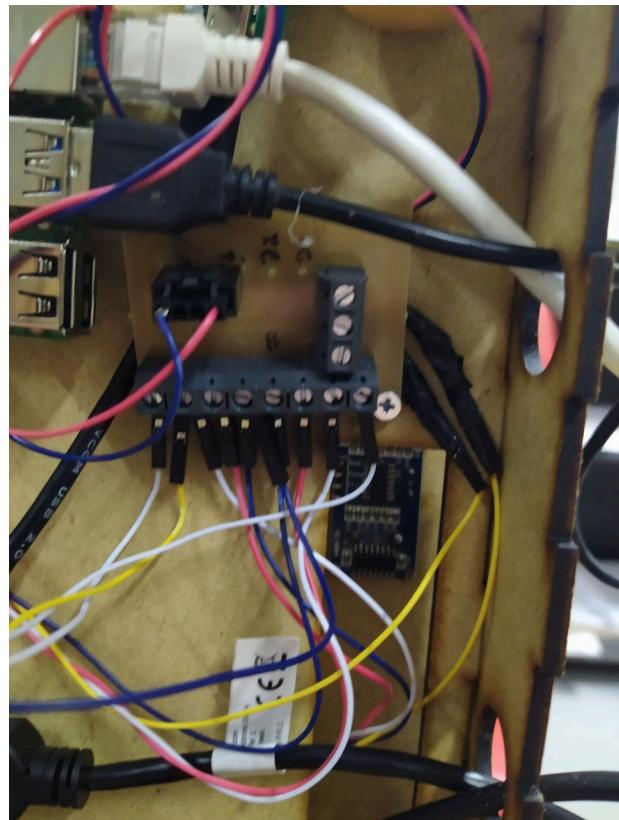


Figure 5.2.: Final PCB

```
/*
15 * @brief The Ad_t struct
*
17 * Structure that encapsulates all the data of an Ad that needs to be sent
*/
19 struct Ad_t
{
21     std::string fname; /*< File name */
22     std::string link; /*< Link to download the file */
23     std::string frag_id; /*< Fragrance ID */
24     std::string filter_id; /*< Filter ID */
25     std::string timeSlot_id; /*< Time Slot ID */
26     std::string enabled; /*< If the ad is enabled */
27 }typedef Ad;
28
29 QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
```

```

QT_END_NAMESPACE

class MainWindow : public QMainWindow
35 {
    Q_OBJECT

public:
39     MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

    /**< Mutexes */
43     pthread_mutex_t _m_connected;
    pthread_mutex_t _m_send_to_ls;
45     pthread_mutex_t _m_update_local_system;

47     /**< Conditional variables */
    pthread_cond_t _cond_update_local_system;
49     pthread_cond_t _send_to_ls;

51     /**< Connection state */
    bool connected = false;

    /**< Threads */
55     pthread_t updts;
    pthread_t server;

57 protected:
    /**
     * @brief closeEvent: Handles the event of closing the app.
     * @param event: the closing event.
     */
    void closeEvent (QCloseEvent *event);

private slots:
65     /**
     * @brief onSignInBtn_pressed: Handles the sign in press.
     */
    void on_registerBtn_pressed();
69     /**
     * @brief on_registerBtn_pressed: Handles the register press.
     */
    void on_signInBtn_pressed();
71     /**
     * @brief on_cancelBtn_pressed: Cancel the register action.
     */

```

## 5.2. Remote Client

---

```
75     */
76     void on_cancelBtn_pressed();
77 /**
78 * @brief on_registerBtn_2_pressed: registers a user if everything is in order
79 *
80 * It registers a user if it inputs a valid email (with @ and .com), if the password is
81 * correctly confirmed and if a username is written.
82 */
83     void on_registerBtn_2_pressed();
84 /**
85 * @brief on_forgotBtn_pressed: used when the user forgets his password
86 */
87     void onLogoutPressed();
88     void timeout_handler();
89     static void* update_local_system_thr(void* );
90     void upload_and_update(void* );
91     static void* server_thr(void* );
92     static void* receive_from_ls_thr(void* );
93     static void* send_to_ls_thr(void* );
94     bool isConnected();
95     void connectionEnable(bool);
96 private:
97     Ui::MainWindow *ui;
98     AdminWindow *_adminWind;
99     BrandWindow *_brandWind;
100    Session *sess;
101    QTimer *update_ls;
102    int update_ls_remaining;
103    std::vector<pthread_t> threads;
104    pthread_t receive, _send;
105    int sock;
106    Ad ad_to_send;
107 };
108 #endif // MAINWINDOW_H
```

Listing 5.1: Declaration of MainWindow class

All the function with the prefix "on\_" on it are private slots that are created to response to the click of some buttons.

Now, looking at the implementation of the Constructor:

## 5.2. Remote Client

---

```
/*
 * @brief The PageViews enum helps to handle the stackWidget in a more practical way.
 */
4 enum PageViews{
    SIGNIN = 0, /*< Sign in screen: displayed when opening the app. */
6     REGISTER, /*< To register a user. */
    ADMIN, /*< The admin view: displayed when a user is an admin */
8     BRAND, /*< The brand view: displayed when a user is a brand */
};

12 /**
 * @brief MainWindow::Constructor to the MainWindow
14 * @param parent
 *
16 * It initializes the MainWindow, creating the connection to the Database and
 * initializing all the threads, mutexes, etc that need to be initialized
18 */
MainWindow::MainWindow(QWidget *parent)
20     : QMainWindow(parent)
21     , ui(new Ui::MainWindow)
22 {
    ui->setupUi(this);

26     /*< Create session to connect to mySQL */
    sess = new Session("localhost", 33060, "root", "ESRG-MDO-Hugo-Ze@2021", "MDO");

30     /*< Initialization - Timer to update local system */
    update_ls = new QTimer(this);
    update_ls_remaining = UPDATE_LS_TIMEOUT;

34     /*< Mutexes initialization */
    pthread_mutex_init(&_m_update_local_system, NULL);
36     pthread_mutex_init(&_m_connected, NULL);
    pthread_mutex_init(&_m_send_to_ls, NULL);

38     /*< Condition variables initialization */
40     pthread_cond_init( &_cond_update_local_system, 0);
    pthread_cond_init( &_send_to_ls, 0);

44     /*< Instantiate other UI Windows */
}
```

```

44     _adminWind = new AdminWindow(sess);
45     _brandWind = new BrandWindow(sess);

46     /**< Add more UI views */
47     ui->stackedWidget->insertWidget(ADMIN, _adminWind);
48     ui->stackedWidget->insertWidget(BRAND, _brandWind);
49     ui->stackedWidget->setcurrentIndex(SIGNIN);

50
51     /**< Connect signals to slots */
52     connect(_adminWind, SIGNAL(home_pressed()), this, SLOT(onLogoutPressed()));
53     connect(_brandWind, SIGNAL(home_pressed()), this, SLOT(onLogoutPressed()));
54     connect(update_ls, SIGNAL(timeout()), this, SLOT(timeout_handler()));

55
56     /**< Start timer*/
57     update_ls->start(TIMEOUT_MS);

58
59     /**< Create thread */
60     pthread_create(&updts_ls, NULL,
61                   &MainWindow::update_local_system_thr, this);
62     pthread_create(&server, NULL,
63                   &MainWindow::server_thr, this);
64
65     /**< Receive thread*/
66     threads.push_back(receive);
67     int sd = 0;
68     pthread_create(&receive, 0, receive_from_ls_thr, this);
69     //pthread_detach(receive);

70
71     /**< Send thread*/
72     threads.push_back(_send);
73     pthread_create(&_send, 0, send_to_ls_thr, this);
74     //pthread_detach(send);

75
76 }

77 }
```

Listing 5.2: Implementation of the constructor **MainWindow**

As it can be seen in Listing 5.2, there's an enumerator that helps the class to navigate through the views. That's because that management is made with the help of **Stacked Widgets**. The **Main Window** also has a pointer to a stacked widget in order to switch between classes and show different views.

In resume, the main steps to have in considering for the **Remote Client** part are:

- Initialization of the session to connect to MySQL server;
- Instantiation of the other two classes;
- Signals connections to return back to the Main Window after a logout in each class.

### Layout

Fig. 5.3a and fig. 5.3b show respectively the Login and Register Views.

These two views interact with help of the database. For example, when registering a user, if every parameter is correctly inserted, it will be queued a message to MySQL inserting a new User with **default brand privilege**. For the example of logging in, it will be made a queue to the MySQL server asking for a row that matches de inputted username and password and if so, the user will be redirectioned to the specific view according to its privileges.

### **BrandWindow class**

The **BrandWindow** class is, as it suggests, the class that handles all of the brand's features. This class handles all the transitions between views, such as Rent Ad View, To Rent View and Logout - emits a signal to go back to login.

```

1 #ifndef BRANDWINDOW_H
2 #define BRANDWINDOW_H

4 #include <QWidget>
5 #include <mysqlcpp/include/mysqlx/xdevapi.h>
6 #include <QFileInfo>

8 using namespace mysqlx;

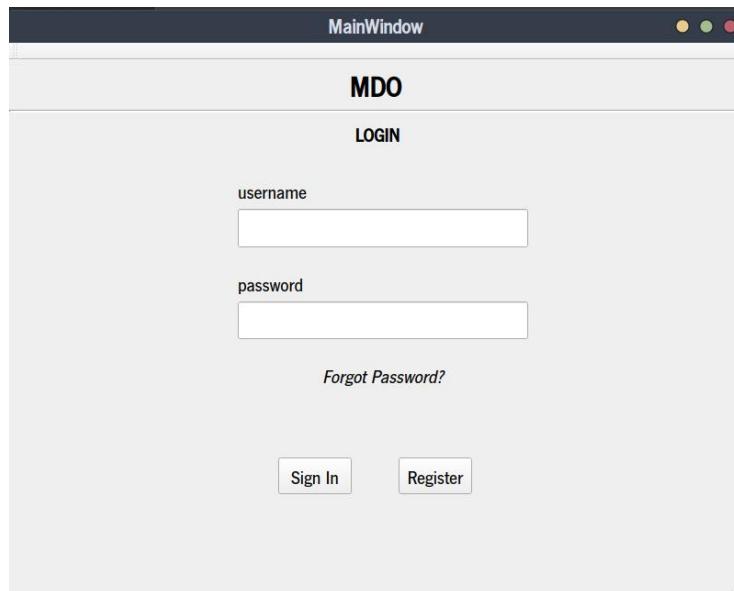
10 namespace Ui {
11     class BrandWindow;
12 }

14 class BrandWindow : public QWidget
15 {
16     Q_OBJECT

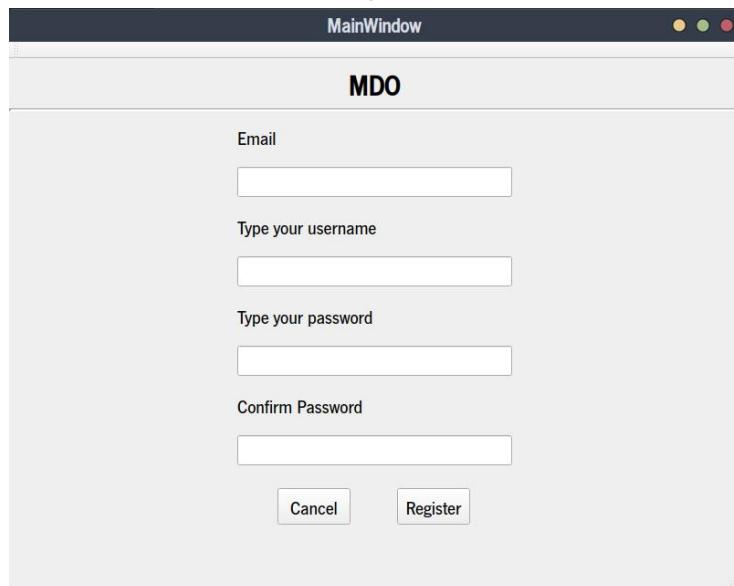
18     public:
19         explicit BrandWindow(Session *sess = nullptr, QWidget *parent = nullptr);
20         ~BrandWindow();

```

## 5.2. Remote Client



(a) Login View



(b) Register View

Figure 5.3.: MainWindow views

```
22     void setUserID(int id);  
  
24     void uploadFile(char* path);  
signals:  
26     void home_pressed(); /*< Dummy signal to indicate return to main Window */  
  
28 private slots:
```

```

void on_logoutBtn_pressed();

void on_goBackBtn_pressed();

void on_rentedBtn_pressed();

void on_stationsComboBox_activated(int index);

void on_goBackBtn_2_pressed();

void on_toRentBtn_pressed();

void on_stationsComboBox_2_activated(int index);

void on_calendarWidget_clicked(const QDate &date);

void on_uploadBtn_pressed();

void on_rentBtn_pressed();

private:
50    Ui::BrandWindow *ui;
    Session *sess;
52    QFileInfo file;
    int user_id;
54};

56 #endif // BRANDWINDOW_H

```

Listing 5.3: Declaration of BrandWindow class

As it can be seen in the header file of this class (listing 5.3), there's also the session to handle the MySQL server connection (`Sess* sess`), a `QFileInfo` to handle the files to upload on an Ad and the `user_id` which refers to the user that is currently using the app. As in every part of the code, here there's also the usage of queues to the MySQL server, to rent ads and to see the ads rented to watch its statistics. In order to make everything more simpler, in the Rent View it is only possible to rent for a range of one week. All the time slots that are already rented appear with a "Rented" `std::string` and the days that do not correspond to the time space available to rent appear with the label "Unavailable".

Perhaps, one of the most difficult functions to implement was the `rent` function, that's because it is necessary to add many rows to different tables in order to build a complete Ad on the database, this means

## 5.2. Remote Client

---

that if something went wrong in the middle of the execution, it is necessary to do the inverse process to delete the already inserted rows, because they will not have a link to every tables.

Listing 5.4 shows the rent ad function.

```
void BrandWindow::on_rentBtn_pressed()
{
    QString text = "Rent Description:\n"
                  "Station: " + ui->stationsComboBox_2->currentText() +
                  "\nDate: " + ui->calendarWidget->selectedDate().toString() +
                  "\nTarget Hours: " + ui->targetHoursComboBox->currentText() +
                  "\nFragrance: " + ui->fragranceComboBox->currentText() +
                  "\nFilter: " + ui->filterComboBox->currentText() +
                  "\nFile: " + ui->filenameLabel->text();
    std::string station_name = ui->stationsComboBox_2->currentText().toStdString();
    /**< Needs to upload a mp4 file*/
    if(ui->filenameLabel->text().isEmpty()) {
        QMessageBox::warning(this, "ERROR!", "Upload a .mp4 file!", QMessageBox::Ok);
    }
    /**< It can not be already rented*/
    else if(ui->targetHoursComboBox->currentText() == "Rented") {
        QMessageBox::warning(this, "ERROR!", "Slot already rented!", QMessageBox::Ok);
    }

    else {
        /**< Confirm the Ad rent */
        auto resbtn = QMessageBox::question(this, "WARNING", "Are you sure of the rent?\n" +
                                             text, QMessageBox::Yes | QMessageBox::No);
        if(resbtn == QMessageBox::Yes) {
            std::stringstream ss1, ss2, ss3, ss4;
            /**< Get Fragrance ID (Through station id and fragrance list id) */
            Row Station = sess->sql(StationDB::select_where + "name = '" + station_name + "'").
                execute().fetchOne();
            int station_id = Station[StationDB::ID];
            ss1 << station_id;
            std::string station_id_str = ss1.str();
            Row FragList = sess->sql(FragranceListDB::select_from_station + station_id_str).
                execute().fetchOne();
            int fragList_id = FragList[FragranceDB::ID];
            ss2 << fragList_id;
            std::string fragList_id_str = ss2.str();
            Row Fragrance = sess->sql(FragranceDB::select_from_fragranceList +
                                         fragList_id_str).execute().fetchOne();
            int frag_id = Fragrance[FragranceDB::ID];
        }
    }
}
```

## 5.2. Remote Client

---

```
36         ss3 << frag_id;
40         std::string fragrance_id_str = ss3.str();

44         /**< Get user id */
48         ss4 << user_id;
52         std::string user_id_str = ss4.str();

56         /**< Insert new timeSlot and get its id */
60         std::stringstream ss5, ss6, ss7;
64         int week = ui->calendarWidget->selectedDate().weekNumber();
68         ss5 << week;
72         std::string week_str = ss5.str();
76         Row TimeTable = sess->sql(TimeTableDB::select_from_station + station_id_str + "
80             and week = " + week_str).execute().fetchOne();
84         int timeTable_id = TimeTable[TimeTableDB::ID];
88         ss6 << timeTable_id;
92         std::string timeTable_id_str = ss6.str();
96         int time_slot_id = ui->targetHoursComboBox->currentIndex();
100        int day_week = ui->calendarWidget->selectedDate().dayOfWeek() - 1;
104        time_slot_id = 24*day_week + ui->targetHoursComboBox->currentIndex();
108        ss7 << time_slot_id;
112        std::string timeSlot_id_str = ss7.str();
116        try {
120            sess->sql(TimeSlotDB::insert + timeSlot_id_str + ", " + timeTable_id_str + ",
124                60, 20)").execute();
128        }
132        catch(const mysqlx::Error &err) {
136            QMessageBox::information(this, "ERROR!!", "Something went wrong!",
140                QMessageBox::Ok);
144            return;
148        }

152         /**< Get filter id */
156         std::stringstream ss8;
160         Row Filter = sess->sql(FilterDb::select_where + "name = '" + ui->filterComboBox->
164             currentText().toStdString() + "'").execute().fetchOne();
168         int filter_id = Filter[FilterDb::ID];
172         ss8 << filter_id;
176         std::string filter_id_str = ss8.str();

180         /**< Adding the Ad */
184         try {
188             sess->sql(AdDB::insert + fragrance_id_str + ", " + user_id_str + ", " +
192                 "
```

## 5.2. Remote Client

---

```
        timeSlot_id_str + ", " + station_id_str + ", " + filter_id_str + ", false
    )").execute();
}

/**< Delete everything that was added if the query doesn't work*/
catch (const mysqlx::Error &err){
    sess->sql("Delete from TimeSlot where id = " + timeSlot_id_str).execute();
    QMessageBox::information(this, "ERROR!!", "Something went wrong!",
                            QMessageBox::Ok);
    return;
}

/**< Adding the file */
std::stringstream ss9, ss10;
qint64 file_size = file.size();
ss9 << file_size;
std::string file_size_str = ss9.str(),
file_name = file.fileName().toStdString();
Row Ad = sess->sql("SELECT * FROM Ad WHERE id = (SELECT max(id) from Ad)").execute().fetchOne();
int ad_id = Ad[AdDB::ID];
ss10 << ad_id;
std::string ad_id_str = ss10.str();
std::string file_type = file.completeSuffix().toStdString();
try {
    sess->sql(MediaFileDb::insert + ad_id_str + ", '" + file_name + "'", " +
file_size_str + ", '" + file_type + "')").execute();
}
/**< Delete everything that was added if the query doesn't work*/
catch(const mysqlx::Error &err) {
    sess->sql("Delete from TimeSlot where id = " + timeSlot_id_str).execute();
    sess->sql("Delete from Ad where id = (SELECT id FROM Ad ORDER BY id DESC
LIMIT 1)").execute();
    QMessageBox::information(this, "ERROR!!", "Something went wrong!",
                            QMessageBox::Ok);
    return;
}
Row User = sess->sql(UserDB::select_where + "id =" + user_id_str).execute().fetchOne();
std::string user_name = static_cast<std::string>(User[UserDB::NAME]);
QString curr_path = file.absoluteFilePath();
QString new_path = QCoreApplication::applicationDirPath() + "/../res/";
if (!QDir(new_path).exists()) {
    QDir().mkdir(new_path);
}
```

## 5.2. Remote Client

---

```
110     /**< Send the file to a specific location */
111     if( QFile :: copy( curr_path , new_path + file.fileName() ) );
112     //char* path;
113     //strcpy(path, curr_path.toStdString().c_str());
114     //uploadFile(path);
115
116     QMessageBox :: information( this , "SUCCESS" , "Ad rented!" , QMessageBox :: Ok );
117     // std::cout << "Ad rented!!" << std::endl;
118     on_toRentBtn_pressed();
119
120 }
121 }
122 }
```

Listing 5.4: Implementation of rent function on BrandWindow class

Because of the length and process capability of this function, every little mistake can take to serious consequences, so, it is mandatory to make all and every validation in order to avoid every mistake in the MySQL queue messages. Common errors can be such like redundant blocs of data, or even just a single piece of data.

