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

November 21, 2021

Contents

List of Figures

List of Abbreviations

Notation	Description	Page List
AC	Alternating Current	12
API	Application Programming Interface	4, 12, 13, 15
BN	Billions	2, 3
BSP	Board Support Package	13–15
CAGR	Compound Annual Growth Rate	3
CLI	Command Line Interface	13, 28
COTS	Commercial off-the-shelf	5
CPS	Cyber—Physical Systems	1, 3
CV	Computer Vision	15
DB	Database	iii, iv, 12, 14, 15, 28, 32, 34, 35, 43, 45–47
DC	Direct Current	12
DOOH	Digital Out-Of-Home	2
GIF	Graphics Interchange Format	iv, 2, 7, 10, 15, 18, 28, 29, 40, 42, 47, 50, 52
HW	Hardware	iii, 5, 8, 11, 43, 53, 54
IP	Internet Protocol	15, 20, 32, 43
IPC	Inter-Process Communication	59

Notation	Description	Page List
MDO	Marketing Digital Outdoor	iii, 2, 8–10, 12
MDO-L	MDO Local System	9, 10, 12, 15, 38, 47
MDO-RC	MDO Remote Client	9, 12, 15, 16
MDO-RS	MDO Remote Server	9, 10, 12, 15
OMT	Object-Modeling Technique	58
OOSE	Object Oriented Software Engineering	58
OS	Operating System	13–15
OSI	Open Systems Interconnection	iv, 13
PCB	Printed Circuit Board	5
POSIX	Portable Operating System Interface	58
R&D	Research and Development	3
RDBMS	Relational Database Management System	14, 32, 34
SoC	System-on-a-Chip	12
SW	Software	iii, 5, 11–16, 53
TCP/IP	Transmission Control Protocol/Internet Protocol	9, 10, 12, 13
UI	User Interface	9, 10, 12, 13, 15, 16, 21, 23, 25, 38, 41, 42, 47
UML	Unified Modeling Language	57

1. Introduction

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

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

1.1. Context and motivation

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

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

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

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

1.2. Problem statement

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

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

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

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

1.3. Market research

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

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

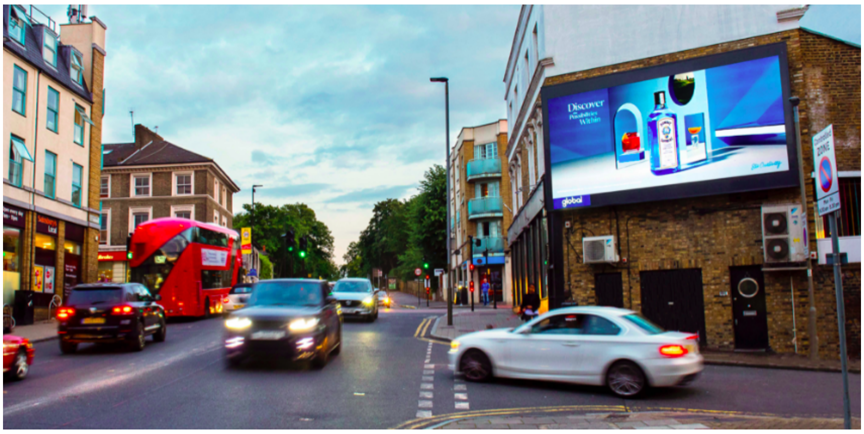


Figure 1.1.: Example of a Digital Outdoor, withdrawn from [4]

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

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

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

1.4. Project goals

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

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

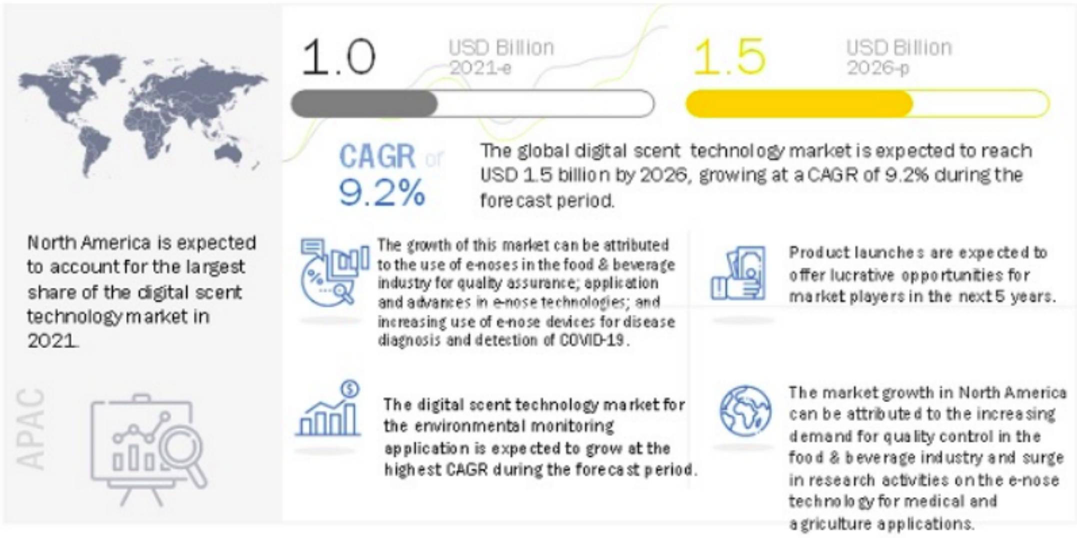


Figure 1.2.: Attractive Opportunities in the Digital Scent Technology Market, withdrawn from [8]

for user input.

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

1.5. Project planning

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

The tasks are described as follows:

- Project planning: in the project planning, a brainstorming about conceivable devices takes place, whose viability is then assessed, resulting in the problem statement (Milestone 0). A market research is performed to assess the product's market space and opportunities. Finally, an initial version of the project planning is conceived to define a feasible timeline for the suggested tasks.

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

1.6. Report Outline

This report is organized as follows:

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

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

- Lastly, the appendices (see Section 3.16) contain detailed information about project planning and development.