### Layout

Figures 5.4a, 5.4c and 5.4b show the layout used for all the Brand Views.

### **AdminWindow class**

The AdminWindow class is, as it suggests, the class that handles all of the admin's features. This class handles all the transitions between views, such as Statistics View, Manage Users View, Ads To Activate View and Test Operation View.

```
#ifndef ADMINWINDOW_H
#define ADMINWINDOW_H

#include <QWidget>
#include <mysqlcpp/include/mysqlx/xdevapi.h>

using namespace mysqlx;

namespace Ui {
class AdminWindow;
```

## 5.2. Remote Client



Figure 5.4.: BrandWindow views

```

}

class AdminWindow : public QWidget
14 {
    Q_OBJECT

public:
18     explicit AdminWindow(Session *sess = nullptr, QWidget *parent = nullptr);
    ~AdminWindow();

    void setUserID(int id);

22 signals:
    void home_pressed(); /*< Dummy signal to indicate return to main Window */

private slots:
26     void on_activatedCheckBox_stateChanged(int arg1);

```

## 5.2. Remote Client

---

```
28     void on_goBackBtn_pressed() ;  
  
30     void on_statisticsBtn_2_pressed() ;  
  
32     void on_stationsComboBox_activated(int index) ;  
  
34     void on_logoutBtn_pressed() ;  
  
36     void on_poweredCheckBox_stateChanged(int arg1) ;  
  
38     void on_brandComboBox_activated(int index) ;  
  
40     void on_usersBtn_2_pressed() ;  
  
42     void on_goBackBtn_2_pressed() ;  
  
44     void on_usersComboBox_activated(int index) ;  
  
46     void on_saveChangesBtn_pressed() ;  
  
48     void on_discardChangesBtn_pressed() ;  
  
50     void on_goBackBtn_3_pressed() ;  
  
52     void on_adsToActivateBtn_2_pressed() ;  
  
54     void on_testOperationBtn_2_pressed() ;  
  
56     void on_goBackBtn_4_pressed() ;  
  
58     void on_transferVideosBtn_pressed() ;  
  
60     void on_acceptAdBtn_pressed() ;  
  
62     void on_denyAdBtn_pressed() ;  
  
64     void on_adsToActComboBox_activated(int index) ;  
  
66     void on_deleteUserBtn_pressed() ;  
  
68 private :  
    Ui::AdminWindow *ui ;  
    Session *sess ;
```

```

    int user_id;
72 };

74 #endif // ADMINWINDOW_H

```

Listing 5.5: Declaration of AdminWindow class

In terms of private attributes, this class is similar to the **BrandWindow** class: it has a **Sess \*sess** to handle the communication to MySQL server and it also has a **user\_id** to know which user is using the app.

Unfortunately, in this case it could not be implemented the Test Operation View because of reasons that will be explained in the next section.

As always, there's a lot of message queues to MySQL in order to handle all the data and show them in the MDO-RC. The most interesting feature in this class is the **Manage Users** function. In this view it is possible to change the permissions of a user, turning into an Admin or a Brand in just a click. It is also possible to delete users, which can be helpful to delete some bots or even users with bad conduct. Also, the **Ads to Activate** is an interesting feature, because it can turn ads into active mode or, if denied, it can remove them. This feature is important once it is important to validate all data that is going to be sent to the machines, in order to avoid some type of non-advertisement videos that could appear in the stations.

Listing 5.6 shows the function **Manage Users**.

```

/**
2  * @brief AdminWindow :: on_usersBtn_2_pressed
*
4  * Event that switches to the Users page and shows all the users and its characteristics
*/
6 void AdminWindow :: on_usersBtn_2_pressed ()
{
8  /**< Remove all previously items added to the combo boxes */
10   for( int i = ui -> usersComboBox -> count () ; i >= 0; i --) {
11     ui -> usersComboBox -> removeItem (i);
12   }
12   for( int i = ui -> userTypeComboBox -> count () ; i >= 0; i --) {
13     ui -> userTypeComboBox -> removeItem (i);
14   }

16   ui -> userTypeComboBox -> addltem ( "BRAND" );
17   ui -> userTypeComboBox -> addltem ( "ADMIN" );

20   /**< Query to search for all the users */
21   SqIResult users = sess -> sql (UserDB :: select_all) . execute ();

```

## 5.2. Remote Client

```
    Row user;
22   std::string name, pass, email;
23   while(user = users.fetchOne()) {
24       name = static_cast<std::string>(user[UserDB::NAME]);
25       ui->usersComboBox->addItem(QString::fromStdString(name));
26   }
27   /*< Show the characteristics of the first user selected on the combo box */
28   users = sess->sql(UserDB::select_all).execute();
29   user = users.fetchOne();
30   ui->userTypeComboBox->setcurrentIndex(user[UserDB::ROLE]);
31   email = static_cast<std::string>(user[UserDB::EMAIL]);
32   ui->emailLabel_2->setText(QString::fromStdString(email));
33   ui->stackedWidget->setcurrentIndex(USERS);
34 }
```

Listing 5.6: Implementation of Manage Users from AdminWindow class

Listing 5.7 shows the **Ads to Activate** feature.

```
/*
2  * @brief AdminWindow::on_adsToActivateBtn_2_pressed
3  *
4  * Event to show all the ads that need to be activated or rejected by the admin
5  */
6 void AdminWindow::on_adsToActivateBtn_2_pressed()
{
7     ui->adsToActivateLabel->setText("Ads To Activate: " + ui->stationsComboBox->currentText());
8
9     for(int i = ui->adsToActComboBox->count(); i >= 0; i--)
10        ui->adsToActComboBox->removeitem(i);
11
12     /*< Get the selected station from the Database */
13     Row station = sess->sql(StationDB::select_where + "name = '" + ui->stationsComboBox->
14                               currentText().toStdString() + "'").execute().fetchOne();
15     int station_id = station[StationDB::ID];
16
17     std::stringstream ss1;
18     ss1 << station_id;
19     std::string station_id_str = ss1.str();
20
21     /*< Get from the Database all the ads from that station that are not activated */
22     QSqlResult Ads = sess->sql(AdDB::select_where + "station_id = " + station_id_str + " and
23                                 active = false").execute();
24
25     Row ad;
26     while(ad = Ads.fetchOne()) {
27         int user_id = ad[AdDB::USER_ID];
```

## 5.2. Remote Client

---

```
22     std::stringstream ss2;
23     ss2 << user_id;
24     std::string user_id_str = ss2.str();
25     Row user = sess->sql(UserDB::select_where + "id = " + user_id_str).execute().fetchOne();
26     ui->adsToActComboBox->addItem(QString::fromStdString(static_cast<std::string>(user[UserDB::NAME])));
27 }
28 /*< Show the informations of the first add in the list*/
29 ad = sess->sql(AdDB::select_where + "station_id = " + station_id_str + " and active = false").execute().fetchOne();
30 if (!ad.isNull()) {
31     int fragrance_id = ad[AdDB::FRAGRANCE_ID];
32     std::stringstream ss2;
33     ss2 << fragrance_id;
34     std::string fragrance_id_str = ss2.str();
35     Row Fragrance = sess->sql(FragranceDB::select_where + "id = " + fragrance_id_str).execute().fetchOne();
36     QString frag_slot = QString::fromStdString(static_cast<std::string>(Fragrance[FragranceDB::NAME]));
37     ui->stateLabel->setText("READY");
38     ui->fragSlotLabel_2->setText(frag_slot + " | 100%");
39     int ad_id = ad[AdDB::ID];
40     std::stringstream ss3;
41     ss3 << ad_id;
42     std::string ad_id_str = ss3.str();
43     Row media_file = sess->sql(MediaFileDb::select_where + "ad_id = " + ad_id_str).execute().fetchOne();
44     QString media_file_name = QString::fromStdString(static_cast<std::string>(media_file[MediaFileDb::FILENAME]));
45     ui->transferVideosBtn->setText(media_file_name);
46     int filter_id = ad[AdDB::FILTER_ID];
47     std::stringstream ss4;
48     ss4 << filter_id;
49     std::string filter_id_str = ss4.str();
50     Row Filter = sess->sql(FilterDb::select_where + "id = " + filter_id_str).execute().fetchOne();
51     QString filter_name = QString::fromStdString(static_cast<std::string>(Filter[FilterDb::NAME]));
52     ui->filterLabel_2->setText(filter_name);
53 }
54 ui->stackedWidget->setcurrentIndex(ADS_TO_ACT);
55 }
```

### 5.3. Remote Server

Listing 5.7: Implementation of Ads TO Activate from AdminWindow class

#### Layout

On figures 5.5a, 5.5b and 5.5c are all the views from the Admin Views.

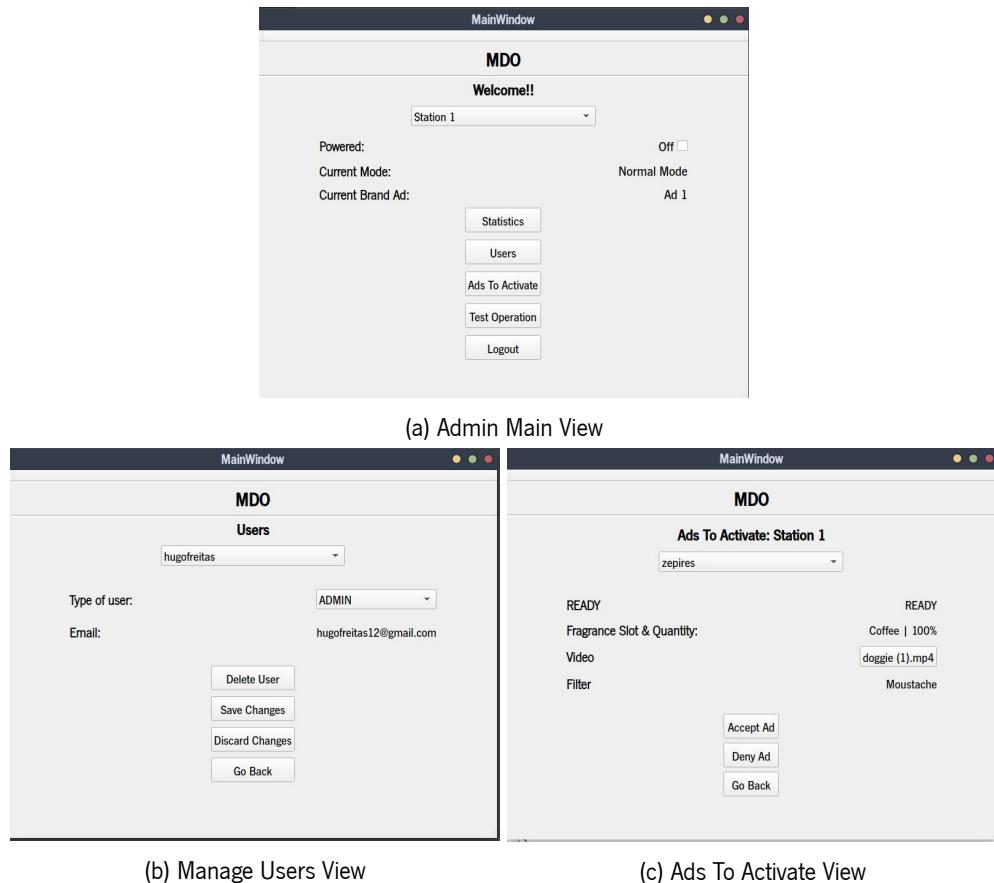


Figure 5.5.: AdminWindow views

## 5.3. Remote Server

The MDO-RS was one of the subsystems previously designed to be implemented. This application would give a more distributed architecture to the overall system.

Unfortunately, because of laps in time spent trying to resolve some unexpected issues and for many other variants, it turned out that it was not possible to develop a real Remote Server. So, in terms to make everything work at its minimums, it was implemented a "kind of" Remote Server in the Remote Client. This improvised Remote Server has a full database running in MySQL Server (as it could be seen before) and it has threads to create connections with clients - in this case, local systems - and send to them the information needed, such as new Ads information.

### 5.3.1. Data Base

The Data Base is one of the most important parts of this system for obvious reasons: this keeps all data stored and easy to access from the Remote Client and the Remote server. Throughout its implementation, it suffered various iterations that were being stored in two different scripts, these scripts are `init.txt` (that initializes the Data Base) and `insert-values.txt` that populates the Data Base. Implementing it by this way made everything more easy to develop because if it were errors on the Data Base, it was very fast to drop it, create it and populate it again.

Listing 5.8 shows the init script to create the database.

```

/*USER TABLE */
2 CREATE TABLE User(id int NOT NULL AUTO_INCREMENT, role int NOT NULL, username varchar(100)
    NOT NULL UNIQUE, email varchar(100) NOT NULL UNIQUE, pass varchar(100) NOT NULL, PRIMARY
    KEY(id));
/*STATIONS TABLE*/
4 CREATE TABLE Station(id int AUTO_INCREMENT, name varchar(100) UNIQUE, location varchar(100),
    ip varchar(50) UNIQUE, PRIMARY KEY(id));
/*CREATE TABLE Station(id int AUTO_INCREMENT, userStations_id int, name varchar(100) UNIQUE,
    location varchar(100), ip varchar(50) UNIQUE, PRIMARY KEY(id), FOREIGN KEY(
    userStations_id) REFERENCES UserStations(id) ON DELETE CASCADE);*/
6 /*USER STATIONS TABLE*/
    CREATE TABLE UserStations(id int AUTO_INCREMENT, user_id int, station_id int, PRIMARY KEY(id)
        , FOREIGN KEY(user_id) REFERENCES User(id) ON DELETE CASCADE, FOREIGN KEY(station_id)
        REFERENCES Station(id) ON DELETE CASCADE);
8 /*CREATE TABLE UserStations(id int AUTO_INCREMENT, user_id int, PRIMARY KEY(id), FOREIGN KEY(
    user_id) REFERENCES User(id) ON DELETE CASCADE);*/
/*FRAGRANCE LIST TABLE*/
10 CREATE TABLE FragranceList(id int AUTO_INCREMENT, station_id int UNIQUE, PRIMARY KEY(id),
    FOREIGN KEY(station_id) REFERENCES Station(id) ON DELETE CASCADE);
/*FRAGRANCE TABLE*/
12 CREATE TABLE Fragrance(id int AUTO_INCREMENT, fragranceList_id int, name varchar(50) NOT NULL
    , intensity varchar(20), vol_ml_max int, vol_ml_level int, description varchar(100),
    PRIMARY KEY(id), FOREIGN KEY(fragranceList_id) REFERENCES FragranceList(id) ON DELETE

```

### 5.3. Remote Server

```
CASCADE);
/*TIME TABLE TABLE*/
14 CREATE TABLE TimeTable(id int AUTO_INCREMENT, week int, station_id int, PRIMARY KEY(id),
    FOREIGN KEY(station_id) REFERENCES Station(id) ON DELETE CASCADE);
/*TIME SLOT TABLE*/
16 CREATE TABLE TimeSlot(id int, timeTable_id int, duration int, cost int, PRIMARY KEY(id),
    FOREIGN KEY(timeTable_id) REFERENCES TimeTable(id) ON DELETE CASCADE);
/*FILTER TABLE*/
18 create table Filter(id int AUTO_INCREMENT, name varchar(20), PRIMARY KEY(id));
/*AD TABLE*/
20 CREATE TABLE Ad(id int AUTO_INCREMENT, fragrance_id int, user_id int, timeSlot_id int,
    station_id int, filter_id int, active boolean, PRIMARY KEY(id), FOREIGN KEY(user_id)
    REFERENCES User(id) ON DELETE CASCADE, FOREIGN KEY(fragrance_id) REFERENCES Fragrance(id)
    ON DELETE CASCADE, FOREIGN KEY(timeSlot_id) REFERENCES TimeSlot(id) ON DELETE CASCADE,
    FOREIGN KEY(station_id) REFERENCES Station(id) ON DELETE CASCADE, FOREIGN KEY(filter_id)
    REFERENCES Filter(id) ON DELETE CASCADE);
/*MEDIAFILE TABLE*/
22 create table MediaFile(id int AUTO_INCREMENT, ad_id int, filename varchar(100), filesize
    varchar(100), filetype varchar(100), mdata blob, description varchar(200), PRIMARY KEY(id)
    ), FOREIGN KEY (ad_id) REFERENCES Ad(id) ON DELETE CASCADE);

24 /* ----- TRIGGER TO NOT INSERT DUPLICATED ROWS -----*/
delimiter @
26 CREATE TRIGGER avoid_duplication
    BEFORE INSERT ON UserStations FOR EACH ROW
28 BEGIN
    IF EXISTS (SELECT * FROM UserStations WHERE (NEW.user_id = user_id and NEW.station_id =
        station_id)) THEN
    30     SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'That user is already added to the station';
    32 END IF;
    33 END @
34 delimiter ;
/* ----- */
```

Listing 5.8: Script to create Data Base

Listing 5.9 shows the init script to create the database.

```
1 /*USER TABLE */
CREATE TABLE User(id int NOT NULL AUTO_INCREMENT, role int NOT NULL, username varchar(100)
    NOT NULL UNIQUE, email varchar(100) NOT NULL UNIQUE, pass varchar(100) NOT NULL, PRIMARY
    KEY(id));
```

### 5.3. Remote Server

---

```
3 /*STATIONS TABLE*/
4 CREATE TABLE Station(id int AUTO_INCREMENT, name varchar(100) UNIQUE, location varchar(100),
5 ip varchar(50) UNIQUE, PRIMARY KEY(id));
6 /*CREATE TABLE Station(id int AUTO_INCREMENT, userStations_id int, name varchar(100) UNIQUE,
7 location varchar(100), ip varchar(50) UNIQUE, PRIMARY KEY(id), FOREIGN KEY(
8 userStations_id) REFERENCES UserStations(id) ON DELETE CASCADE);*/
9 /*USER STATIONS TABLE*/
10 CREATE TABLE UserStations(id int AUTO_INCREMENT, user_id int, station_id int, PRIMARY KEY(id),
11 , FOREIGN KEY(user_id) REFERENCES User(id) ON DELETE CASCADE, FOREIGN KEY(station_id)
12 REFERENCES Station(id) ON DELETE CASCADE);
13 /*CREATE TABLE UserStations(id int AUTO_INCREMENT, user_id int, PRIMARY KEY(id), FOREIGN KEY(
14 user_id) REFERENCES User(id) ON DELETE CASCADE);*/
15 /*FRAGRANCE LIST TABLE*/
16 CREATE TABLE FragranceList(id int AUTO_INCREMENT, station_id int UNIQUE, PRIMARY KEY(id),
17 FOREIGN KEY(station_id) REFERENCES Station(id) ON DELETE CASCADE);
18 /*FRAGRANCE TABLE*/
19 CREATE TABLE Fragrance(id int AUTO_INCREMENT, fragranceList_id int, name varchar(50) NOT NULL,
20 , intensity varchar(20), vol_ml_max int, vol_ml_level int, description varchar(100),
21 PRIMARY KEY(id), FOREIGN KEY(fragranceList_id) REFERENCES FragranceList(id) ON DELETE
22 CASCADE);
23 /*TIME TABLE TABLE*/
24 CREATE TABLE TimeTable(id int AUTO_INCREMENT, week int, station_id int, PRIMARY KEY(id),
25 FOREIGN KEY(station_id) REFERENCES Station(id) ON DELETE CASCADE);
26 /*TIME SLOT TABLE*/
27 CREATE TABLE TimeSlot(id int, timeTable_id int, duration int, cost int, PRIMARY KEY(id),
28 FOREIGN KEY(timeTable_id) REFERENCES TimeTable(id) ON DELETE CASCADE);
29 /*FILTER TABLE*/
30 create table Filter(id int AUTO_INCREMENT, name varchar(20), PRIMARY KEY(id));
31 /*AD TABLE*/
32 CREATE TABLE Ad(id int AUTO_INCREMENT, fragrance_id int, user_id int, timeSlot_id int,
33 station_id int, filter_id int, active boolean, PRIMARY KEY(id), FOREIGN KEY(user_id)
34 REFERENCES User(id) ON DELETE CASCADE, FOREIGN KEY(fragrance_id) REFERENCES Fragrance(id)
35 ON DELETE CASCADE, FOREIGN KEY(timeSlot_id) REFERENCES TimeSlot(id) ON DELETE CASCADE,
36 FOREIGN KEY(station_id) REFERENCES Station(id) ON DELETE CASCADE, FOREIGN KEY(filter_id)
37 REFERENCES Filter(id) ON DELETE CASCADE);
38 /*MEDIAFILE TABLE*/
39 create table MediaFile(id int AUTO_INCREMENT, ad_id int, filename varchar(100), filesize
40 varchar(100), filetype varchar(100), mdata blob, description varchar(200), PRIMARY KEY(id)
41 ), FOREIGN KEY(ad_id) REFERENCES Ad(id) ON DELETE CASCADE);
42
43 /* ----- TRIGGER TO NOT INSERT DUPLICATED ROWS -----*/
44 delimiter @@
45 CREATE TRIGGER avoid_duplication
```

```

27 BEFORE INSERT ON UserStations FOR EACH ROW
28 BEGIN
29 IF EXISTS (SELECT * FROM UserStations WHERE (NEW.user_id = user_id and NEW.station_id =
30     station_id)) THEN
31     SIGNAL SQLSTATE '45000'
32     SET MESSAGE_TEXT = 'That user is already added to the station';
33 END IF;
34 END @
35 delimiter ;
36 /* ----- */

```

Listing 5.9: Script to populate Data Base

## 5.3.2. Threads

Looking back to Listing 5.1, the threads that are being used are:

- **server\_thr** – used to receive connections from other systems;
- **receive\_from\_ls\_thr** – used to receive messages from other systems;
- **send\_to\_ls\_thr** – used to send data to local system;
- **update\_local\_system\_thr** – updates the local system periodically.

### **server\_thr**

In listing 5.10 it is possible to take a look to the thread that waits for the connection of the local system.

```

1 void* MainWindow::server_thr(void* arg) {
2     MainWindow *mw = (MainWindow *)arg;
3
4
5     int socket_desc , client_sock , c , read_size;
6     struct sockaddr_in server , client;
7     char client_message[2000];
8
9     // Create socket
10    socket_desc = socket(AF_INET , SOCK_STREAM , 0);
11    if (socket_desc == -1)
12    {
13        std::cout << "Could not create socket" << std::endl;
14        return 0;
15    }
16
17    bind(socket_desc,(struct sockaddr *)&server , sizeof(server));
18    listen(socket_desc , 1);
19
20    c = accept(socket_desc,(struct sockaddr *)&client , &read_size);
21
22    if (c > 0)
23    {
24        cout << "Connection accepted" << endl;
25
26        read(client_message , read_size);
27
28        write(c,"Hello Client" , strlen("Hello Client"));
29
30        close(c);
31    }
32
33    else
34    {
35        cout << "Connection not accepted" << endl;
36    }
37
38    return 0;
39}

```

### 5.3. Remote Server

---

```
15     }
16     std::cout << "Socket created" << std::endl;
17
18     // Prepare the sockaddr_in structure
19     server.sin_family = AF_INET;
20     server.sin_addr.s_addr = INADDR_ANY;
21     server.sin_port = htons( 8888);
22
23     // Bind
24     if( bind(socket_desc,( struct sockaddr *)&server , sizeof(server)) < 0)
25     {
26         // print the error message
27         std::cout << "bind failed. Error" << std::endl;
28         return 0;
29     }
30     std::cout << "bind done" << std::endl;
31
32     // Listen
33     listen(socket_desc , 3);
34
35     //Accept and incoming connection
36     std::cout<< "Waiting for incoming connections..." << std::endl;
37     c = sizeof(struct sockaddr_in);
38
39     while(1) {
40         // accept connection from an incoming client
41         client_sock = accept(socket_desc , (struct sockaddr *)&client , (socklen_t*)&c);
42         if (client_sock < 0)
43         {
44             std::cout << "accept failed" << std::endl;
45             return 0;
46         }
47         std::cout << "Connection accepted" << std::endl;
48         mw->connectionEnable(true);
49         mw->sock = client_sock;
50     }
51 }
```

Listing 5.10: Implementation of Server Thread

This function creates a socket with a default port and starts the binding. After a success on binding, the thread blocks on `listen` to listen for an incoming connection. As soon as it comes an incoming connection,

the thread accepts it and runs the functions connectionEnable and stores the socket in order to keep the communication and signalize other functions that they can start sharing content with the local system.

### **receive\_from\_ls\_thr**

In listing 5.11 is the implementation of the thread that receives messages from the local system.

```

1 void * MainWindow::receive_from_ls_thr(void *arg) {
2     MainWindow *mw = (MainWindow *) arg;
3     char buffer[256];
4
5     /**< Wait for the connection */
6     while (!mw->isConnected());
7     /**< When connected, wait to receive something from the Local System*/
8     while (1) {
9         recv(mw->sock, buffer, sizeof(buffer), 0);
10        std::cout << "Received: " << buffer << std::endl;
11    }
12
13    return 0;
14 }
```

Listing 5.11: Implementation of Receive from MDO-L Thread

The function wait for the server to be connected to the local system, once that connection is established it blocks on the `recv`, waiting to receive something from the previously stored socket.

### **send\_to\_ls\_thr**

Listing 5.12 shows how the thread to send data to the local system works

```

1 void * MainWindow::send_to_ls_thr(void *arg) {
2     MainWindow *mw = (MainWindow *) arg;
3
4     std::cout << "Entrou em Enviado" << std::endl;
5     while (!mw->isConnected());
6     while (1) {
7         /**< When connected, wait for the conditional signal to be set*/
8         pthread_mutex_lock(&mw->_m_send_to_ls);
9         pthread_cond_wait(&mw->_send_to_ls, &mw->_m_send_to_ls);
10        pthread_mutex_unlock(&mw->_m_send_to_ls);
```

```

13     /**< Construct the string to send and send it to the Local System*/
14     std::string buff = mw->ad_to_send.fname + "," + mw->ad_to_send.link + "," + mw->
15         ad_to_send.frag_id + "," + mw->ad_to_send.filter_id + "," + mw->ad_to_send.
16         timeSlot_id + "," + mw->ad_to_send.enabled;
17     int buf_size = buff.size();
18
19     std::stringstream ss;
20     ss << buf_size;
21     std::string buf_size_str = ss.str();
22     std::string buffer = "A," + buf_size_str + "," + buff;
23     send(mw->sock , buffer.c_str() , buffer.size() , 0);
24     std::cout << "Enviado!!" << std::endl;
25 }/* close the client's channel */
26 return 0;
}

```

Listing 5.12: Implementation of Send to MDO-L Thread

Firstly, as in the previous thread, it waits until connection is established. After that, it waits for a conditional signal that is sent when a timer expires. When that signal triggers to HIGH, it creates a frame to send that will be interpreted by the local system.

#### **update\_local\_system\_thr**

This function waits for a conditional signal to continue executing. This conditional signal is set when a timer expires, after that it jumps to the function `upload_and_update` that uploads the file with `curlpp` and then signalizes the previous function to execute.

```

1 /**
2  * @brief MainWindow::update_local_system_thr
3  * @param arg
4  * @return
5  *
6  * Thread do handle the update of the local System.
7  * The thread waits for the condition variable to be set, as soon as
8  * it occurs, it calls the function that uploads the file to the internet
9  * and send the updates to the local system
10 */
11 void* MainWindow::update_local_system_thr(void *arg) {
12     MainWindow *mw = (MainWindow *)arg;

```

```

14     while(1) {
15         pthread_mutex_lock( &mw->_m_update_local_system );
16         pthread_cond_wait( &mw->_cond_update_local_system , &mw->_m_update_local_system );
17         pthread_mutex_unlock( &mw->_m_update_local_system );
18
19         mw-> upload_and_update(mw);
20
21     }
22
23
24     return nullptr;
}

```

Listing 5.13: Implementation of Update MDO-L Thread

Obviously, this was the fastest way used to implement a Remote Server, encapsulating it in the Remote Client. In the future, all these threads will leave this app and will migrate and be adjusted to work in a more expandable and robust way.

## 5.4. Local System

In this section, the local system implementation is discussed, walking through its most fundamental aspects.

### 5.4.1. Buildroot configuration

Buildroot is a tool to build custom tailored embedded Linux systems through cross-compilation, easing the creation and deployment of such systems to be loaded in the target system. The base configuration resides in the Buildroot installation path. As a first step the default configuration for a specific target must be generated running `make <TARGET_ARCH_defconfig>`. Then, `make menuconfig` can be executed to provide a GUI to interface the configurations of the custom embedded Linux image.

The fundamental configurations for the project are:

- system configuration: Enabling root login with password; Run a login prompt after boot;
- filesystem images: selection of file format and size for the image;
- Toolchain: setup of the development toolchain for the target, defining the C library and thread library debugging.

- Network: setup of the network utilities such as `dropbear` and `dhcpd` to ease the TCP/IP communication with the target.
- Packages: setup of the relevant packages for the project, namely, `opencv4` for image acquisition and processing, `Qt` for UI, `libcurl` to download ads, and `libcamera` to interface the CSI camera of the Raspberry Pi.

After setting the Buildroot configurations, the target image is generated using `make` and, if successful, it can be loaded into the target platform.

Fig. ?? through Fig. 5.9 presents the Buildroot setup of some features using the `make menuconfig` utility.

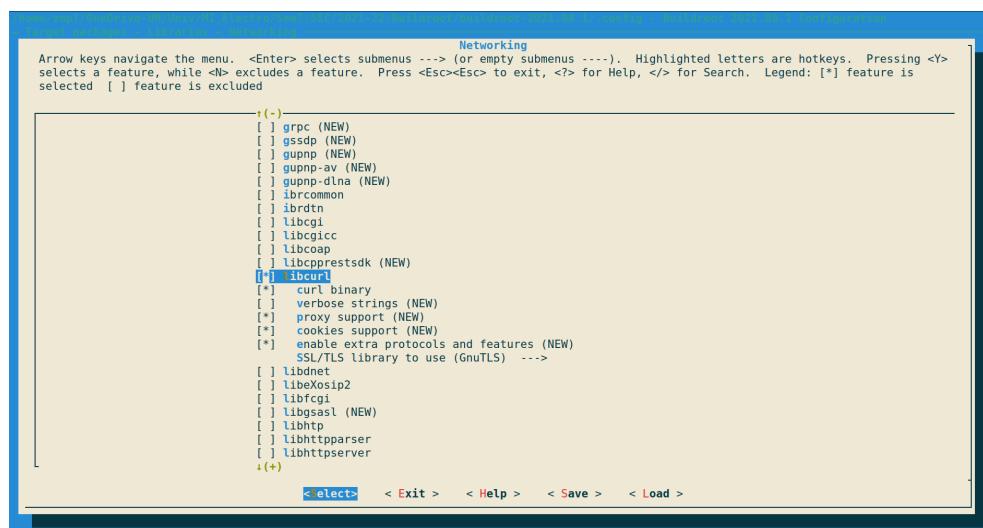


Figure 5.6.: Buildroot setup: libcurl

### 5.4.2. Automated build system for Local system software

To build the Local System software an automated build tool was used, namely `CMake`. `CMake` is a cross-platform tool to control the software compilation process using platform and compiler independent configuration files, and generating native `makefiles` to be executed, building the required targets. For this purpose, a `CMakeLists.txt` configuration file must be created in the software's path, as illustrated in Listing 5.14.

```

cmake_minimum_required(VERSION 3.5)

project(LSApp VERSION 0.1 LANGUAGES CXX)

set(CMAKE_INCLUDE_CURRENT_DIR ON)

```

## 5.4. Local System

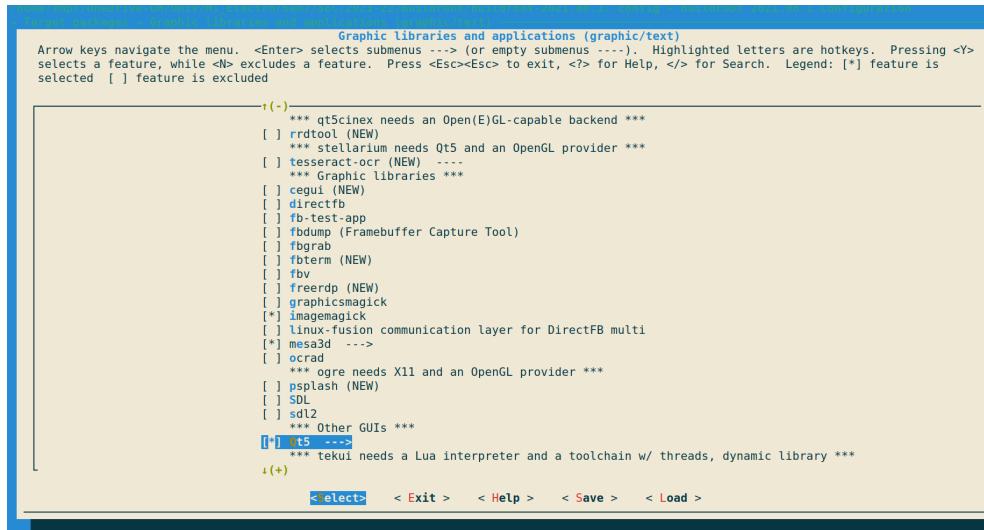


Figure 5.7.: Buildroot setup: Qt5

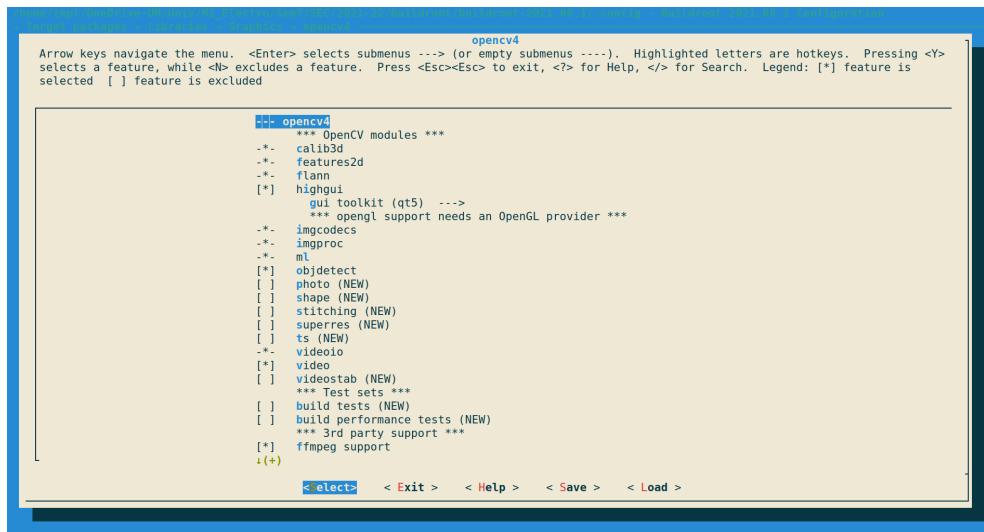


Figure 5.8.: Buildroot setup: Opencv4

```

set(CMAKE_AUTOUIC ON)
8 set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 11)
12 set(CMAKE_CXX_STANDARD_REQUIRED ON)

14 # ===== OPENCV
15 set(OpenCV_DIR "/usr/local/lib/cmake/opencv4")
16 find_package(OpenCV REQUIRED)