2. Analysis

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

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

2.1. Requirements and Constraints

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

2.1.1. Functional requirements

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

2.1.2. Non-functional requirements

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

2.1.3. Technical constraints

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

2.1.4. Non-technical constraints

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

2.2. System overview

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

Considering the system interactions, three main actors were identified:

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

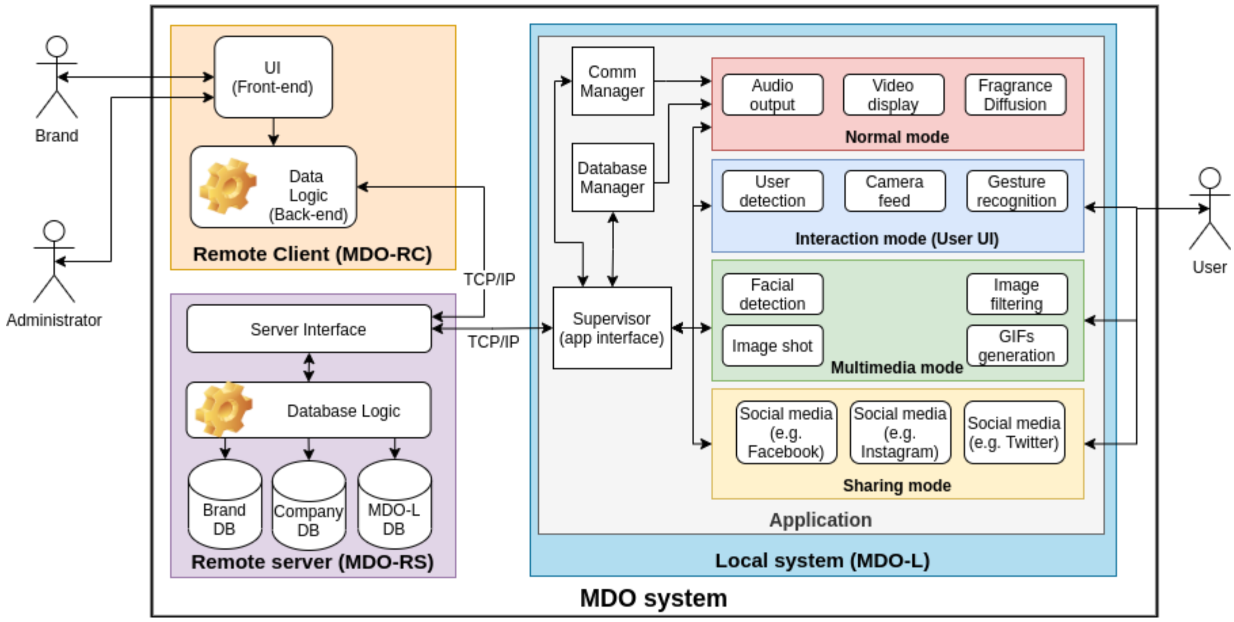


Figure 2.1.: MDO system overview

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

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

2.2.1. MDO Remote Client

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

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

2.2.2. MDO Remote Server

Although the MDO-RC could communicate directly with the MDO-L, this is not desirable or a good architecture mainly due to: communications failure could result in data loss, compromising the system's integrity; the remote client and the local system become tightly coupled, meaning the remote client must be aware of all the available local systems; if the data storage in the local system fails, the remote client would have to provide the backup information.