```

## 5.4. Local System

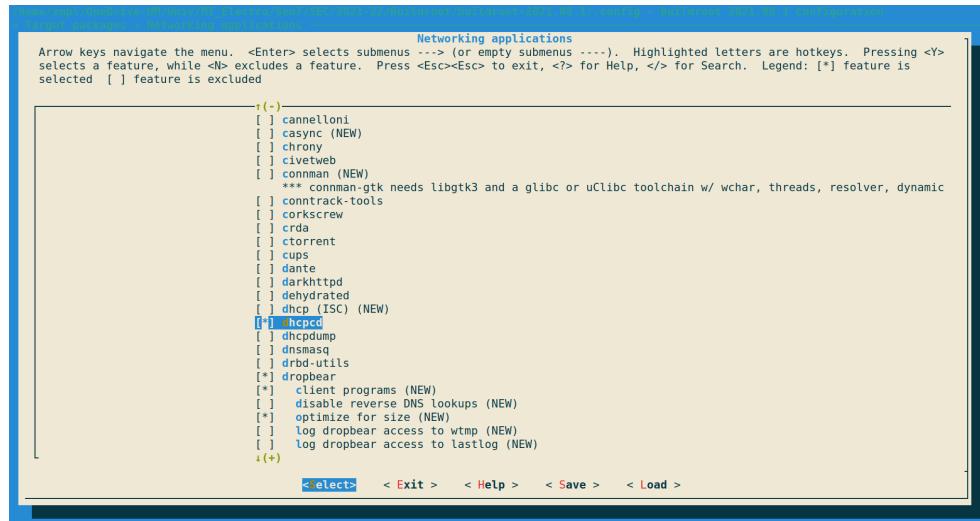


Figure 5.9.: Buildroot setup: Networking

```
find_package(OpenCV REQUIRED PATHS "/usr/local/lib/")
18 include_directories(${OpenCV_INCLUDE_DIRS})

20 # ====== Imagemagick
  find_program(MAGICK_CONFIG "Magick++-config")
22 execute_process(COMMAND "${MAGICK_CONFIG}" "--libs" OUTPUT_VARIABLE MAGICK_LD_FLAGS)
  # Remove trailing whitespace (CMAKE warns about this)
24 string(STRIPE ${MAGICK_LD_FLAGS} MAGICK_LD_FLAGS)
  add_definitions(-DMAGICKCORE_QUANTUM_DEPTH=16)
26 add_definitions(-DMAGICKCORE_HDRI_ENABLE=0)
  find_package(ImageMagick COMPONENTS Magick++)
28 include_directories(${ImageMagick_INCLUDE_DIRS})

30 # ====== Qt
  set(CMAKE_PREFIX_PATH "/home/zmpl/Qt/5.13.2/gcc_64/lib/cmake")

  find_package(Qt NAMES Qt6 Qt5 COMPONENTS Widgets VirtualKeyboard Multimedia MultimediaWidgets REQUIRED)
34 find_package(Qt${QT_VERSION_MAJOR} COMPONENTS Widgets VirtualKeyboard Multimedia MultimediaWidgets REQUIRED)

36 # ====== TWITCURL
  set(TWITCURL_LIB twitcurl)

  # SRC files
40 file(GLOB RESOURCE_FILES *.qrc)
```

```
file(GLOB SRC_FILES *.cpp)
42 file(GLOB HEADER_FILES *.h)

44 # Message Queue flags
set(MQUEUE_LD_FLAGS -lrt)

# set project sources
48 set(PROJECT_SOURCES
${HEADER_FILES}
50 ${SRC_FILES}
${RESOURCES_FILES}
52 )

54 # Add executable
add_executable(${PROJECT_NAME}
56 ${PROJECT_SOURCES}
)

# Link to libraries
60 target_link_libraries(${PROJECT_NAME} PRIVATE
${OpenCV_LIBS}
62 Qt${QT_VERSION_MAJOR}::Widgets Qt${QT_VERSION_MAJOR}::VirtualKeyboard Qt${QT_VERSION_MAJOR}
    ::Multimedia Qt${QT_VERSION_MAJOR}::MultimediaWidgets pthread ${TWITCURL_LIB} ${
MQUEUE_LD_FLAGS} ${MAGICK_LD_FLAGS} ${ImageMagick_LIBRARIES})
```

Listing 5.14: CMakeLists.txt file: main build configuration file

Additionally, CMake provides better cross-compilation capabilities, by enabling the programmer to specify the path to a toolchain configuration file, keeping the original file general and delegating the specific compilation tools for this file, as illustrated in Listing 5.15.

```
SET(CMAKE_SYSTEM_NAME Linux)

set(HOME /home/zmpl)
4 set(BUILDROOT_DIR "${HOME}/OneDrive -UM/Univ/MI_Electro/Sem7/SEC/2021-22/Buildroot/buildroot
    -2021.08.1")
set(HOST_PATH ${BUILDROOT_DIR}/output/host)
6 set(TOOLCHAIN_PATH ${HOST_PATH}/bin)

8 SET(TOOLS_PREFIX aarch64-buildroot-linux-gnu)

10 # Define the cross compiler locations
```

```

SET(CMAKE_C_COMPILER ${TOOLCHAIN_PATH}/${TOOLS_PREFIX}-gcc)
12 SET(CMAKE_CXX_COMPILER ${TOOLCHAIN_PATH}/${TOOLS_PREFIX}-g++)

14 # Define the sysroot path for the RaspberryPi distribution in our tools folder
SET(CMAKE_FIND_ROOT_PATH ${HOST_PATH}/${TOOLS_PREFIX}/sysroot)

SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
18 SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

```

Listing 5.15: raspToolchain.cmake file: toolchain setup file

### 5.4.3. System initialization

Embedded systems are typically self-contained and must be started without human intervention, thus requiring an initialization script. The `inittab` file contains the initialization scripts. In this file a line is added to redirect to the initialization script of the **Local system**. The process is relatively straightforward: adding modules to the **Linux** kernel, setting up IPC communication mechanisms like messages queues, inserting the device driver modules into the **Linux** kernel, starting daemons, running initialization commands (e.g., rotating the screen view), and finally launching the main application.

```

1#!/bin/sh

# add Video for Linux modules to kernel
modprobe bcm2835-v4l2

6# Creating and installing message queues
mkdir /dev/mqueue
mount -t mqueue distance /dev/mqueue

# Inserting device drivers kernel modules into Linux Kernel
11insmod /etc/LS/frag.ko
insmod /etc/LS/uss.ko

# starting the daemon associated with the Ultrasonic sensor
/etc/LS/uss_daemon.elf

# Rotating screen view
xrandr -o right

```

```

# Starting main process
21 /etc/LS/LSApp

```

Listing 5.16: Initialization script

#### 5.4.4. Device drivers

Two device drivers were developed/adapted for fragrance diffusion and user detection using two ultrasonic sensors. The device driver for user detection is instantiated twice (in different HW pins obviously), and then a daemon is responsible to read periodically from these device drivers and assert if an user was detected by counting these events in a sliding time window.

##### Fragrance diffusion

The fragrance diffusion device driver must write to a specific pin to enable/disable the device. The pin is statically defined in the module, thus requiring recompilation for several diffuser actuators. In the future, this should be changed to provide a more versatile interface. The module contains all the typical functions associated with `static struct file_operations`, namely `read`, `write`, `close` and `open`, although the first is not required. The most important function is the `write` one, which, upon writing to the associated file descriptor, sets the associated pin state to 0 or 1. An excerpt of this device is illustrated in Listing 5.17.

```

1 #include <linux/cdev.h>
2 #include <linux/module.h>
3 #include <linux/fs.h>
4 #include <linux/device.h>
5 #include <linux/init.h>
6 #include <linux/module.h>
7 #include <asm/io.h>
8 #include <linux/timer.h>
9 #include <linux/device.h>
10 #include <linux/err.h>
11 #include <linux/mm.h>
12 #include <linux/io.h>

13 #include "utils.h"

14 #define DEVICE_NAME "frag0"
15 #define CLASS_NAME "fragClass"

```

```

19 MODULE_LICENSE("GPL");
20 MODULE_INFO(intree, "Y");

21 /* Device variables */
22 struct class* fragDevice_class = NULL;
23 static dev_t fragDevice_majorminor;
24 static struct cdev c_dev; // Character device structure

25 struct GpioRegisters *s_pGpioRegisters;
26 static const int fragGpioPin = 26;

27 ssize_t frag_device_write(struct file *pfile, const char __user *pbuff, size_t len, loff_t *
28                           off) {
29     struct GpioRegisters *pdev;
30
31     pr_alert("%s: called (%u)\n", __FUNCTION__, len);
32
33     if(unlikely(pfile->private_data == NULL))
34         return -EFAULT;
35
36     pdev = (struct GpioRegisters *)pfile->private_data;
37     if (pbuff[0]== '0')
38         SetGPIOOutputValue(pdev, fragGpioPin, 0);
39     else
40         SetGPIOOutputValue(pdev, fragGpioPin, 1);
41     return len;
42 }

43 static struct file_operations fragDevice_fops = {
44     .owner = THIS_MODULE,
45     .write = frag_device_write,
46     .read = frag_device_read,
47     .release = frag_device_close,
48     .open = frag_device_open,
49 };

```

Listing 5.17: Fragrance diffuser kernel module (excerpt)

## Ultrasonic sensor

A device driver was implemented for the ultrasonic sensor, which can be instantiated multiple times (in this case two), requiring the trigger and echo pins, and the associated timeout. This device driver operates by emitting a trigger signal and measuring the time between the trigger and echo signal to determine the time of flight, and consequently the distance. Thus, this device driver requires a specific signal timing as aforementioned in Section 3.12..

The device driver implemented was adapted from [114], being ported to a more updated version, and abstracting it. The most important functions are shown in Listing 5.18. Multiple device drivers can be instantiated or removed in different hardware pins on demand, providing a more flexible interface through the `sysfs_configure_store` function. Then, on the background a periodic task performs the measurement—`do_measurement`—and when the echo is triggered a signal an interruption is issued to handle this event, flushing the result to the associated file descriptor.

It should be noted that for the present use case both triggers are shared, as we expect to have ‘simultaneous’ readings to work with.

```

1  /* Precise measurements of time delta between sending a trigger signal
2   * to the HC-SR04 distance sensor and receiving the echo signal from
3   * the sensor back. This has to be precise in the usecs range. We
4   * use trigger interrupts to measure the signal, so no busy wait :)
5   *
6   * This supports an (in theory) unlimited number of HC-SR04 devices.
7   * To add a device, do a (as root):
8   *
9   * # echo 23 24 1000 > /sys/class/distance-sensor/configure
10  *
11  * (23 is the trigger GPIO, 24 is the echo GPIO and 1000 is a timeout in
12  * milliseconds)
13  *
14  * Then a directory appears with a file measure in it. To measure, do a
15  *
16  * # cat /sys/class/distance-sensor/distance_23_24/measure
17  *
18  * You'll receive the length of the echo signal in usecs. To convert (roughly)
19  * to centimeters multiply by 17150 and divide by 1e6.
20  *
21  * To deconfigure the device, do a
22  *
23  * # echo -23 24 > /sys/class/distance-sensor/configure
24  *
25  * (normally not needed).

```

```
*  
27 * DO NOT attach your HC-SR04's echo pin directly to the raspberry, since  
28 * it runs with 5V while raspberry expects 3V on the GPIO inputs.  
29 *  
30 */  
  
31 #include <linux/kernel.h>  
32 #include <linux/module.h>  
33 #include <linux/timekeeping.h>  
34 #include <linux/gpio/consumer.h>  
35 #include <linux/mutex.h>  
36 #include <linux/device.h>  
37 #include <linux/sysfs.h>  
38 #include <linux/interrupt.h>  
39 #include <linux/kdev_t.h>  
40 #include <linux/list.h>  
41 #include <linux/slab.h>  
42 #include <linux/gpio/driver.h>  
43 #include <linux/delay.h>  
44 #include <linux/sched.h>  
45 #include <linux/time.h>  
46 #include <linux/ktime.h>  
  
47 struct hc_sro4 {  
48     int gpio_trig;  
49     int gpio_echo;  
50     struct gpio_desc *trig_desc;  
51     struct gpio_desc *echo_desc;  
52     // struct old_timeval32 time_triggered;  
53     // struct old_timeval32 time_echoed;  
54     struct timespec64 time_triggered;  
55     struct timespec64 time_echoed;  
56     int echo_received;  
57     int device_triggered;  
58     struct mutex measurement_mutex;  
59     wait_queue_head_t wait_for_echo;  
60     unsigned long timeout;  
61     struct list_head list;  
62 };  
  
63 static LIST_HEAD(hc_sro4_devices);  
64 static DEFINE_MUTEX(devices_mutex);
```

```

69 static struct hc_sro4 *create_hc_sro4(int trig, int echo, unsigned long timeout)
    /* must be called with devices_mutex held */
71 {
72     struct hc_sro4 *new;
73     int err;
74
75     new = kmalloc(sizeof(*new), GFP_KERNEL);
76     if (new == NULL)
77         return ERR_PTR(-ENOMEM);
78
79     new->gpio_echo = echo;
80     new->gpio_trig = trig;
81     new->echo_desc = gpio_to_desc(echo);
82     if (new->echo_desc == NULL) {
83         kfree(new);
84         return ERR_PTR(-EINVAL);
85     }
86     new->trig_desc = gpio_to_desc(trig);
87     if (new->trig_desc == NULL) {
88         kfree(new);
89         return ERR_PTR(-EINVAL);
90     }
91
92     err = gpiod_direction_input(new->echo_desc);
93     if (err < 0) {
94         kfree(new);
95         return ERR_PTR(err);
96     }
97     err = gpiod_direction_output(new->trig_desc, 0);
98     if (err < 0) {
99         kfree(new);
100        return ERR_PTR(err);
101    }
102    gpiod_set_value(new->trig_desc, 0);
103
104    mutex_init(&new->measurement_mutex);
105    init_waitqueue_head(&new->wait_for_echo);
106    new->timeout = timeout;
107
108    list_add_tail(&new->list, &hc_sro4_devices);
109
110    return new;
111 }

```

```

113 static irqreturn_t echo_received_irq(int irq, void *data)
114 {
115     struct hc_sro4 *device = (struct hc_sro4 *) data;
116     int val;
117     // struct old_timeval32 irq_tv;
118     struct timespec64 irq_tv;
119     // do_gettimeofday32(&irq_tv);
120     ktime_get_ts64(&irq_tv);

122     if (!device->device_triggered)
123         return IRQ_HANDLED;
124     if (device->echo_received)
125         return IRQ_HANDLED;

127     val = gpiod_get_value(device->echo_desc);
128     if (val == 1) {
129         device->time_triggered = irq_tv;
130     } else {
131         device->time_echoed = irq_tv;
132         device->echo_received = 1;
133         wake_up_interruptible(&device->wait_for_echo);
134     }

136     return IRQ_HANDLED;
137 }

139 static int do_measurement(struct hc_sro4 *device,
140                           unsigned long long *usecs_elapsed)
141 {
142     long timeout;
143     int irq;
144     int ret;

146     if (!mutex_trylock(&device->measurement_mutex)) {
147         mutex_unlock(&devices_mutex);
148         return -EBUSY;
149     }
150     mutex_unlock(&devices_mutex);

152     msleep(60);
153     /* wait 60 ms between measurements.
154      * now, a while true ; do cat measure ; done should work

```

```

155     */
156
157     irq = gpiod_to_irq(device->echo_desc);
158     if (irq < 0)
159         return -EIO;
160
161     device->echo_received = 0;
162     device->device_triggered = 0;
163
164     ret = request_any_context_irq(irq, echo_received_irq,
165         IRQF_SHARED | IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
166         "hc_sro4", device);
167
168     if (ret < 0)
169         goto out_mutex;
170
171     gpiod_set_value(device->trig_desc, 1);
172     udelay(10);
173     device->device_triggered = 1;
174     gpiod_set_value(device->trig_desc, 0);
175
176     ret = gpiochip_lock_as_irq(gpiod_to_chip(device->echo_desc),
177         device->gpio_echo);
178     if (ret < 0)
179         goto out_irq;
180
181     timeout = wait_event_interruptible_timeout(device->wait_for_echo,
182         device->echo_received, device->timeout);
183
184     if (timeout == 0)
185         ret = -ETIMEDOUT;
186     else if (timeout < 0)
187         ret = timeout;
188     else {
189         *usecs_elapsed =
190             (device->time_echoed.tv_sec - device->time_triggered.tv_sec) * 1000000 +
191             (device->time_echoed.tv_nsec - device->time_triggered.tv_nsec) / 1000;
192         ret = 0;
193     }
194     /* TODO: unlock_as_irq */
195 out_irq:
196     free_irq(irq, device);
197 out_mutex:

```

```

        mutex_unlock(&device->measurement_mutex);

    return ret;
201 }

203 static struct class hc_sro4_class = {
204     .name = "distance-sensor",
205     .owner = THIS_MODULE,
206     .class_groups = hc_sro4_class_groups,

207     static ssize_t sysfs_configure_store(struct class *class,
208                                         struct class_attribute *attr,
209                                         const char *buf, size_t len)
210 {
211     int add = buf[0] != '-';
212     const char *s = buf;
213     int trig, echo, timeout;
214     struct hc_sro4 *new_sensor, *rip_sensor;
215     int err;

216     if (buf[0] == '-' || buf[0] == '+')
217         s++;

218     if (add) {
219         if (sscanf(s, "%d %d %d", &trig, &echo, &timeout) != 3)
220             return -EINVAL;

221         mutex_lock(&devices_mutex);
222         if (find_sensor(trig, echo)) {
223             mutex_unlock(&devices_mutex);
224             return -EEXIST;
225         }
226     }

227     new_sensor = create_hc_sro4(trig, echo, timeout);
228     mutex_unlock(&devices_mutex);
229     if (IS_ERR(new_sensor))
230         return PTR_ERR(new_sensor);

231     device_create_with_groups(class, NULL, MKDEV(0, 0), new_sensor,
232                               sensor_groups, "distance-%d-%d", trig, echo);
233 } else {
234     if (sscanf(s, "%d %d", &trig, &echo) != 2)
235         return -EINVAL;

```

```

        mutex_lock(&devices_mutex);
243    rip_sensor = find_sensor(trig, echo);
    if (rip_sensor == NULL) {
245        mutex_unlock(&devices_mutex);
        return -ENODEV;
247    }
    err = remove_sensor(rip_sensor);
249    mutex_unlock(&devices_mutex);
    if (err < 0)
251        return err;
}
253 return len;
}

```

Listing 5.18: Ultrasonic sensor kernel module (excerpt) – adapted from [114]

## 5.4.5. Daemons

As aforementioned, a daemon was created to periodically read the ultrasonic sensors and count the detection events in a time sliding window, defining such event as both sensors must be enabled. The daemon follows the required initialization steps to detach itself from the outside world, as mentioned in Section 3.5. Next, the message queue associated with the daemon is destroyed if present and then (re)created. Then, it should read the sensors periodically, applying a sliding window and signaling the user detection event to a message queue shared with the main application. The main application should periodically consume this information to trigger the appropriate event handling. This daemon is illustrated in Listing 5.19.

```

1 #include "ultrassonicsensor.h"
2 #include <fcntl.h>
3 #include <iostream>
4 #include <mqueue.h>
5 #include <signal.h>
6 #include <sstream>
7 #include <stdio.h>
8 #include <stdlib.h>
9 #include <string.h>
10 #include <string>
11 #include <sys/stat.h>
12 #include <sys/syslog.h>

```

```
13 #include <sys/types.h>
14 #include <time.h>
15 #include <unistd.h>

17 using namespace DeviceDriver;

19 void signal_handler(int sig) {
20     switch (sig) {
21     case SIGHUP:
22         syslog(LOG_INFO, "Hangup signal catched");
23         break;
24     case SIGTERM:
25         syslog(LOG_INFO, "Terminate signal catched");
26         exit(0);
27         break;
28     }
29 }

31 #define MSGQ_PATH "/distance"
32 #define MAX_MSG_LEN 64

34 mqd_t create_mqueues() {
35     struct mq_attr attr;

36     attr.mq_flags = 0;
37     attr.mq_maxmsg = 1;
38     attr.mq_msgsize = MAX_MSG_LEN;
39     attr.mq_curmsgs = 0;

41     mqd_t msgq_path = mq_open(MSGQ_PATH, O_RDWR | O_CREAT | O_NONBLOCK,
42                               S_IRWXU | S_IRWXG, &attr);

43     if (msgq_path == (mqd_t)-1) {
44         std::cerr << "In mq_open()" << std::endl;
45         mq_close(msgq_path);
46         exit(1);
47     }
48 }

49 return msgq_path;
50 }

52 bool checkDistance(int dist) {
53 #define MIN_TH 2
```

```

#define MAX_TH 30
57   return (dist > MIN_TH && dist < MAX_TH);
}

int main(int argc, char *argv[]) {
61   pid_t pid, sid;
62   int len, fd;
63   time_t timebuf;

65   pid = fork(); // create a new process

67   if (pid < 0) { // on error exit
68     syslog(LOG_ERR, "%s\n", "fork");
69     exit(EXIT_FAILURE);
70   }

72   if (pid > 0) {
73     printf("Deamon PID: %d\n", pid);
74     exit(EXIT_SUCCESS); // parent process (exit)
75   }
76   sid = setsid(); // create a new session

78   if (sid < 0) { // on error exit
79     syslog(LOG_ERR, "%s\n", "setsid");
80     exit(EXIT_FAILURE);
81   }
82   // make '/' the root directory
83   if (chdir("/") < 0) { // on error exit
84     syslog(LOG_ERR, "%s\n", "chdir");
85     exit(EXIT_FAILURE);
86   }
87   umask(0);
88   close(STDIN_FILENO); // close standard input file descriptor
89   close(STDOUT_FILENO); // close standard output file descriptor
90   close(STDERR_FILENO); // close standard error file descriptor

92   signal(SIGHUP, signal_handler); /* catch hangup signal */
93   signal(SIGTERM, signal_handler); /* catch kill signal */

95 #define TIMEOUT_S 1
96 #define ITER 8
97 #define S1_ECHO 22
98 #define S2_ECHO 23

```

```

99 #define S_TRIG 17
100 #define S_TIMEOUT 1000
101 #define LOGFILE "/var/log/distance.log"
102     int _delay = 1000 / ITER;

103     /**< Instantiation sensors */
104     UltrasonicSensor s1(S_TRIG, S1_ECHO, S_TIMEOUT);
105     UltrasonicSensor s2(S_TRIG, S2_ECHO, S_TIMEOUT);
106     /**< Open them */
107     if (!s2.Open() || !s1.Open()) {
108         perror("open");
109         exit(EXIT_FAILURE);
110     }
111     /**< Unlink the message queue if it already created */
112     mq_unlink(MSGQ_PATH);
113     /**< Create msg queues */
114     mqd_t msgq_path = create_mqueues();
115     /**< Perform sliding window on both sensors readings */
116     while (1) {
117         int count = 0;
118         for (int j = 0; j < ITER * TIMEOUT_S; j++) {
119             count = count + (checkDistance(s1.Read()) & checkDistance(s2.Read()));
120             usleep(_delay * 1000);
121         }
122         char c = (count > (TIMEOUT_S * ITER / 2)) ? '1' : '0';
123         /**< Flush the result to the associated message queue */
124         mq_send(msgq_path, &c, 2, 1);
125     }
126     exit(EXIT_SUCCESS);
127 }
```

Listing 5.19: User detection daemon

## 5.4.6. Classes

In these sections the main classes for the Local System are discussed.