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

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

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

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

2.2.3. MDO Local system

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

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

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

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

2.3. System architecture

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

2.3.1. Hardware architecture

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

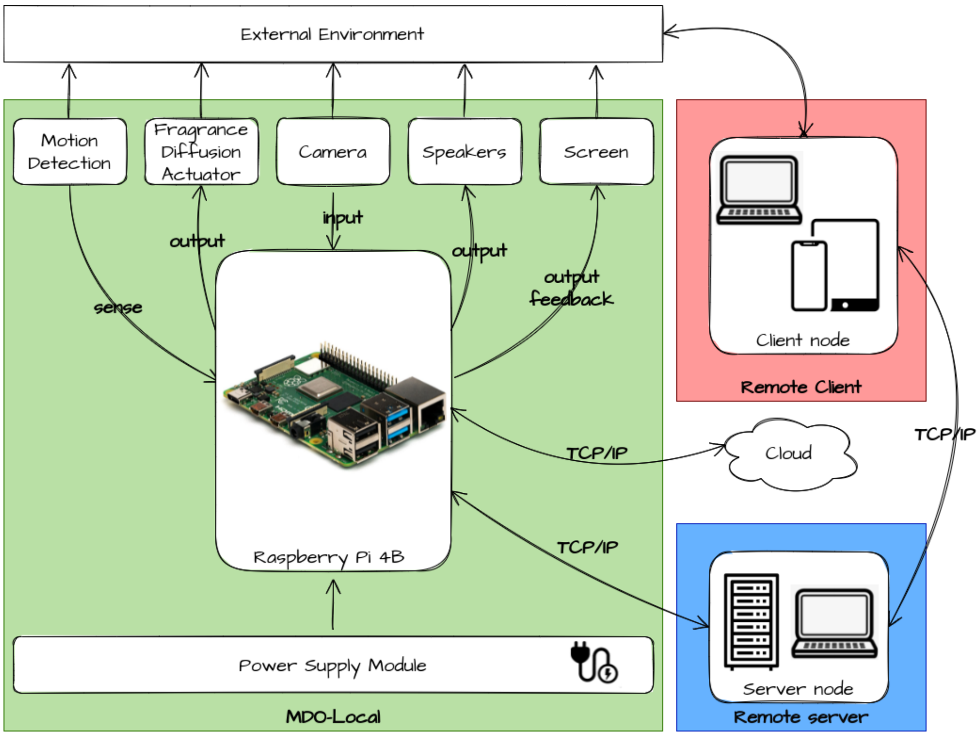


Figure 2.2.: HW architecture diagram

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

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

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

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

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

2.3.2. Software architecture

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

MDO remote client

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

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

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

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

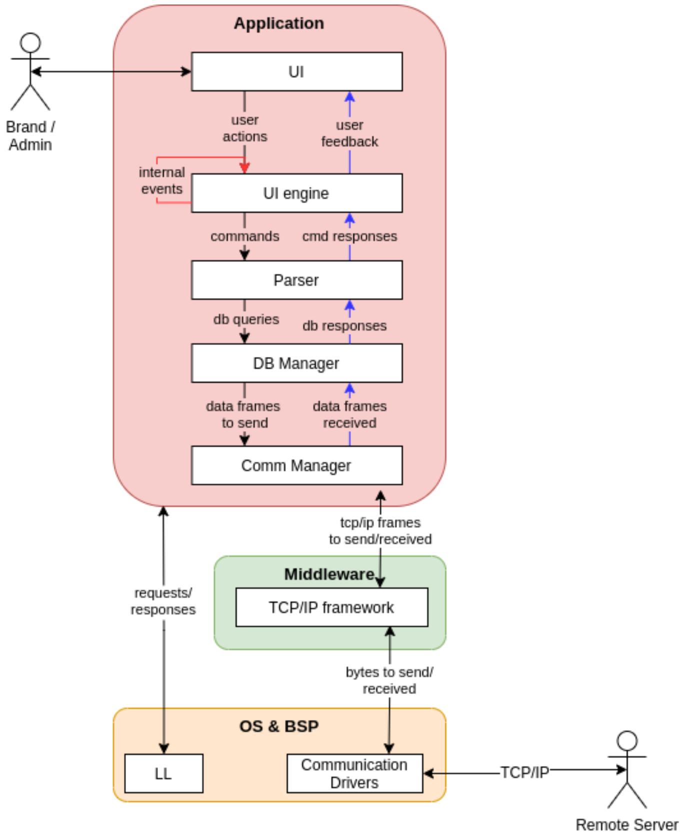


Figure 2.3.: SW architecture diagram: remote client

MDO remote server

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

- Application: contains the remote server application. It provides a Command Line Interface (CLI) to handle **Remote client** requests. The CLI engine is notified and handles all UI events — internal or external — providing the appropriate feedback. The relevant commands are then parsed — **Parser** component — and responded: DB queries are handled by the **Relational Database Management System (RDBMS)** issuing DB transactions; other commands received from the **Remote Client** are handled internally and translated, being dispatched to the **Local System** by the **Comm Manager** (via **Communication drivers**). Internal events can also trigger the **RDBMS** to issue database transactions for the **Remote Client** or **Local System**. The **Comm Manager** is responsible for wrapping/unwrapping the data frames received by or sent to the **Remote Client** or **Local System**.