## UI

The UI classes handle the interaction between the user and the application and are defined according to the application mode, namely:

- **MainWindow**: main application's window. Manages all other windows (views), being the controller. The other views emit signals that are handled by these class to avoid circular dependencies and centralize the control.
- **NormalWindow**: handles the Normal Window view application logic.
- **InterWindow**: handles the Interaction Window view application logic.
- **ImgFilterWindow**: handles the Image Filtering view application logic.
- **SharWindow**: handles the Sharing view application logic.

In order to ease this model of control, the subordinate views are restricted to a small subset of the application window, i.e., the only part that actually requires replacing in each view. This means that the canvas to display the images and the status bar is preserved throughout the application and solely belongs to the **MainWindow**.

### NormalWindow

An example of a subordinate view is the **NormalWindow** which interface is presented in Listing 5.20. It can be seen that it contains a Qt signal to indicate when **Normal** view must be exited, and it was used to test the application logic. This Qt signal is then connected to the callback of a recipient object (a slot) to perform some action.

```

1 #ifndef NORMALWINDOW_H
2 #define NORMALWINDOW_H

4 #include <QWidget>
namespace Ui { class NormalWindow ;}

6 class NormalWindow : public QWidget
8 {
    Q_OBJECT
10 public:
    explicit NormalWindow(QWidget *parent = nullptr);
12 ~NormalWindow();
14 signals:
16     void home_pressed(); /*< Dummy signal to indicate return to main Window */
18 private slots:
20     void on_pushButton_clicked(); /*< Dummy button to go to Main Window */
22 private:

```

```
18     Ui :: NormalWindow * ui;  
19 };  
20 #endif // NORMALWINDOW_H
```

Listing 5.20: NormalWindow – an example of a subordinate view

## MainWindow

It is the view controller and the application logic manager, holding the definition for each of the modes. Listing 5.21 presents the interface for this class.

```
1 #ifndef MAINWINDOW_H  
2 #define MAINWINDOW_H  
3 #include < QMainWindow >  
4 #include "msgqueue.h"  
5 #include "normalwindow.h"  
6 #include "interwindow.h"  
7 #include "imgfiltwindow.h"  
8 #include "sharwindow.h"  
9 #include <QGraphicsScene >  
10 #include <QGraphicsPixmapItem >  
11 #include <QImage >  
12 #include <opencv2/objdetect.hpp>  
13 #include <qfiledialog.h>  
14 #include <qgraphicsitem.h>  
15 #include <qgraphicsscene.h>  
16 #include <QWidget.h>  
17 /**< OpenCV */  
18 #include <opencv2/opencv.hpp>  
19 /**< Pthreads */  
20 #include <pthread.h>  
21 #include "pEvent.h"  
22 /**< STL containers */  
23 #include <vector >  
24 #include <list >  
25 /**< Image filter */  
26 #include "imgfilter.h"  
27 /**< Twitter */  
28 #include "include/twcurl.h"  
29 /**< Post */  
30 #include "post.h"  
31 /**< Imagemagick (for GIF generation) */
```

```

32 #include <Magick++.h>
33 /*< VideoPlayer */
34 #include <QMediaPlayer>
35 #include <QMediaPlaylist>
36 /*< Ad */
37 #include "ad.h"
38 /*< Fragrance */
39 #include "frag.h"
40 #include "fragManager.h"
41 #include "fragDiffuser.h"
42 /*< Client Server Architecture */
43 /* Cant use Linux system call connect() due to naming clashes with
44 * Qt connect() function
45 */
46 #include <QTcpSocket>
47 /*< Message queue */
48 #include "msgqueue.h"

49 /**
50 * @brief App modes
51 *
52 * Used to handle application logic and UIs views
53 * - WELCOME = 0, Welcome screen; displayed when no ad is running
54 * - NORMAL, Reproduces video in fullscreen (with audio)
55 * - INTER, Interaction mode: main menu; picture mode; GIF mode
56 * - IMGFILT, Image filtering menu
57 * - SHAR, Sharing mode: main menu; editing post; status
58 * - QUIT Quit application
59 */
60 enum AppMode { WELCOME = 0, NORMAL, INTER, IMGFILT, SHAR, QUIT };
61 typedef enum AppMode AppMode_t;

62 QT_BEGIN_NAMESPACE
63 namespace Ui { class MainWindow; }
64 QT_END_NAMESPACE

65 class MainWindow : public QMainWindow
66 {
67     Q_OBJECT
68     public:
69         MainWindow(QWidget *parent = nullptr);
70         ~MainWindow();
71     private slots:
72 }
```

```

/* ----- DUMMY -----
76   void on_pushButton_clicked(); /*< Dummy button to go to Normal Mode */
    void on_pushButton_2_clicked(); /*< Dummy button to go to Interaction Mode */
78   void on_pushButton_3_clicked(); /*< Dummy button to go to ImgFilt Mode */
    void on_pushButton_4_clicked(); /*< Dummy button to go to Sharing Mode */
80   void onHome_pressed(); /*< slot to handle dummy signal to return to main window */

/* ----- END DUMMY -----
82   /**< Signals handlers */
83   void onNormalMode_pressed();
84   void onInter_mode_pressed();
85   void onShar_mode_pressed();
86   void onImgFilt_mode_pressed();
87   void onCam_started();
88   void onImgFiltSelected(int idx);
89   void onTwitterShare(const QString &);

90   void onTakePic_complete();
91   void onImgFiltGlobal(bool enable);
92   void onGifEnabled(bool enable);
93   void OnMediaStatusChanged(QMediaPlayer::MediaStatus status);
94   void onFragTimerElapsed();
95   void onCheckModeTimerElapsed();
96   void onRemoteConnectionStateChanged(QAbstractSocket::SocketState);
97   void onRemoteConnected();
98   void OnTcpDataAvail();

100  /**< Display image */
101  void displayImg(cv::Mat frame);
102  /**< ImgFilter */
103  void createFilters();
104  void detectFaces(cv::Mat *frame);
105  void transparentOv(cv::Mat *src, cv::Mat *dst, cv::Mat *overlay);
106  void applyFilterOverlay(cv::Mat &frame, ImgFilter &filt);

107  /**< Thread workers */
108  static void* frame_grabber_worker_thr(void* arg);
109  static void* gif_save_worker_thr(void* arg);
110  static void* rx_worker_thr(void* arg);
111  static void* process_worker_thr(void* arg);
112  static void* download_ad_worker_thr(void* arg);

113  /**< Twitter sharing */
114  bool TwitterAuthenticate();
115  /**< recognize gestures */
116  void Mat2Magick(cv::Mat& src, Magick::Image &mgk);
117  /**< Scene */

```

```

118     void updateScene(AppMode_t mode);
119     /**< VideoPlayer */
120     bool openVideo(const QString fname);
121     /**< Client-server */
122     void connectToRemote();
123     void pushTcpData(const QString &s);
124     void popTcpData(QString &s);
125     /**< Helpers */
126     void updateStatusBar(const QString str);
127     AppMode_t appMode();
128     void setAppMode(AppMode_t mode);
129     void filterEnable(bool enable);
130     bool filterEnabled();
131     void curAd(Ad &ad);
132     void setCurAd(Ad &ad);
133     void curFrag(Frag::Fragrance &f);
134     void setCurFrag(Frag::Fragrance &f);
135     bool detectUser();
136 /**
137 * @brief Compare rectangles by area (descending)
138 * @param a: first rectangle to compare
139 * @param b: second rectangle to compare
140 * @return comparison result: true if first rectangle area is bigger than second one,
141         otherwise false;
142 *
143 * detailed
144 */
145 inline static bool compareRects(const cv::Rect &a, const cv::Rect &b){
146     return a.area() - b.area();
147 }
148 signals:
149     void interWindUpdateStatus(const QString str);
150     void textChanged(QString);
151     void imgGrabbed(cv::Mat frame);
152 private:
153     Ui::MainWindow *ui = nullptr; /**< UI main view */
154     NormalWindow *_normalWind = nullptr; /**< Normal Window ptr */
155     InterWindow *_interWind = nullptr; /**< Normal Window ptr */
156     ImgFiltWindow *_imgFiltWind = nullptr; /**< Normal Window ptr */
157     SharWindow *_sharWind = nullptr; /**< Normal Window ptr */
158     QGraphicsPixmapItem *_pixmap = nullptr; /**< Holds the grabbed frames */
159     QGraphicsPixmapItem *_welcome_img = nullptr; /**< Welcome img for the UI */
160     cv::VideoCapture _video; /**< CV video object to handle video */

```

## 5.4. Local System

---

```
160     cv::CascadeClassifier _face_cascade; /*< Haar cascade to detect faces */
161     cv::CascadeClassifier _gesture_cascade; /*< Haar cascade to detect faces */
162     AppMode_t _appmode; /*< Stores app mode */

164     /*< Threads */
165     pthread_t _frame_grab_thr; /*< Frame Grabber thread */
166     pthread_t _gif_save_thr; /*< GIF save thread */
167     pthread_t _rx_thr; /*< Receive from Remote System thread */
168     pthread_t _process_rx_thr; /*< Process rx thread */
169     pthread_t _download_ad_thr; /*< Download Ad thread */

171     /*< Mutexes */
172     /* Normal */
173     pthread_mutex_t _m_status_bar; /*< Protects access to UI status bar */
174     pthread_mutex_t _m_mode; /*< Protects access to mode state variable */
175     pthread_mutex_t _m_curFrame; /*< Protects access to current frame */
176     pthread_mutex_t _m_imgFilter; /*< Protects access to img filter */
177     pthread_mutex_t _m_gif; /*< Protects access to GIF resources */
178     pthread_mutex_t _m_cur_ad; /*< Protects access to current Ad */
179     pthread_mutex_t _m_cur_frag; /*< Protects access to current Frag */
180     pthread_mutex_t _m_tcp_buff; /*< Protects access to the TCP buffer */

182     /*< Pthread events: Condition variables */
183     pEvent * _ev_gif_save; /*< Event GIF save */
184     pEvent * _ev_frame_grab; /*< Event Frame Grabber */
185     pEvent * _ev_diff; /*< Event Fragrance diffuser */
186     pEvent * _ev_rx; /*< Event Rx: data received from remote system */
187     pEvent * _ev_download; /*< Event Download: download a new Ad */
188     pEvent * _ev_process; /*< Event process: Process Rx data */

190     /*< Filters */
191     QString _filtName; /*< Filter name */
192     cv::Mat _filter;
193     std::vector<ImgFilter> _filters;
194     int _filters_idx;
195     bool _filter_on;

197     /*< Scenes */
198     QGraphicsScene *_welcome_scene = nullptr;
199     QGraphicsScene *_video_scene = nullptr;
200     QGraphicsScene *_inter_scene = nullptr;
201     /*< Image Acquisition */
202     cv::Mat _curFrame;
```

```

        bool _updateCanvas;
204    /**< GIF */
        bool _gif_on;
206    bool _gif_complete;
        std::vector<cv::Mat> frames;
208    std::list<Magick::Image> images;
    /**< Twitter obj */
210    twitCurl _twitterObj;
        bool _twitterAuthenticated;
212    /**< Post */
        Post _post;
    /**< VideoPlayer */
        QMediaPlayer *_mediaPlayer = nullptr;
216    QMediaPlaylist *_mediaPlaylist = nullptr;
        QGraphicsVideoItem *_videoItem = nullptr;
218    /**< Ad */
        Ad _curAd;
220    Frag::Fragrance _curFrag;
    /**< Fragrance */
        Frag::Manager *_fragMan = nullptr;
        Frag::Diffuser *_fragDiff = nullptr;
224    QTimer *_fragTimer = nullptr;
        bool _event_diff = false;
226    /**< Normal mode */
        QTimer *_checkModeTimer; /**< Check periodically if normal mode needs to run */

    /**< Client-Server: Local system is the client */
230    /* Can't use Linux system call connect() due to naming clashes with
     * Qt connect() function
232    */
        QTcpSocket *_remoteSock;
234    bool _remoteConnected;
        QStringList *_remoteDataBuff;
236    /**< Message Queue */
        msgQueue *_mq_user_detect = nullptr;
238};

#endif // MAINWINDOW_H

```

Listing 5.21: MainWindow – interface

It is composed of several objects, namely:

- subordinate views (windows);

- video capture
- threads
- mutexes: for synchronization
- pEvents: wrapper around condition variables; for synchronization
- image filters
- scenes: welcome, video, and interaction
- GIF
- Twitter
- Post: Twitter post
- Video Player
- Ad: current ad
- Fragrance: current fragrance, fragrance manager to determine fragrance settings from the database, a diffuser and the associated timer for actuation
- Timer: for periodic tasks
- Socket: to connect to remote system
- Message queue to interface the user detection sensors via daemon

It provides functions to:

- handle subordinate views signals and other relevant events like taking a picture, sharing a post, connection to the remote, receiving data from remote system, etc.
- Handle computer vision tasks: grab frames, display images, detect faces, recognize gestures to navigate the interface and apply filters.
- Thread workers: functions associated to each thread.
- Twitter authentication
- GIF creation
- Handling communication
- and several helpers

The threads and the associated workers will be detailed later.

### Ad

This class handles ad logic, namely (see Listing 5.22):

- storing relevant data
- saving to and restoring data from the database
- Retrieving important data like the filterID, the fragID, the timeslot, or the enabled state.
- Enabling the Ad, dispatching it for execution in the normal mode, obviously if there is no user interaction.

It is important to note that this class is thread safe as it includes a synchronization mechanism — a mutex — to prevent unattended access.

```

1 /**
 * @file ad.h
3 * @author Jose Pires
 * @date 2022-02-03
5 *
 * @brief Ad class
7 */
#ifndef _AD_H_
#define _AD_H_
#include <string>
#include <pthread.h>
class Ad{
private:
    std::string _fname; /*< filename */
    std::string _link; /*< download link from Proxy server */
    std::string _mediaPath; /*< download link from Proxy server */
    int _fragranceID; /*< fragrance ID */
    int _filterID; /*< Filter ID */
    int _timeslot; /*< integer defining the timeslot from the beginning of the week */
    bool _enabled; /*< asserts if the ad is currently enabled */
    pthread_mutex_t _mutex; /*< protect access to the enabled state */
    static const std::string ads_path_prefix;
    static const std::string media_path_prefix;
public:
    /*< Constructors/Destructors */
    Ad(std::string fname = "", std::string link = "",
        int fragID = 0, int filterID = 0, int timeslot = 0);
    ~Ad();
    /*< Getters */
    void fname(std::string &fname);
    void link(std::string &link);
    void mediaPath(std::string &link);
    int fragID();
    int filterID();
}

```

```

35     int timeslot();
36     bool enabled();
37     inline bool operator==(const Ad& rhs)
38     { return (this->_timeslot == rhs._timeslot); }
39     /**< Setters / Mutators */
40     void enable(const bool enabled);
41     bool save();
42     bool load(std::string fname);
43 };
#endif // _AD_H__H

```

Listing 5.22: Ad — interface

## DigitalOutput

This class, enclosed in the `Device::Driver` namespace, models a digital output device driver, as the fragrance diffuser. It provides methods to open, write and close the device driver, as illustrated in Listing 5.23.

```

1 /**
2  * @file ddDigitalOut.h
3  * @author Jose Pires
4  * @date 2022-02-04
5  *
6  * @brief Device Driver for Digital Output
7  */
8 #pragma once
9 #include <string>
10 namespace DeviceDriver{
11     class DigitalOutput{
12     private:
13         static const std::string devPath;
14         static const std::string modulePath;
15         int _id; /*< Device id */
16         int _fd; /*< File descriptor to manage the device */
17         std::string _devPath;
18         std::string _modulePath;
19     public:
20         DigitalOutput(std::string devName, int id);
21         ~DigitalOutput();
22         bool Open();
23         void Write(bool enable);

```

```

24     void Close();
25 };
26 };

```

Listing 5.23: DigitalOut — interface

## Fragrance classes

The fragrance classes, enclosed in the `Fragrance` namespace, handle the fragrance logic, namely, creating, diffusing and managing it.

### Frag

The `Frag` class defines a Fragrance as illustrated in Listing 5.24. It allows to calculate the durations of the enabled and disabled state of the fragrance diffuser associated with it, and besides the normal getters and setters, it provides methods to serialize and deserialize this object, enabling it to be saved and loaded from the database.

```

1 /**
2  * @file frag.h
3  * @author Jose Pires
4  * @date 2022-02-03
5  *
6  * @brief Fragrance class
7  */
8 #ifndef _FRAG_H_
9 #define _FRAG_H_
10 #include <string>
11
12 /**
13  * @brief Fragrance namespace
14 */
15 namespace Frag{
16 #define VOL_MAX 100
17     enum Intensity_t {
18         LOW = 1,
19         MEDIUM,
20         HIGH,
21     };
22     typedef enum Intensity_t Intensity;

```

```

24
25     /**
26      * @brief Fragrance class
27      */
28
29     class Fragrance{
30
31     private:
32
33         int _id; /*< fragrance ID */
34         Intensity _intensity; /*< fragrance intensity */
35         int _duration_on; /*< On time (in ms) */
36         int _duration_off; /*< Off time (in ms) */
37         int _vol_max; /*< Max volume (in ml) */
38         int _vol_level; /*< Max volume (in ml) */
39
40         void calcDurations(Intensity intensity); /*< Calculate durations */
41
42     public:
43
44         /*< Constructors/Destructors */
45         Fragrance(int id = 0, Intensity intensity = Intensity::LOW,
46                   int vol_max = VOL_MAX, int vol_level = VOL_MAX);
47
48         ~Fragrance();
49
50         /*< Getters */
51
52         inline int id() const{
53             return _id;
54         }
55
56         inline Intensity intensity() const{
57             return _intensity;
58         }
59
60         inline int durationOn() const{
61             return _duration_on;
62         }
63
64         inline int durationOff() const{
65             return _duration_off;
66         }
67
68         inline int volMax() const{
69             return _vol_max;
70         }
71
72         inline int volLevel() const{
73             return _vol_level;
74         }
75
76         inline bool operator==(const Fragrance& rhs)
77         { return (this->_id == rhs._id); }
78
79         /*< Setters / Mutators */
80
81         void setIntensity(Intensity intensity);
82
83         inline void setVolLevel(int level){
84             if( !(level < 0) )
85                 vol_level = level;
86         }

```

```

66     }
67     /**
68      * @brief Serializing object into string
69      * @param s: string with the output result of the serialization
70      *
71      * Useful to write to streams
72      */
73     void serialize(std::string &s);
74     /**
75      * @brief Deserializing string into object
76      * @param s: input string containing the data to build the object
77      *
78      * Useful to read from streams
79      */
80     void deserialize(const std::string &s);
81 };
82 }

#endif // _FRAG_H_H

```

Listing 5.24: Frag – interface

## fragDiffuser

The fragrance diffuser class associates a fragrance with the respective device driver, enabling it to appropriately handle the requirements of each fragrance (see Listing 5.25). It is important to note that this class is thread safe as it includes a synchronization mechanism — a mutex — to prevent unattended access.

```

1 /**
2  * @file fragDiffuser.h
3  * @author Jose Pires
4  * @date 2022-02-03
5  *
6  * @brief Fragrance diffuser class
7  *
8  */

9 #ifndef _FRAGDIFFUSER_H_
10 #define _FRAGDIFFUSER_H_

11 #include <pthread.h>
12 #include "frag.h"

```

```

15 #include "ddDigitalOut.h"

17 namespace Frag{
18     class Diffuser{
19     private:
20         static int Instances;
21         bool _enabled; /*< asserts if the ad is currently enabled */
22         pthread_mutex_t _mutex; /*< protect access to the enabled state */
23         int _id; /*< device driver ID associated */
24         Fragrance _frag; /*< fragrance associated with the diffuser */
25         DeviceDriver::DigitalOutput *_dd;
26         bool _ddWorking;
27     public:
28     /**< Constructors/Destructors */
29     Diffuser(Fragrance &f);
30     ~Diffuser();
31     /**< Getters */
32     /**
33     * @brief Asserts the diffuser state
34     * @return the enables state
35     */
36     bool enabled();
37     /**
38     * @brief Retrieves the fragrance used by the diffuser
39     * @param f: fragrance to be filled (output)
40     */
41     void fragrance(Fragrance &f);
42     /**< Setters / Mutators */
43     /**
44     * @brief Enables/disables the diffuser
45     * @param enabled: the enabling state of the diffuser
46     */
47     void enable(bool enabled);
48     /**
49     * @brief Sets the diffuser fragrance
50     * @param f: fragrance
51     */
52     void setFragrance(const Fragrance &f);
53 };
54 };
55 #endif // _FRAGDIFFUSER_H__H

```

Listing 5.25: fragDiffuser – interface

## fragManager

The **FragManager** classes manages the fragrance database containing its settings (see Listing 5.26). It is important to note that this class is thread safe as it includes a synchronization mechanism — a mutex — to prevent unattended access.

```

1 /**
 * @file fragManager.h
3 * @author Jose Pires
 * @date 2022-02-03
5 *
 * @brief Manages the fragrances for the station
7 */
#ifndef _FRAGMANAGER_H_
#define _FRAGMANAGER_H_
#include "frag.h"
#include <vector>
#include <pthread.h>
namespace Frag{
    class Manager{
private:
    pthread_mutex_t _mutex; /*< protect access to the push and pop */
    std::vector<Fragrance> *_fragrances = nullptr; /*< desc */
    int _activeFrag; /*< Current fragrance in the station (idx) */
public:
    /*< Constructors/Destructors */
    Manager();
    ~Manager();
    /**
     * @brief Adds a fragrance to the Manager
     * @param f: fragrance to be added
     */
    void add(Fragrance &f);
    /**
     * @brief Removed a fragrance from the Manager
     * @param f: the fragrance to be removed
     * @return true if removed, false otherwise
     *
     * It compares the fragrance to be removed with the already present
     * and removes it if found.
     */
    bool remove(Fragrance &f);
    /**
     * @brief Find a fragrance in the manager

```

```
39         * @param f: fragrance to be found
40         * @return the idx if found; -1 otherwise
41         *
42         * It compares the fragrance to be found with the already present
43         * and returns the comparison result.
44         */
45     int find(Fragrance &f);
46 /**
47     * @brief Load fragrances from database
48     * @return true if loaded, false otherwise
49     */
50     bool load();
51 /**
52     * @brief Saves the fragrances to the database
53     * @return true if loaded, false otherwise
54     */
55     bool save();
56 /**
57     * @brief Populates the manager with fragrances
58     * @return true if populated, false otherwise
59     *
60     * If the loading fails, the populate can be used to create the
61     * database.
62     */
63     bool populate();
64 /**
65     * @brief Sets the active fragrance
66     * @param f: the fragrance to set
67     *
68     * It sets the active fragrance. First, it tries to find in the DB.
69     * If not found, it is added to the DB.
70     */
71     void setActiveFrag( Fragrance &f );
72 /**
73     * @brief It retrieves the active fragrance in the station.
74     * @param f: active fragrance (output)
75     * @return true if there is an active fragrance, false otherwise;
76     */
77     bool ActiveFrag(Fragrance &f);
78 };
79 }*/

#endif // _FRAGMANAGER_H__H
```

Listing 5.26: fragManager – interface

## ImgFilter

The ImgFilter class wraps the image filter functionality, mainly as a container (see Listing 5.27).

```
1 #ifndef _IMGFILTER_H_
2 #define _IMGFILTER_H_
3 #include <string>
4 #include <opencv2/core/mat.hpp>
5 class ImgFilter{
6 public:
7     ImgFilter(std::string path, float width_offset_coef,
8               float height_offset_coef, float scale_width,
9               float scale_height);
10    inline bool isEmpty() const { return _empty; }
11    inline float width_offset_coef() const { return _width_offset_coef; }
12    inline float height_offset_coef() const { return _height_offset_coef; }
13    inline float scale_width() const { return _scale_width; }
14    inline float scale_height() const { return _scale_height; }
15    inline cv::Mat & Mat() { return _filter; }
16 private:
17     float _width_offset_coef;
18     float _height_offset_coef;
19     float _scale_width;
20     float _scale_height;
21     bool _empty;
22     cv::Mat _filter;
23 };
24 #endif // _IMGFILTER_H__H
```

Listing 5.27: imgFilter – interface

## msgQueue

The msgQueue class provides an abstraction over the Linux system calls to ease the usage of this IPC mechanism (see Listing 5.28).

```
/*
2  * @file msgqueue.h
3  * @author Jose Pires
4  * @date 2022-02-06
5  *
6  * @brief Message queue wrapper class
7  */
8 #ifndef _MSGQUEUE_H_
9 #define _MSGQUEUE_H_
10 #include <mqueue.h>
11 #include <string>
12 class msgQueue{
13     private:
14         mqd_t _fd; /*< Msg queue file descriptor */
15         std::string _path;
16         char *_buff;
17         int _sz;
18     public:
19         msgQueue(std::string path, int size);
20         ~msgQueue();
21         bool Open();
22         int Receive();
23     };
24 #endif // _MSGQUEUE_H_H
```

Listing 5.28: msgQueue — interface

## pEvent

The pEvent class provides an abstraction over the pthread condition variables, comprising the loosely named POSIX Event. This class enables to emit and receive signals in a much more straightforward way, but always ensuring its integrity, as it is protected by a mutex (see Listing 5.29).

```
/*
2  * @file pEvent.h
3  * @author Jose Pires
4  * @date 2022-02-02
5  *
6  * @brief Pthread event: Provides an abstraction over the pthread condition variables
7  * synchronization mechanism
```

```
  */
8 #ifndef _SYNCOBJECT_H_
9 #define _SYNCOBJECT_H_
10 #include <pthread.h>
11 class pEvent
12 {
13     private:
14         bool signalled;
15         pthread_mutex_t mutex;
16         pthread_cond_t cond;
17     public:
18         pEvent();
19         ~pEvent();
20         void WaitForSignal();
21         void Signal();
22 };
23 #endif // _SYNCOBJECT_H_H
```

Listing 5.29: pEvent – interface

## Post

The Post class handles the Twitter posts, containing the message and the associated media type (see Listing 5.30).

```
1 #ifndef _POST_H_
2 #define _POST_H_
3 #include <string>
4 enum MediaType { PNG = 0, GIF };
5 typedef enum MediaType MediaType_t;
6 class Post{
7 public:
8     Post(std::string msg = "#MDO ", MediaType_t medType = MediaType::PNG);
9     void setMsg(std::string msg);
10    void setMediaType(MediaType_t medType);
11    void Msg(std::string &msg);
12    void MediaPath(std::string &media_path);
13    MediaType_t MediaType() const;
14 private:
15     std::string _msg;
16     std::string _mediaPath;
```

```

17     MediaType_t _mediaType;
18 };
19 #endif // _POST_H__H

```

Listing 5.30: Post – interface

### 5.4.7. Threads

This section addresses the Local System threads, illustrated in Listing 5.31.

```

1 /**
 * @brief Frame grabber thread function
 * @param arg: ptr to a UI::MainWindow
 *
5 * detailed
 */
7 void* MainWindow::frame_grabber_worker_thr(void *arg) {
8     using namespace cv;
9     MainWindow *mw = (MainWindow *)arg;
10    AppMode_t mode = AppMode::INTER;
11    Mat frame;
12    QString label_text;

14    while(1) {
15        /**< Getting the mode to decide flow */
16        mode = mw->appMode();
17        /**< If App was quitted, terminate thread */
18        if(mode == AppMode::QUIT)
19            return NULL;
20        /**< Check for interaction or img filter modes */
21        if(mode == AppMode::INTER || mode == AppMode::IMGFLT
22           || mode == AppMode::SHAR) {
23            if(mw->_video.isOpened()) {
24                /**< Acquiring frame */
25                mw->_video >> frame;
26                if(!frame.empty()) {
27                    emit mw->imgGrabbed(frame);
28                }
29            }
30            else {
31                /**< Open camera to capture video */

```

## 5.4. Local System

---

```
        if (!mw->_video.open(CAM_IDX, CAP_V4L2)) {
            label_text = "ERROR: could not open camera...";
        } else {
            mw->_video.set(cv::CAP_PROP_FRAME_WIDTH, CAMERA_RES_W);
            mw->_video.set(cv::CAP_PROP_FRAME_HEIGHT, CAMERA_RES_H);
            label_text = "Camera OK!";
        }
        emit(mw->textChanged(label_text));
    }
} else {
    /**< If video was opened but we changed mode, close video feed */
    if (mw->_video.isOpened())
        mw->_video.release();
}
}
return NULL;
}
/***
 * @brief Rx worker thread function
 * @param arg: ptr to a UI::MainWindow
 *
 * Handles Rx connection
 */
void* MainWindow::rx_worker_thr(void *arg){
    MainWindow *mw = (MainWindow *)arg;
    QString data;
    while(1){
        /**< Wait for RX signal */
        mw->_ev_rx->WaitForSignal();
        /**< Read TCP data and push it to vector */
        while( mw->_remoteSock->canReadLine() ){
            data = QString( mw->_remoteSock->readLine() );
            std::cout << "Rx Data: " << data.toStdString()
                << std::endl;
            mw->pushTcpData( data );
        }
        /**< Trigger Processing thread */
        mw->_ev_process->Signal();
    }
    return NULL;
}
/***
 * @brief Process worker thread function
*/
```

```

75 * @param arg: ptr to a UI::MainWindow
*
77 * Processes Rx data
*/
79 void* MainWindow::process_worker_thr(void *arg) {
    MainWindow *mw = (MainWindow *)arg;
81     QString data;
#define CMD_AD "A"
83     QStringList list;
     std::string link;
85     while(1){
         /**< Wait for RX signal */
87         mw->_ev_process->WaitForSignal();
         /**< Read TCP data from buffer */
89         while( !mw->_remoteDataBuff->empty() ){
             mw->popTcpData( data );
             /**< Parse input */
91                 list = data.split(',');
93                 if( list.at(0) == CMD_AD ){
                     Ad ad( list.at(2).toStdString(),
                            list.at(3).toStdString(),
                            list.at(4).toInt(),
                            list.at(5).toInt(),
                            list.at(6).toInt()
                           );
                     /**< Update current Ad and fragrance */
95                     mw->setCurAd(ad);
                     Frag::Fragrance f(ad.fragID());
97                     mw->setCurFrag( f );
                     /**< Download media files */
101                    mw->_ev_download->Signal();
                     }
103                }
105            }
107        }
109    return NULL;
}
111 /**
 * @brief GIF save thread function
113 * @param arg: ptr to a UI::MainWindow
*
115 * detailed
*/
117 void* MainWindow::gif_save_worker_thr(void *arg){

```

```

        using namespace cv;

119     MainWindow *mw = (MainWindow *)arg;
    std::string path;
121     while(1){
        /**< Waiting for a UI signal */
        mw->_ev_gif_save->WaitForSignal();
        /**< Save file to disk */
125     mw->_post.setMediaType(MediaType::GIF);
        mw->_post.MediaPath(path);
127     writeImages( mw->images.begin(), mw->images.end(), path );
        /**< Update Status */
129     mw->updateStatusBar("GIF saved!");
        /**< Reset gif vector */
131     mw->images.clear();
        /**< Enable UI pushbuttons again */
133     mw->_interWind->updateGIFStatus();
    }
135     return NULL;
}
137 /**
 * @brief Download Ad worker thread function
 * @param arg: ptr to a UI::MainWindow
 *
141 * Downloads Ad using curl
 */
143 void * MainWindow::download_ad_worker_thr(void *arg){
    MainWindow *mw = (MainWindow *)arg;
    std::string link;
    Ad ad;
147 #define CMD_AD "A"
    QStringList list;
149     while(1){
        /**< Wait for Download signal */
        mw->_ev_download->WaitForSignal();
        /**< Get current ad */
153     mw->curAd(ad);
        /**< Download link */
155     ad.link(link);
        /**< Update status bar */
157     mw->updateStatusBar( "Downloaded " + QString::fromStdString(link) );
        std::cout << "Downloaded " << link << std::endl;
159     }
    return NULL;
}

```

161 }

Listing 5.31: Local system threads

### frameGrab

The frameGrab thread is responsible for acquiring a camera frame and feeding the computer vision engine to detect faces, recognize gestures, or apply image filters. It checks the mode and if adequate the frame is captured and dispatched for further processing.

### Rx and ProcessRx

The Rx and ProcessRx threads work together following the producer–consumer model: the first receives data frames from the Remote System and pushes it to a shared buffer, signaling this event to the ProcessRx thread; the second consumes these data frames, parsing them, and signaling the relevant events, like downloading an Ad.

### gifSave

The gifSave thread takes care of saving a GIF into disk. A thread is required because this I/O operation can take a significant amount of time.

### DownloadAd

The downloadAd thread takes care of downloading an ad and the associated media through the proxy server transfer.sh. When the event ev\_download is signaled, the current ad is retrieved and the ad is downloaded.

### Main thread

The main thread belongs to the UI, and it responsible for processing its events. Associated with this thread, a periodic task is executed to verify the application mode and stimulate the application (see Listing 5.32), namely: checking normal mode, checking if a user was detected, or if a remote connection was lost.

```
void MainWindow :: onCheckModeTimerElapsed () {  
3 #define PERIODIC_LIMIT_MS ((int)3600000)
```

```

#define CNT_RELOAD ((int)( PERIODIC_LIMIT_MS / MODE_PERIOD_MS ) )
5   static int cnt = CNT_RELOAD; /*< define ratio between normal and interaction mode checks
                                : interaction mode is checked cnt times more than normal */
6   AppMode_t mode;
7   Ad ad;
8
9   if (!cnt) {
10    /*< Update reload */
11    cnt = CNT_RELOAD;
12    /*< Check Normal Mode */
13    /* If current Ad timeslot == time timeslot */
14    if (curAd.timeslot() == _timeslot)
15      onNormalMode_pressed();
16  } else {
17    /*< Check Interaction mode */
18    mode = _appmode;
19    /*< If User detected on Welcome or Normal modes */
20    if (detectUser()) {
21      if ((mode == AppMode::WELCOME) || (mode == AppMode::NORMAL)) {
22        updateStatusBar("User detected");
23        onInter_mode_pressed();
24      }
25    } else {
26      curAd(ad);
27      if (ad.enabled())
28        onNormalMode_pressed();
29      else
30        onHome_pressed();
31    }
32    /*< Check Remote connection */
33    if (!_remoteConnected)
34      connectToRemote();
35    else {
36      const QString str = "Hello from LS";
37      _remoteConnected = true;
38      _remoteSock->write(str.toLocal8Bit());
39      std::cout << "Write: " << str.toStdString() << std::endl;
40    }
41  }
}

```

Listing 5.32: Main thread periodic task: checkMode

## **5.5. Deployment**

As aforementioned the implementation of the Local System was performed and tested on the host before cross-compiling and deploying to the target embedded system.

However, and even though a successful target system image was obtained, the deployment was not possible as the cross-compiling revealed itself more complicated due to the dependency on external libraries.

Nonetheless, as the application logic was successfully validated, the deployment task resides only on the successful cross-compiling which could not be attended due to time restrictions.

# 6. Testing

In this section the hardware and software tests performed are described on a unit and integrated level.

## 6.1. Remote Client

In the Remote Client application, there were many test cases to implement in order to verify all the integrity of the app. There are two main types that can be distinguished: layout tests and logic tests.

### 6.1.1. Layout

On the layout tests, the main tests were functional, such as try to navigate between windows, verify some critical decisions of the user, prevent some bugs on some button click, and so on.