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

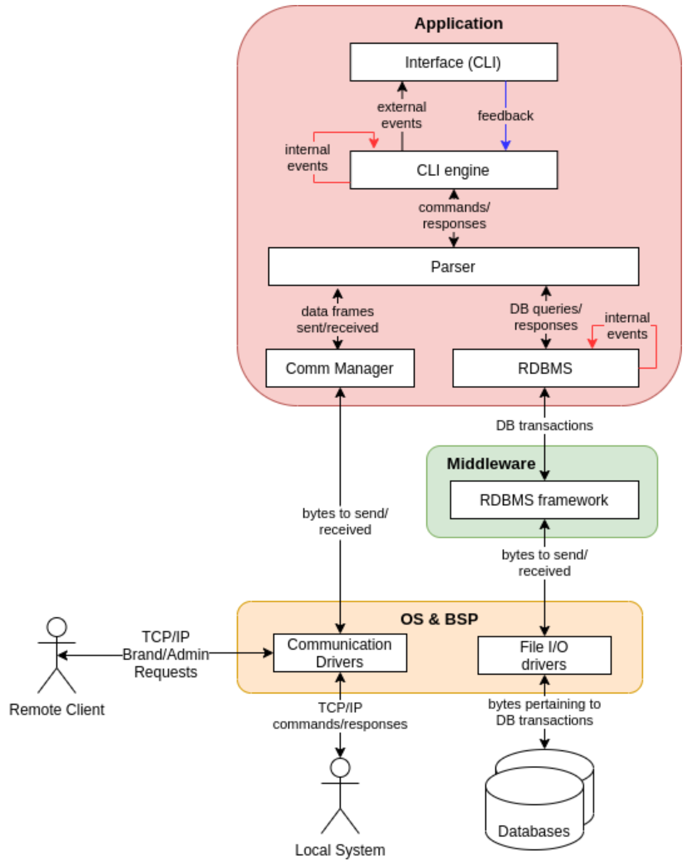


Figure 2.4.: SW architecture diagram: remote server

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

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

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

MDO local system

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

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

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

2.4. Subsystem decomposition

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

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

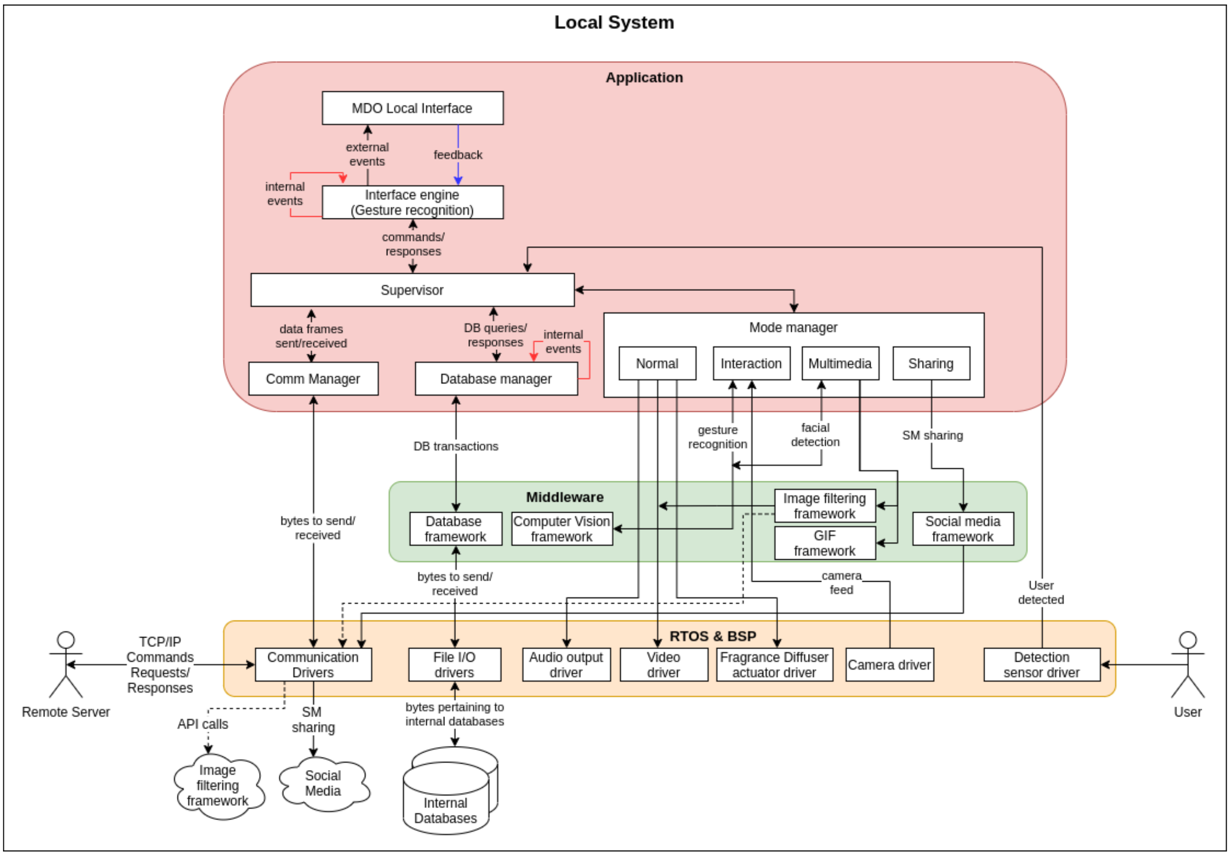


Figure 2.5.: SW architecture diagram: local system

2.4.1. Remote Client

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

User mock-ups

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

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

Firstly, the **Admin** workflow:

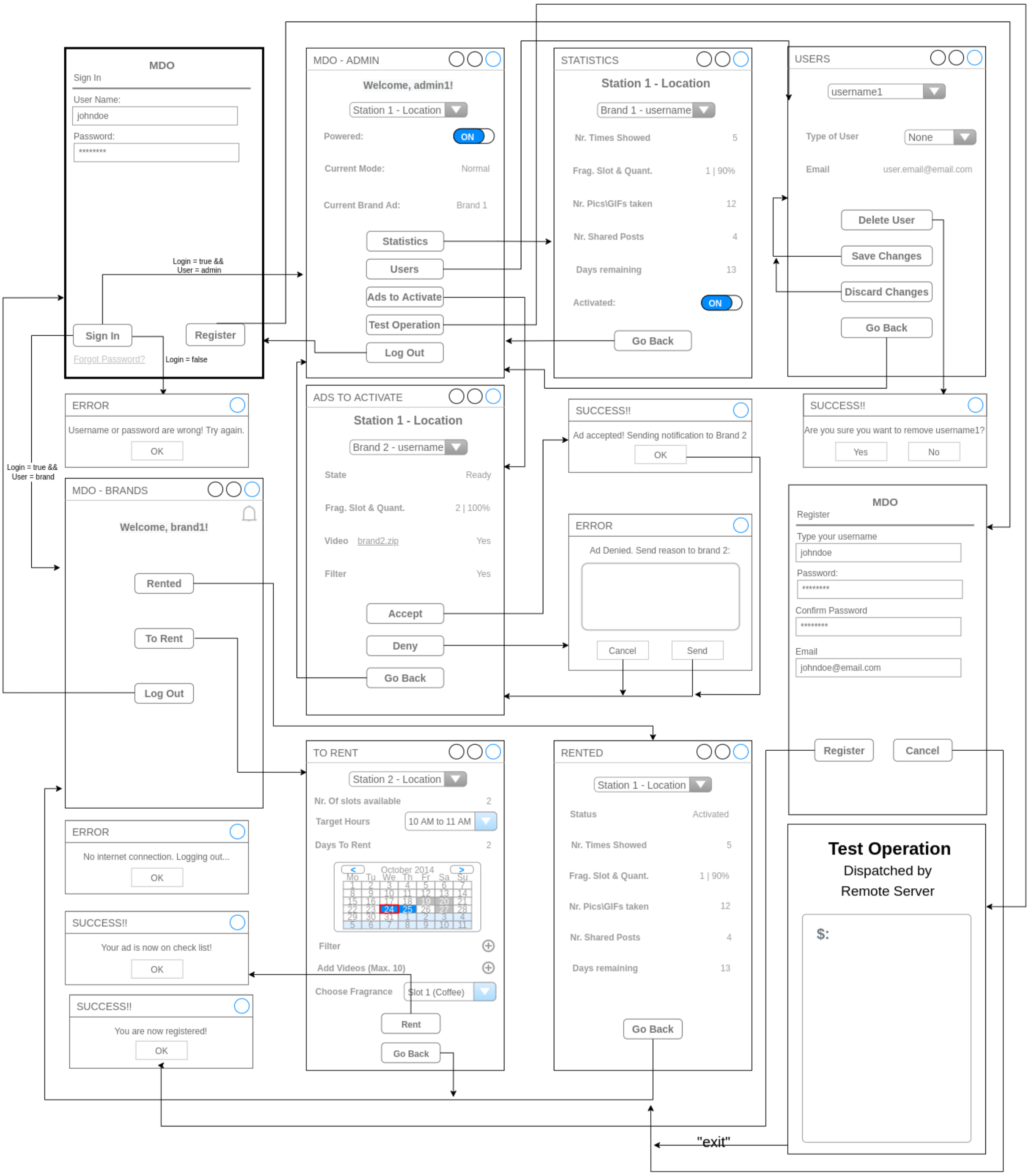


Figure 2.6.: User mockups: remote client

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

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

Secondly, the **Brand** workflow:

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

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

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

Table 2.1.: Events: remote client

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

Events

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

Use cases

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

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

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

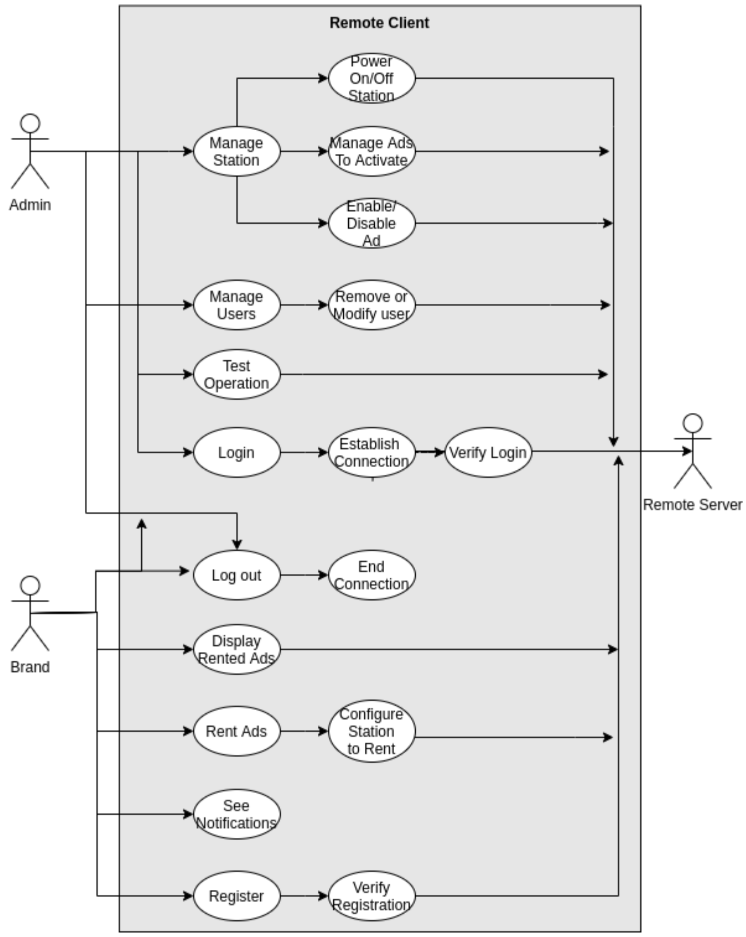


Figure 2.7.: Use cases: remote client

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

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

Dynamic operation

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

There are two main states:

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

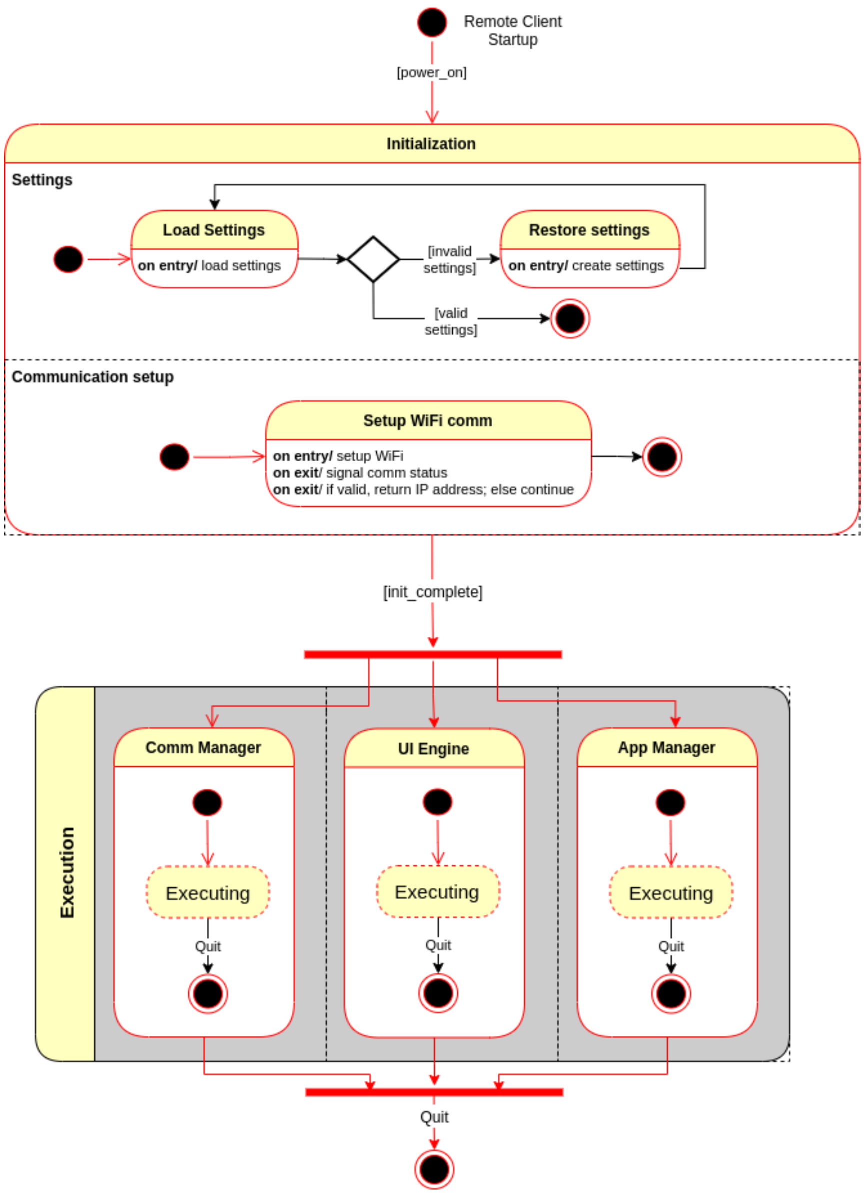


Figure 2.8.: State Machine Diagram: remote client

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

Communication Manager

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

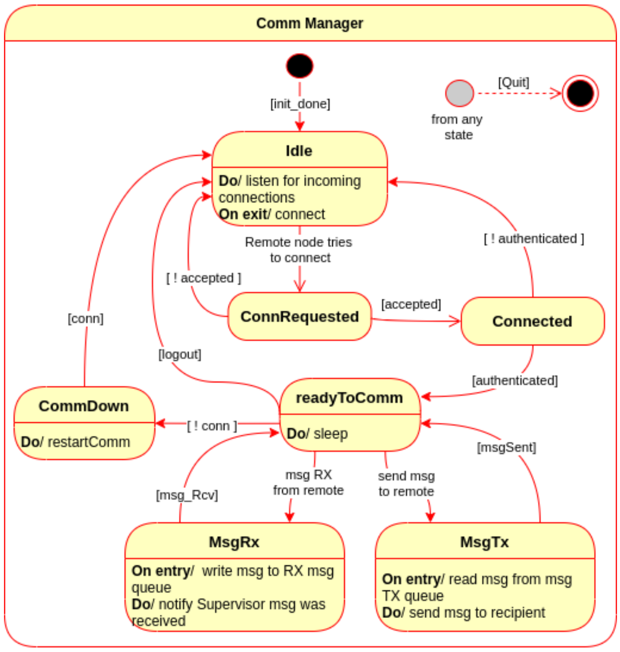


Figure 2.9.: State Machine Diagram: remote client — Communication Manager

App Manager

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

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

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