On fig [6.1a](#) shows the behaviour of the application when the user clicks on the exit button, while on fig [6.1b](#) shows the behavior to a user trying to log out. These are just some examples of test cases on the layout that turned out to behave as expected, such as the others not mentioned.

### 6.1.2. Logic

On the logic tests, there were two different types of tests: the ones who were only with Qt based inputs and classes and error handling with queries on MySQL Server.

#### Interface based tests

The interface based tests are related to validate some kind of inputs before the data processing or data sent, in order to minimize errors or unexpected scenarios as much as possible. Two examples of this type of tests can be shown on the register view, where the user needs to input a valid email with the character '@' and the string ".com" (Fig. [6.4a](#)) and the need to validate the password (Fig. [6.4b](#)).

### 6.1. Remote Client

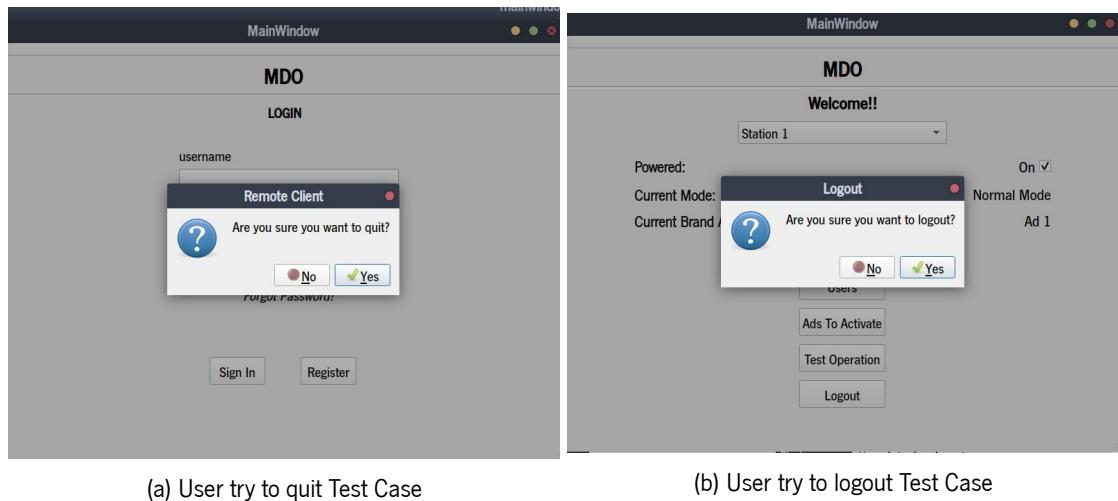


Figure 6.1.: MDO-L Layout Test Cases

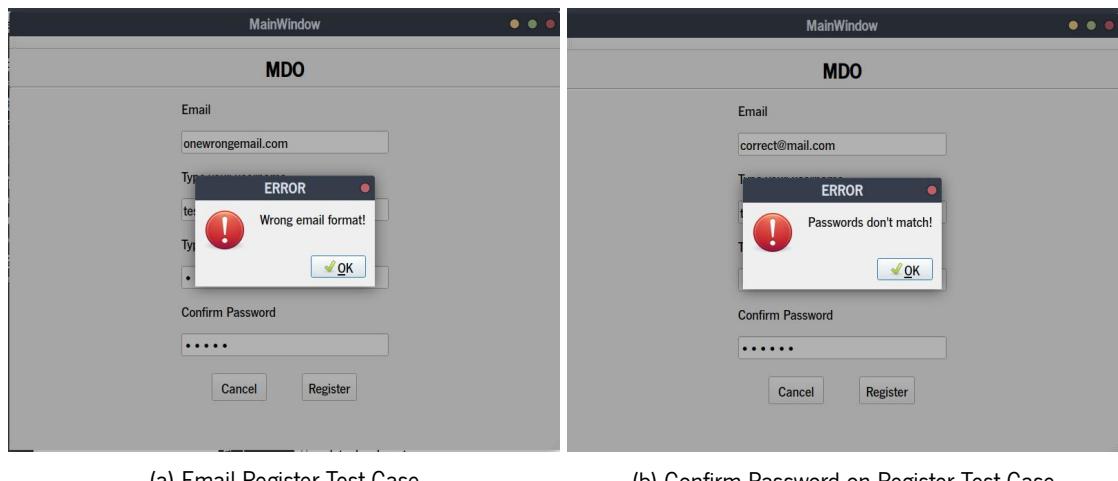


Figure 6.2.: MDO-L Logic (interface based) Test Cases

### Error handling tests (MySQL)

The error handling tests are related mainly with queries with MySQL that can not return as expected. One good example of that is on the login menu, because if there's no username and password matching the input, the SQL server will not return any type of data, and in this case it is necessary to handle that error and process it in the best way possible.

Fig. 6.3 depicts the example described above.

Also, tests related to add a new user can be made to verify if a user is created. Fig 6.4 shows an example of that

## 6.2. Remote Server

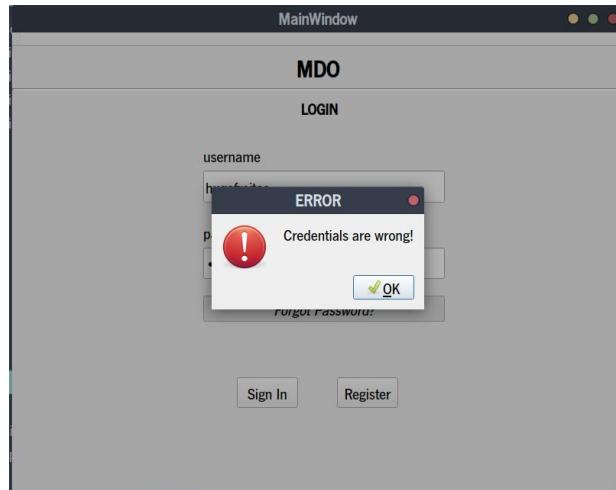


Figure 6.3.: Bad Login Test Case

The image is divided into two parts. Part (a) shows a "New Register Test Case" where a user has entered "teste@email.com" in the "Email" field. A modal dialog box titled "SUCCESS" displays the message "User created Successfully!" with an "OK" button. Part (b) shows a "Confirm Register on SQL Test" with a screenshot of a terminal displaying a SQL dump of a database table. The table has columns: id, role, username, and email. The data shown is:

id	role	username	email
1	1	hugofreitas	hugofreitas12@gmail.com
2	0	zepires	a50178@alunos.uminho.pt
4	0	teste	teste@email.com

(a) New Register Test Case

(b) Confirm Register on SQL Test

Figure 6.4.: Register Test Case

## 6.2. Remote Server

In the Remote Server application, the tests that were needed to make were exclusively with the good creation of the database, the connection between local system and remote client and the upload and update local system.

In Fig. 6.5 is the proof that the database was well created with the scripts created.

Now, on Figure 6.6 is a validation of the connection between local system and remote client.

Lastly, on Fig. 6.7a and Fig. 6.7b is the Test case of the update of the local system uploading the file of a new Ad.

### 6.3. Local System

---

```
MySQL [localhost:33060+ ssl MDO SQL] > show tables;
+-----+
| Tables_in_MDO |
+-----+
| Ad           |
| Filter       |
| Fragrance    |
| FragranceList |
| MediaFile    |
| Station      |
| TimeSlot     |
| TimeTable    |
| User         |
| UserStations |
+-----+
10 rows in set (0.0009 sec)
```

Figure 6.5.: Database Test Case

```
Entrou em Enviado
Socket created
bind done
Waiting for incoming connections...
Connection accepted
Enviado!!
Received: Teste12
Received: Funciona?
Received: Funcionasim
Received: TopTopnasim
```

Figure 6.6.: Communication Between systems Test Case

## 6.3. Local System

The Local System is the most critical and more susceptible subsystem to cause errors. The probability of one error occur is bigger than on the other two subsystems and that's why this system is the one that needs to perform more test cases to be in the best performance possible.

The test cases can be divided in two main parts: hardware tests and software tests.

### 6.3.1. Hardware Tests

There are many parts of hardware that need to be tested:

- Ultrasonic Sensors;
- Fragrance Diffusion (Actuator and Module);
- Camera;
- Speakers;
- Screen;

#### Ultrasonic Sensors

The ultrasonic sensors need to be with strong connections to all the supply sources and GPIO pins, after that, it has been ran a driver program that returned if the two sensors detected the presence of an obstacle.

### 6.3. Local System

```
hugof37@hugof37-FX503VD:~/Desktop $ ./client
Socket created
Connected

Server reply :
A,57,doggie.mp4,https://transfer.sh/QwHj0w/doggie.mp4,1,1,95,1
[]
```

(a) Send and upload of file

doggie.mp4

type: video/mp4  
size: 10,150,671 bytes

download



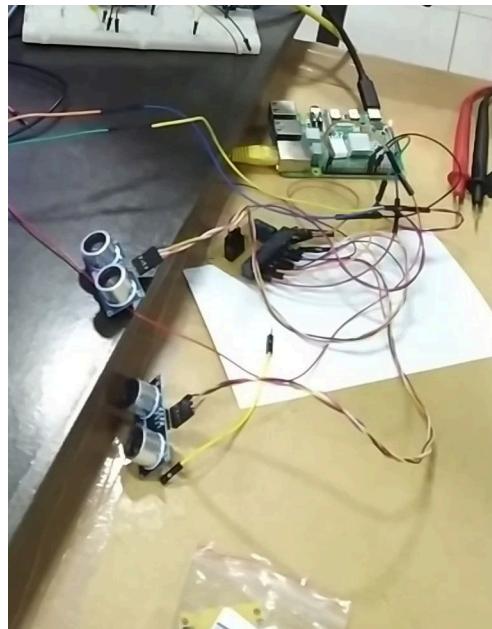
(b) Download of file

Figure 6.7.: Update and upload Test

This program has a specific sample time that is more than enough to detect the presence of something in front of the sensors. In fig. 6.8a is the cable management of the sensors, while in fig. 6.8b is the test output that ran as expected.

### Fragrance Diffusion (Actuator and Module)

The fragrance diffuser module and its respective actuator need to be tested in order to respond to some signals given by the main board. In order for this to happen, it was tested with a driver program on the board and with all the module and its components and the result was, as it can be seen in Fig. 6.9 what one expected.



(a) Sensors Cable Management

```
# cat /var/log/distance.log
0
0
1
1
1
1
0
1
0
0
0
```

(b) Sensors Output Test

Figure 6.8.: Ultrasonic Sensors Test Cases

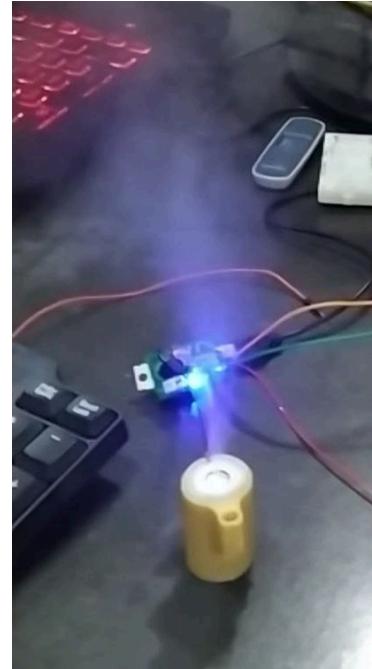


Figure 6.9.: Fragrance Diffuser Output Test

## Camera

For the interaction mode it is mandatory that the camera module works properly to take pictures and gifs. For this piece of hardware the test case is simple: turn on the camera and try to take a photo.

### 6.3. Local System

---

In this case it was used an image of Raspbian and with the help of an online app, the test of the camera worked as well as expected. The result is on Fig. 6.10.

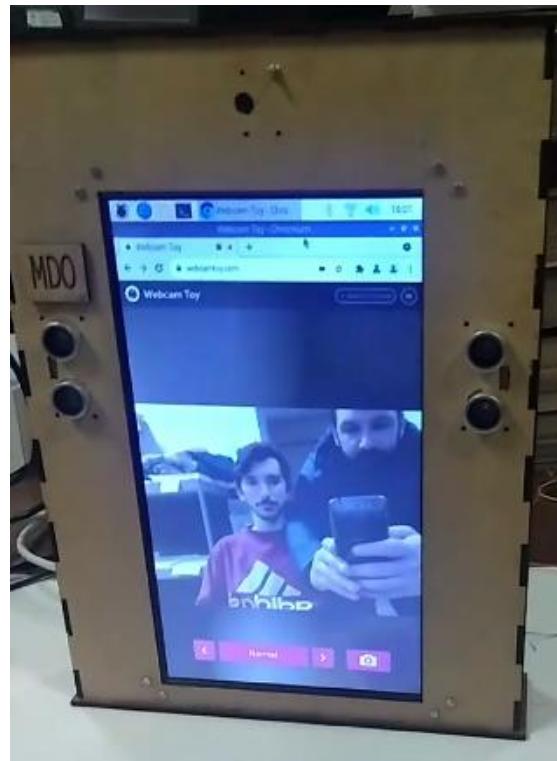


Figure 6.10.: Camera Output Test

## Speakers

The speakers test cases are in a certain way different to test, because there's no way to show how it worked. However, the test was as simple as connect the speakers to the screen module board and play a video or an audio.

The audio played perfectly and the sound was well detected by human ears.

## Screen

Testing the screen is similar to test the speakers. It's just simply connect the screen to the board and test its execution. As it can be seen in the previous figure when testing the camera (Fig. 6.3.1) the screen was already being tested and it's more than proved that the screen works with no problems.

In conclusion, all hardware components and modules were tested successfully, which means that all test cases are now validated and it is possible to take the next step.

## 6.4. Software

In this section the software tests conducted over the Remote Client, Remote Server, Database and Local System are presented.

### 6.4.1. Local System

The Local system tests were performed on a host computer, prior to its deployment to the Raspberry Pi. These tests are described next.

#### Computer vision

In this section are described the frame acquisition, face detection, and gesture recognition tests. Fig. 6.11 illustrates the combination of these tests. It can be seen that the camera frames are acquired and processed to detect multiple faces and apply filters overlay, and also to detect gestures that can be used to trigger UI events. Additionally, it can be seen a picture that was taken, stored and displayed, which is ready for sharing on social media.

#### Normal mode

Fig. 6.12 illustrates the normal mode testing. It demonstrated that a video can be reproduced on a loop while the fragrance diffuser was also enabled and disabled according to its on and off times. The normal mode only exited if there was no current ad enabled or if a user was detected.

#### Interaction mode

The interaction mode tests demonstrated that it was possible to take pictures and create GIFs on demand, as illustrated in Fig. 6.11.

#### Twitter sharing

Fig. 6.13 illustrates the Twitter sharing mode. It demonstrated that a post can be successfully shared from the Local System into Twitter. Nonetheless, at the current time, it is still not possible to upload media to Twitter, as the API recently changed, requiring extra time to implement.

#### 6.4. Software

---



Figure 6.11.: Computer visions tests

#### Image filtering mode

From Fig. 6.11 and Fig. 6.13, it is possible to verify that several filters were successfully implemented. Moreover, it is possible to observe from the last figure that the image filter can be persistent if desired.

#### 6.4. Software

---



Figure 6.12.: Normal mode: testing

#### Download Ad

Fig. 6.14 illustrates the download of an Ad in the background, while the application was currently busy in the interaction mode.

#### 6.4. Software

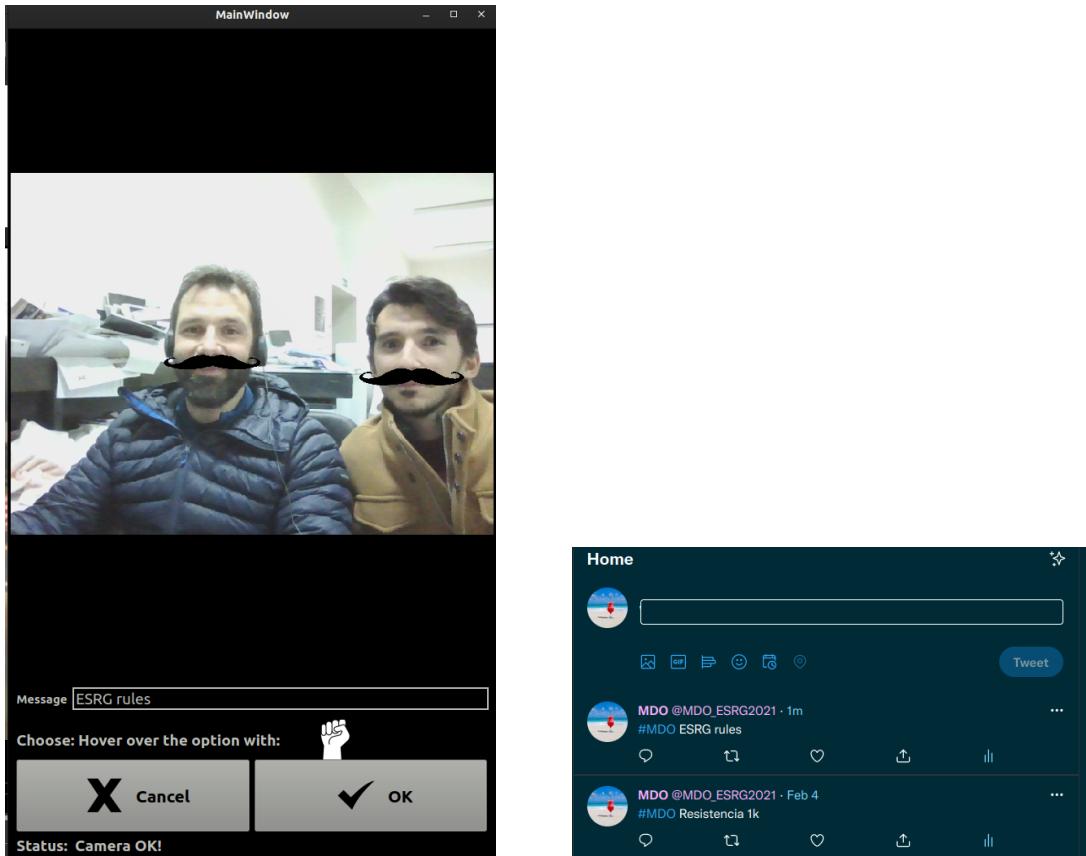


Figure 6.13.: Twitter sharing: testing

### Connection to Remote System

The connection to the remote system was simulated and tested deploying a server listening in the localhost and observing the connection and data exchange between the two nodes. This test was successful validating the client-server architecture logic and implementation.

### User detection

The user detection was tested by deploying the user detection daemon to stimulate the application. It was verified that the daemon successfully detects the ultrasonic sensors triggering when a person passes and that this event is conveyed to the local system via message queue where it is successfully detected, signaling a user was detected.

#### 6.4. Software

---



Figure 6.14.: Download Ad: testing

# **7. Conclusion**

In this chapter are outlined the conclusions and the prospect for future work regarding the marketing digital outdoor.

## **7.1. Conclusions**

The Marketing Digital Outdoor can be described as a multisensory marketing device, developed to help brands to share their products in a more efficient and attractive way through a photo booth with gesture recognition and face based filters, video and sound display, pictures and gif taking and also fragrance emission.

This project followed closely the waterfall methodology, which assists the designer to quickly and systematically develop cyber physical systems.

In the analysis stage, the foreseen specifications were listed, as well as the envisioned tests for verification and validation of the product. The analysis elicited the main requirements and constraints about the problem at hands, easing the conception of a design solutions pool. It was clear from early on that the development of a distributed architecture was obviously the most extensible and efficient way to solve the problem statement.

In the design phase, the considerations drawn in the analysis, combined with the initial design, were used to conceptualize a viable solution for the product materialization, comprising three main components: remote client, remote server and local system.

The implementation phase mapped the devised solution into actual software and hardware modules for the several subsystems. The Remote Client was successfully implemented, enabling a Brand or a Admin user to interact with the MDO device with a scalable structure, consisting of a UI and MySQL database. The Local System implemented all the major features proposed namely multisensory marketing with video and audio reproduction, fragrance diffusion, user interaction through a non-physical interface (gestures), providing image filtering, picture and GIF sharing features. Additionally, Twitter sharing was added to share posts as a way to increase brand awareness. Related to the technical details, a distributed architecture was successfully implemented in a multithreaded way, enabling multiple concurrent actions to be performed

## **7.2. Prospect for Future Work**

---

without impact to system performance from the user perspective. Device drivers, daemons, message queues, mutexes, conditional variables were used for this purpose.

Considering the deployment, the process was well established considering the custom embedded Linux OS generation and initialization. However, problems in the cross compilation of the application prevented the successfully deployment of the application on the target hardware on a timely manner.

The tests performed over the hardware and software components demonstrated the implemented solution performed according to the expectations, although not in the target hardware conceived.

From a critical point of view, this project provides a solid base for developing robust embedded systems, as the process, although laborious and complex, was well supported, proving the concept. The deployment is lacking, but, is the authors opinion, that this is only a question of time, as the logic behind was proven.

## **7.2. Prospect for Future Work**

In the foreseeable future, the deployment of the local system to the target hardware must be performed and intensively tested to analyze system behavior and performance. Twitter sharing with media upload is lacking due to Twitter API changes and is an important feature. Additionally, the gesture detection engine could be improved by training more accurate models.

On the Remote System side, the Remote server must be completed and deployed externally as part of some cloud-based solution.

# 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.

## BIBLIOGRAPHY

---

- [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](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.

## BIBLIOGRAPHY

---

- [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] Source code documentation best practices, 2017. URL <https://easternpeak.com/blog/source-code-documentation-best-practices/>. accessed: 2021-12-2.
- [29] Doxygen, . URL <https://www.doxygen.nl/index.html>. accessed: 2021-12-2.
- [30] Doxygen: Documenting the code, . URL <https://www.doxygen.nl/manual/docblocks.html>. accessed: 2021-12-2.
- [31] Tengteng Wen, Dehan Luo, Yongjie Ji, and Pingzhong Zhong. Development of a piezoelectric-based odor reproduction system. Electronics, 8(8):870, 2019.
- [32] What's the difference between an ultrasonic diffuser and a nebulizing diffuser. URL <https://www.aromaoutfitters.com/whats-the-difference-between-an-ultrasonic-diffuser-and-a-nebulizing-diffuser/>. accessed: 2021-12-5.
- [33] Saad Hameed Abid, Zhiyong Li, Renfa Li, and Jumana Waleed. A novel olfactory displays' scent dispersing module. International Journal of Hybrid Information Technology, 8(1):435–442, 2015.
- [34] 113khz d20mm ultrasonic mist maker atomizing fogger humidifier module w/ pcb. URL <https://shorturl.at/hlxFQ>. accessed: 2021-12-6.
- [35] Make your own super simple ultrasonic mist maker. URL <https://www.instructables.com/Make-Your-Own-Super-Simple-Ultrasonic-Mist-Maker/>. accessed: 2021-12-6.
- [36] Adrian Kaehler and Gary Bradski. Learning OpenCV 3: computer vision in C++ with the OpenCV library. " O'Reilly Media, Inc.", 2016.
- [37] 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.

## BIBLIOGRAPHY

---

- [38] 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.
- [39] 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.
- [40] Cha Zhang and Zhengyou Zhang. A survey of recent advances in face detection. 2010.
- [41] Paul Viola and Michael J Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.
- [42] 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.
- [43] 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.
- [44] 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.
- [45] 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.
- [46] 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.
- [47] 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.
- [48] 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.
- [49] 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.

## BIBLIOGRAPHY

---

- [50] 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.
- [51] Cascade classifier. URL [https://docs.opencv.org/3.4/db/d28/tutorial\\_cascade\\_classifier.html](https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html). accessed: 2021-11-22.
- [52] Haar cascades, explained. URL [https://docs.opencv.org/3.4/db/d28/tutorial\\_cascade\\_classifier.html](https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html). accessed: 2021-11-22.
- [53] H Aashni, S Archanasri, A Nivedhitha, P Shristi, and SN Jyothi. International conference on advances in computing & communications. *Science Direct*, 115:367–374, 2017.
- [54] Mais Yasen and Shaidah Jusoh. A systematic review on hand gesture recognition techniques, challenges and applications. *PeerJ Computer Science*, 5:e218, 2019.
- [55] 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.
- [56] 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.
- [57] 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-e09b0a12b0> accessed: 2021-11-28.
- [58] 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.
- [59] Raghu Ramakrishnan, Johannes Gehrke, and Johannes Gehrke. *Database management systems*, volume 3. McGraw-Hill New York, 2003.
- [60] Crow's foot notation. URL <https://vertabelo.com/blog/crow-s-foot-notation/>. accessed: 2021-11-29.
- [61] Which modern database is right for your use case? URL <https://www.xplenty.com/blog/which-database/>. accessed: 2021-11-30.

## BIBLIOGRAPHY

---

- [62] Sqlite vs mysql – what's the difference, . URL <https://www.hostinger.com/tutorials/sqlite-vs-mysql-whats-the-difference/>. accessed: 2021-11-30.
- [63] Carlos Coronel and Steven Morris. Database systems: design, implementation, & management. Cengage Learning, 2016.
- [64] Mysql connectors and apis, . URL <https://dev.mysql.com/doc/index-connectors.html>. accessed: 2021-11-30.
- [65] 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.
- [66] 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.
- [67] The beginner's guide to motion sensors, . URL <https://www.safewise.com/resources/motion-sensor-guide/>. accessed: 2021-12-6.
- [68] Understanding how ultrasonic sensors work, . URL <https://www.linearmotiontips.com/ultrasonic-sensors-for-linear-position-and-distance-measuring/>. accessed: 2021-12-6.
- [69] Understanding how ultrasonic sensors work, . URL <https://www.maxbotix.com/articles/how-ultrasonic-sensors-work.htm>. accessed: 2021-12-6.
- [70] How to spot the biscuit thief with raspberry pi and c++, . URL <https://www.rs-online.com/designspark/beginners-guide-to-computer-vision-with-raspberry-pi-4>. accessed: 2021-12-6.
- [71] What are video encoding formats? | video formats, . URL <https://www.cloudflare.com/learning/video/video-encoding-formats/>. accessed: 2021-12-8.
- [72] Using your raspberry pi as a video player, . URL <https://pimylifeup.com/raspberry-pi-video-player/>. accessed: 2021-12-8.
- [73] Working with video using opencv and qt, . URL <http://codingexodus.blogspot.com/2012/12/working-with-video-using-opencv-and-qt.html>. accessed: 2021-12-9.
- [74] Give hermione granger a cool pair of glasses by building snapchat filter using opencv. URL <https://www.analyticsvidhya.com/blog/2021/07/give-hermione-granger-a-cool-pair-of-glasses-by-building-snapchat-filter-using-opencv/>. accessed: 2021-12-10.

## BIBLIOGRAPHY

---

- [75] Graphics interchange format (gif) specification. URL <https://www.w3.org/Graphics/GIF/spec-gif87.txt>. accessed: 2021-12-10.
- [76] John Miano. Compressed image file formats: Jpeg, png, gif, xbm, bmp. Addison-Wesley Professional, 1999.
- [77] Imagemagick. URL <https://imagemagick.org/index.php>. accessed: 2021-12-10.
- [78] Magickwand, . URL <https://imagemagick.org/script/magick-wand.php>. accessed: 2021-12-10.
- [79] Magickcore, . URL <https://imagemagick.org/script/magick-core.php>. accessed: 2021-12-10.
- [80] Magick++, . URL <https://imagemagick.org/script/magick++.php>. accessed: 2021-12-10.
- [81] Magick++ | documentation, . URL <https://www.imagemagick.org/Magick++/Documentation.html>. accessed: 2021-12-10.
- [82] Top 10 social media apis: Twitter, facebook, instagram, and many more. URL <https://medium.com/rakuten-rapidapi/top-10-social-media-apis-twitter-facebook-instagram-and-many-more-5c13262c61fe>. accessed: 2021-12-7.
- [83] Top 10 social media apis for developers. URL <https://www.ayrshare.com/top-10-social-media-apis-for-developers/>. accessed: 2021-12-7.
- [84] Getting started with the twitter api | docs | twitter developer platform, . URL <https://developer.twitter.com/en/docs/twitter-api/getting-started/about-twitter-api>. accessed: 2021-12-8.
- [85] Getting access to the twitter api | docs | twitter developer platform, . URL <https://developer.twitter.com/en/docs/twitter-api/getting-started/getting-access-to-the-twitter-api>. accessed: 2021-12-8.
- [86] What is a rest api? URL <https://www.redhat.com/en/topics/api/what-is-a-rest-api>. accessed: 2021-12-8.
- [87] Make your first request | docs | twitter developer platform, . URL <https://developer.twitter.com/en/docs/twitter-api/getting-started/make-your-first-request>. accessed: 2021-12-8.
- [88] Manage tweets: Introduction | docs | twitter developer platform, . URL <https://developer.twitter.com/en/docs/twitter-api/tweets/manage-tweets/introduction>. accessed: 2021-12-8.

## BIBLIOGRAPHY

---

- [89] Manage tweets: Introduction | docs | twitter developer platform, . URL <https://developer.twitter.com/en/docs/twitter-api/tweets/manage-tweets/quick-start>. accessed: 2021-12-8.
- [90] Post tweets — api reference | docs | twitter developer platform, . URL <https://developer.twitter.com/en/docs/twitter-api/tweets/manage-tweets/api-reference/post-tweets>. accessed: 2021-12-8.
- [91] curl. URL <https://curl.se/>. accessed: 2021-12-8.
- [92] Creating twitter libraries in c++, . URL <https://blog.hashtagify.me/2020/03/18/creating-twitter-libraries-in-c/>. accessed: 2021-12-8.
- [93] libcurl | the multiprotocol file transfer library, . URL <https://curl.se/libcurl/>. accessed: 2021-12-8.
- [94] libcurl bindings, . URL <https://curl.se/libcurl/bindings.html>. accessed: 2021-12-8.
- [95] twitcurl | github repository, . URL <https://github.com/swatkat/twitcurl>. accessed: 2021-12-8.
- [96] C++ ui libraries. URL [https://philippegroarke.com/posts/2018/c++\\_ui\\_solutions/](https://philippegroarke.com/posts/2018/c++_ui_solutions/). accessed: 2021-12-2.
- [97] Qt framework and qml, . URL <https://www.sam-solutions.com/blog/qt-framework/>. accessed: 2021-12-1.
- [98] The building blocks of qt, . URL <https://www.qt.io/product/frameworkhttps://www.qt.io/product/framework>. accessed: 2021-12-2.
- [99] Signals & slots | qt core 5.15.7, . URL <https://doc.qt.io/qt-5/signalsandslots.html>. accessed: 2021-12-2.
- [100] Coffee machine example, . URL <https://doc.qt.io/qt-5/qtdoc-demos-coffee-example.html>. accessed: 2021-12-2.
- [101] Embedded linux – working with qt, . URL <https://devarea.com/embedded-linux-working-with-qt/#Ya3vmWjP0kl>. accessed: 2021-12-6.
- [102] What are the top file transfer protocols? URL <https://www.goanywhere.com/blog/what-are-the-top-file-transfer-protocols>. accessed: 2021-12-8.
- [103] Cisco. Transferring files using http or https. <https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ifs/configuration/xe-16/ifs-xe-16-book/ifs-file-trans-http.pdf>, 2018. accessed: 2021-12-08.

## BIBLIOGRAPHY

---

- [104] Transfer sh website, . URL <https://transfer.sh>. accessed: 2021-12-10.
- [105] Raspberry pi 4 tech specs, . URL <https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/>. accessed: 2021-12-11.
- [106] How hc-sr04 ultrasonic sensor works & interface it with arduino, . URL <https://lastminuteengineers.com/arduino-hc-sr04-ultrasonic-sensor-tutorial/>. accessed: 2021-12-11.
- [107] Micro usb ultrasonic atomizing humidifier module fogger mist generator uk. URL <https://www.ebay.co.uk/itm/203635455945?hash=item2f699e77c9:g:j-cAAOSwE75hVti0>. accessed: 2021-12-11.
- [108] Raspberry pi camera module 2, . URL <https://www.raspberrypi.com/products/camera-module-v2/>. accessed: 2021-12-11.
- [109] 10.1 inch hdmi screen lcd display with audio driver board monitor for raspberry pi banana/orange pi mini computer. URL [https://www.aliexpress.com/item/33005274109.html?spm=a2g0o.detail.1000013.3.38072ea5MrfWq9&gps-id=pcDetailBottomMoreThisSeller&scm=1007.13339.169870.0&scm\\_id=1007.13339.169870.0&scm-url=1007.13339.169870.0&pvid=e1784609-f2f3-4547-9d9e-d5461fddc4c8,scm-url:1007.13339.169870.0,pvid:e1784609-f2f3-4547-9d9e-d5461fddc4c8,tpp\\_buckets:668%232846%238110%231995&&pdp\\_ext\\_f=%7B"sceneld":"3339","sku\\_id":"12000023440173741"%7D](https://www.aliexpress.com/item/33005274109.html?spm=a2g0o.detail.1000013.3.38072ea5MrfWq9&gps-id=pcDetailBottomMoreThisSeller&scm=1007.13339.169870.0&scm_id=1007.13339.169870.0&scm-url=1007.13339.169870.0&pvid=e1784609-f2f3-4547-9d9e-d5461fddc4c8&_t=gps-id:pcDetailBottomMoreThisSeller,scm-url:1007.13339.169870.0,pvid:e1784609-f2f3-4547-9d9e-d5461fddc4c8,tpp_buckets:668%232846%238110%231995&&pdp_ext_f=%7B). accessed: 2021-12-11.
- [110] Passive vs. active speakers, which is right for you? URL <https://www.aperionaudio.com/blogs/aperion-audio-blog/passive-vs-active-speakers>. accessed: 2021-12-11.
- [111] 5w speaker - para display (h). URL <https://www.botnroll.com/pt/colunas-sirenes/2726-5w-speaker-para-display-h.html>. accessed: 2021-12-11.
- [112] Carregador 4x usb 5v / 2,4a (branco) - fonestar. URL <https://www.castroelectronica.pt/product/carregador-4x-usb-5v-24a-branco--fonestar>. accessed: 2021-12-11.
- [113] Raspberry pi 4 pinout, . URL <https://www.the-diy-life.com/raspberry-pi-4-pinout/>. accessed: 2021-12-13.
- [114] Linux hc-sr04 device driver. URL <https://github.com/johannesthma/linux-hc-sr04>. accessed: 2022-01-20.

# **Appendices**

## **A. Project Planning – Gantt diagram**

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

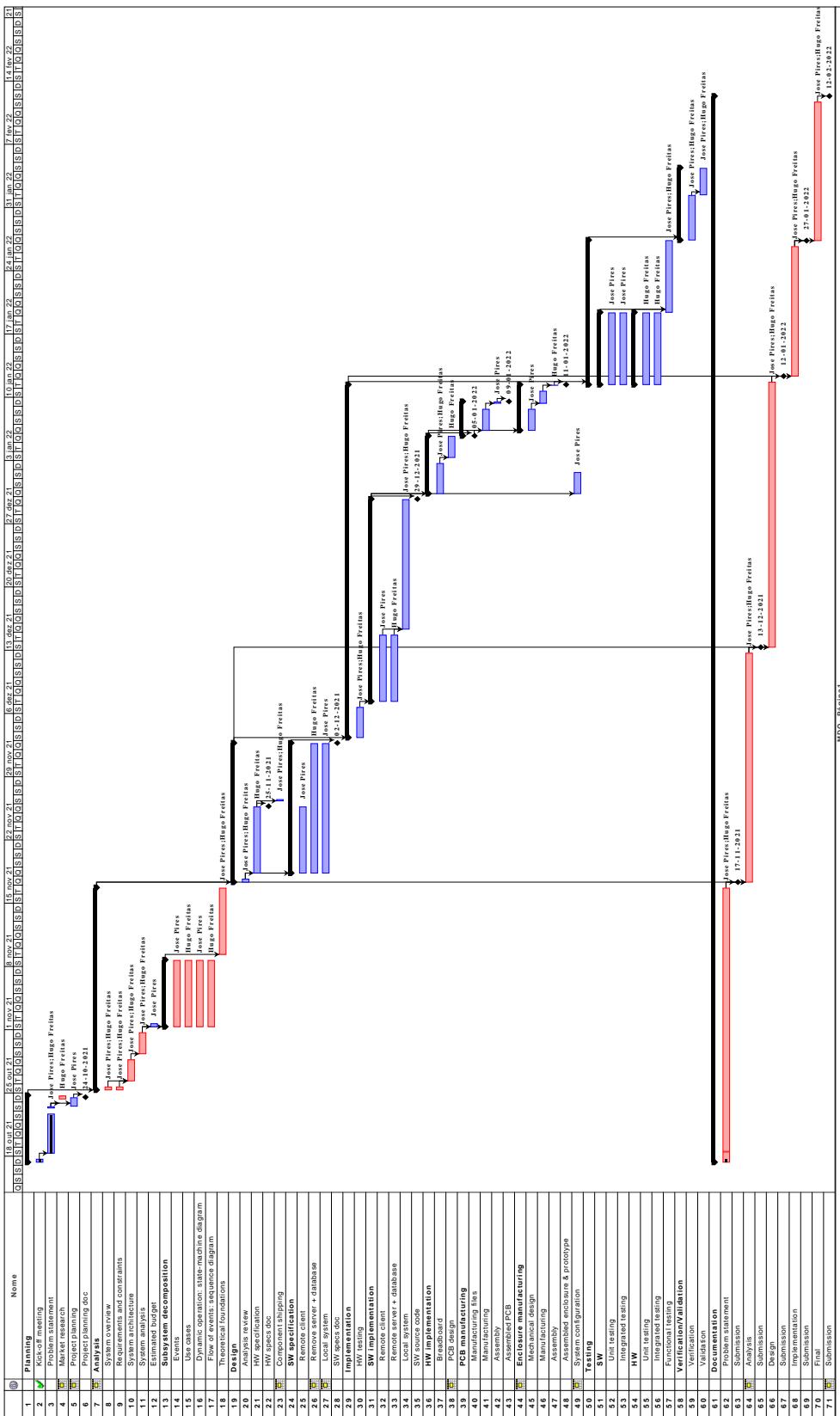


Figure A.1.: Project planning – Gantt diagram