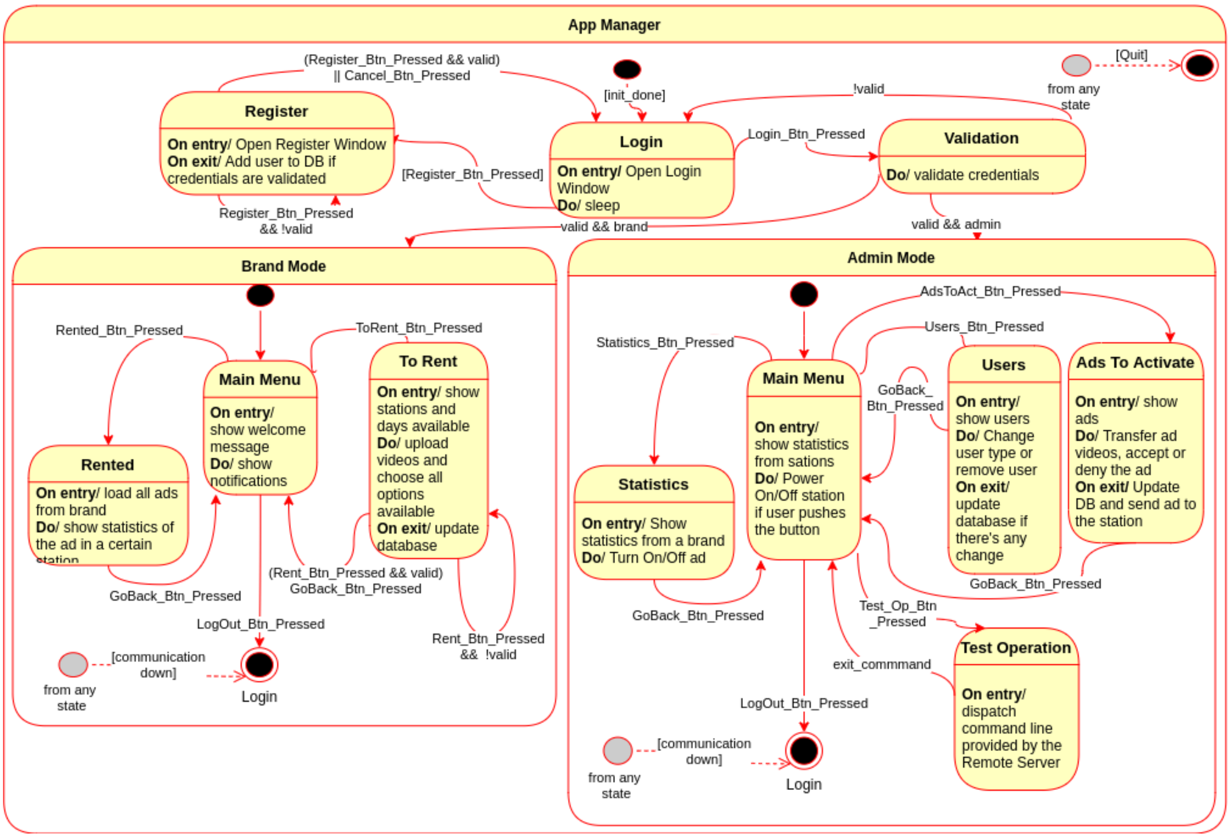


Figure 2.10.: State Machine Diagram: remote client — App Manager

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

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

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

Flow of events

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

«««< HEAD

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

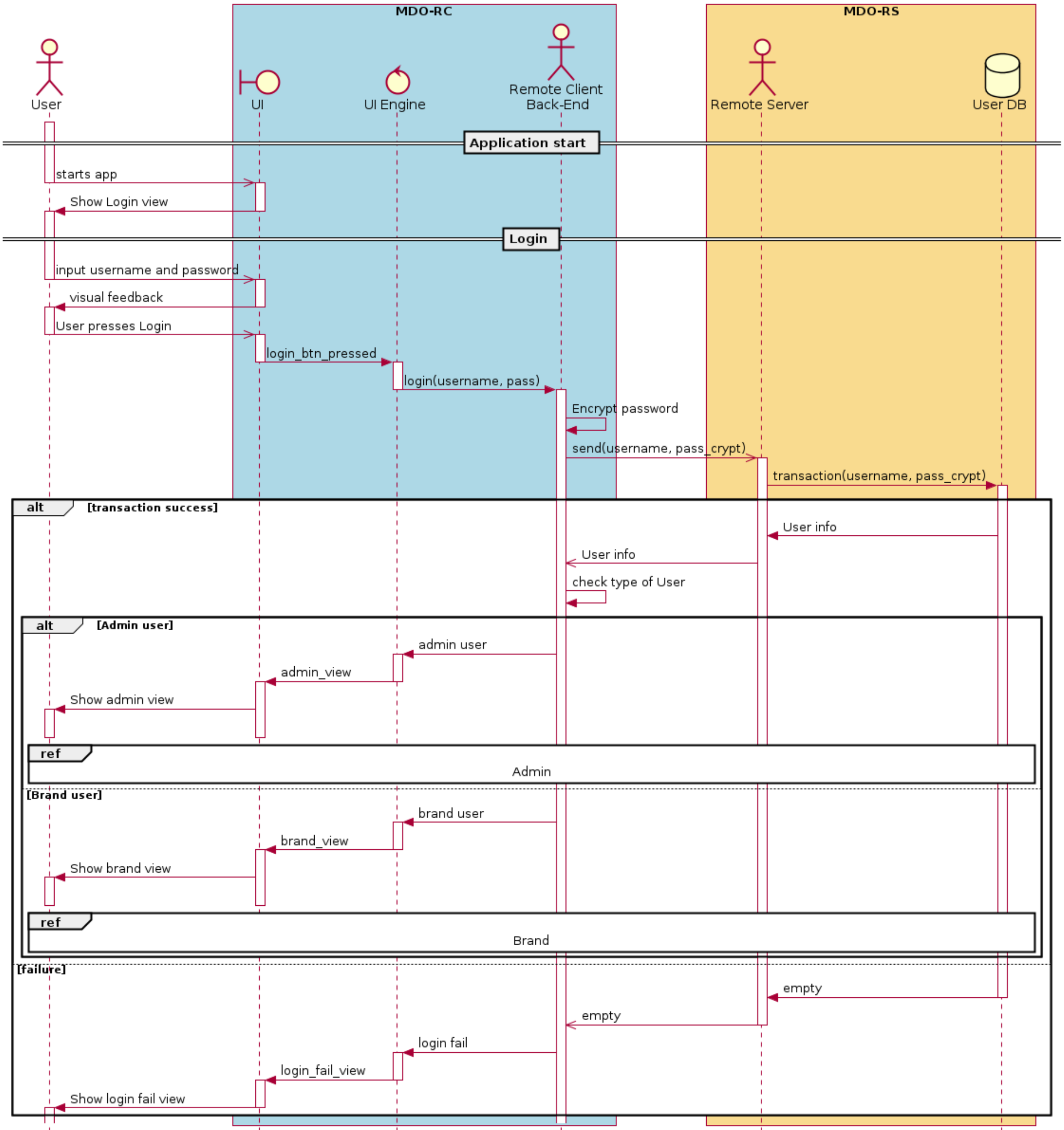


Figure 2.11.: Sequence Diagram: remote client — Login

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

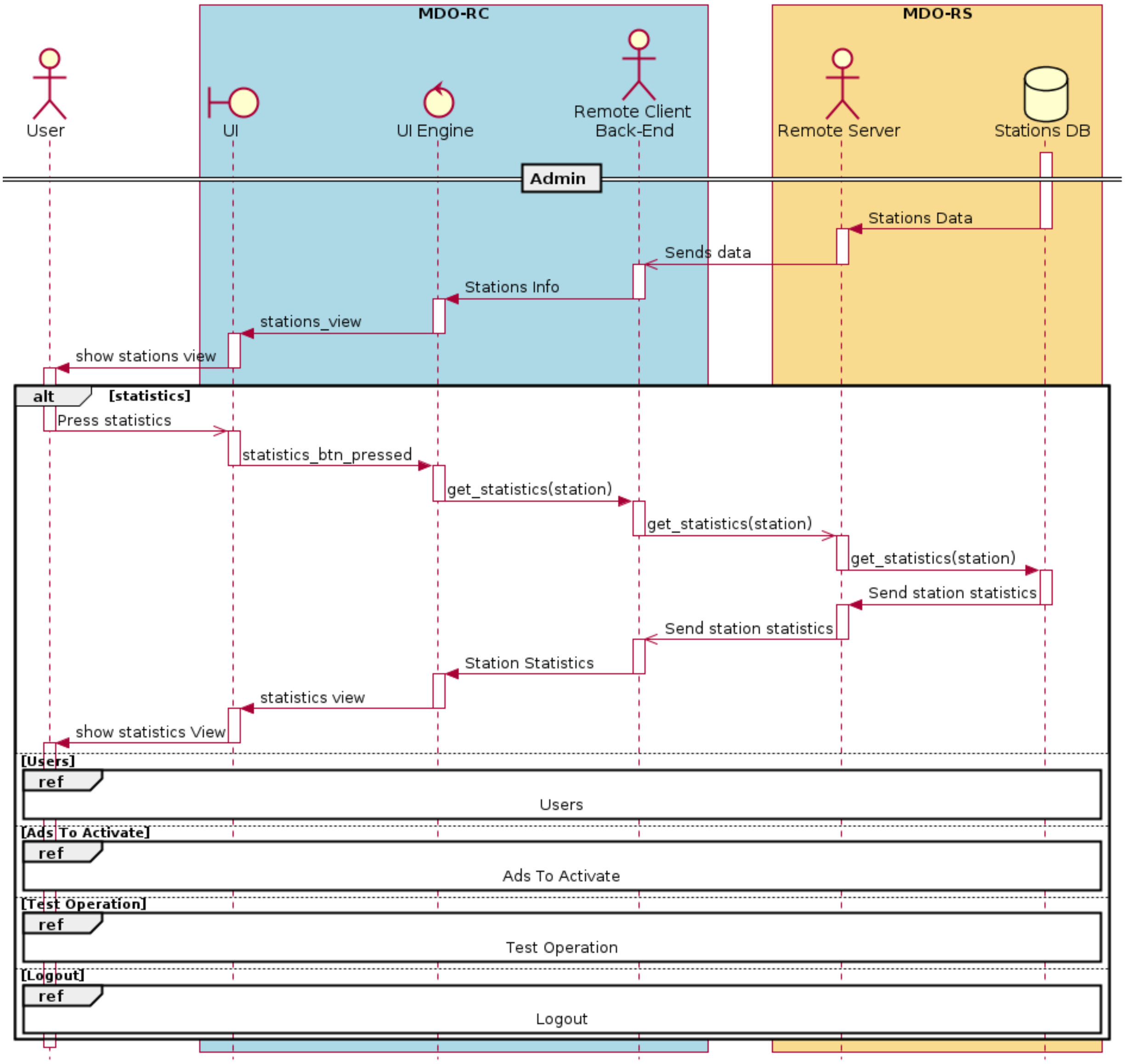


Figure 2.12.: Sequence Diagram: remote client — admin statistics

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

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

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

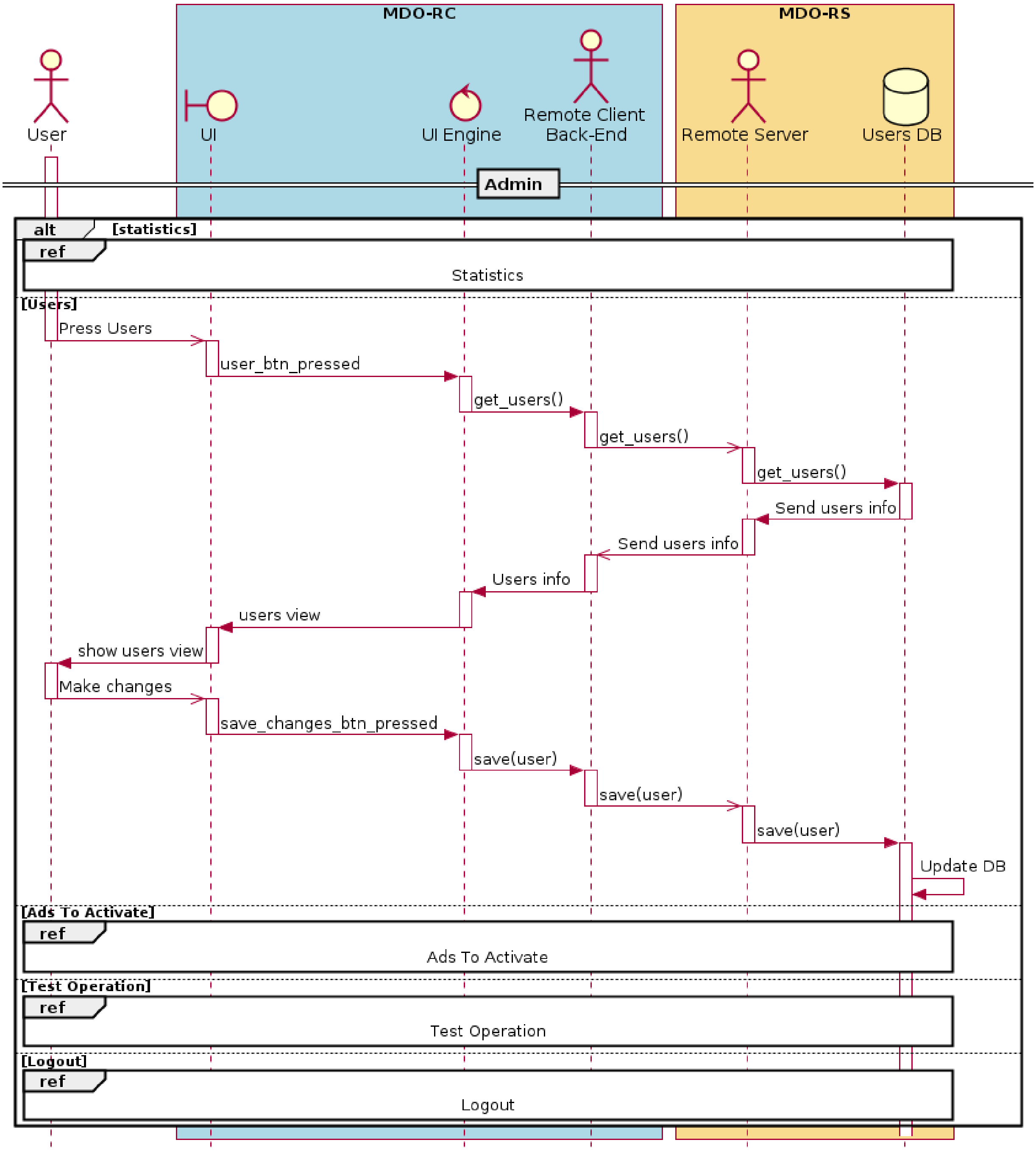


Figure 2.13.: Sequence Diagram: remote client — admin users

query or update them. »»»> 1b64a1fa2e419d79801eaaa806f4d9675f06902e

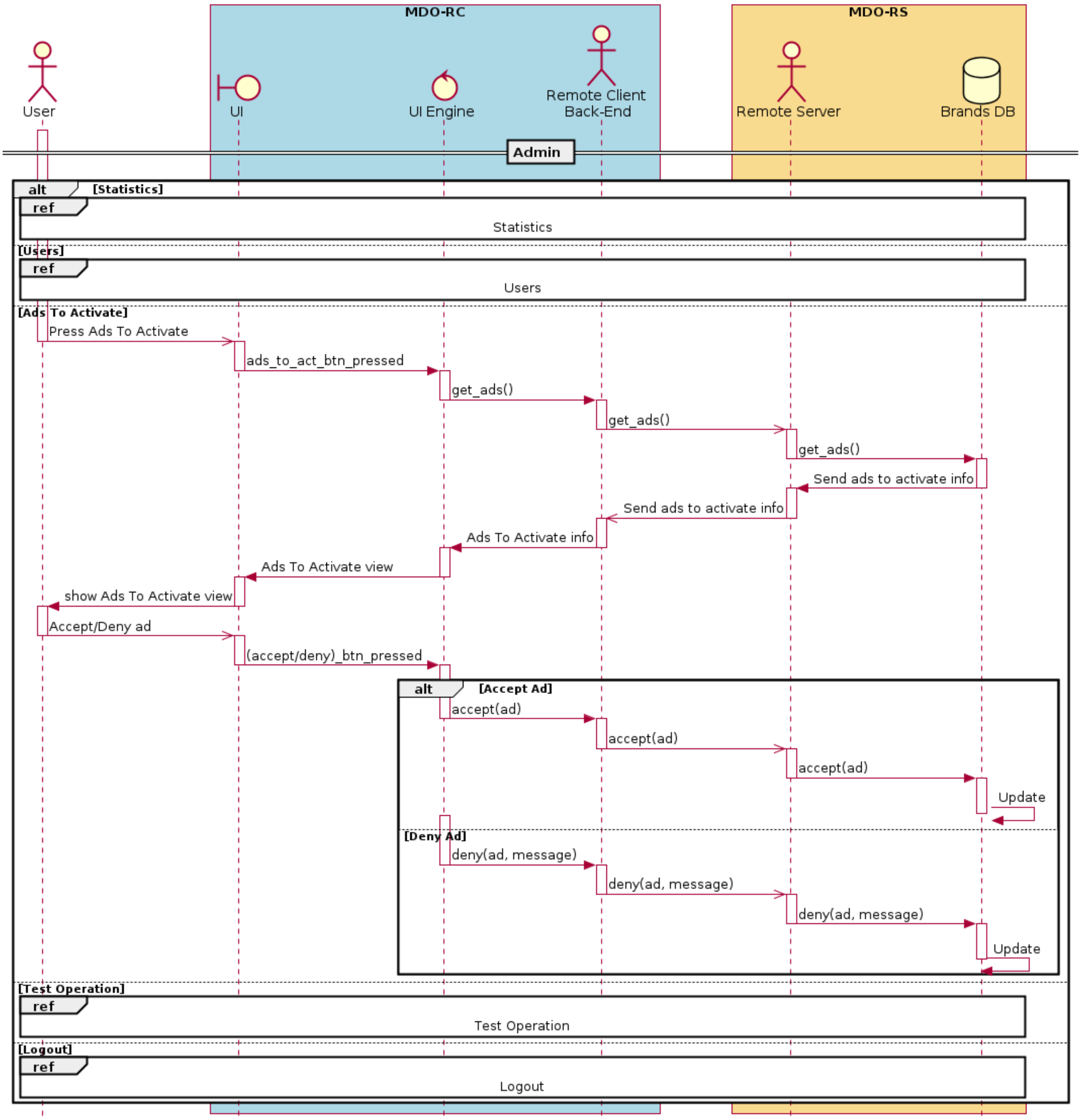


Figure 2.14.: Sequence Diagram: remote client — admin ads to activate

2.4.2. Remote server

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

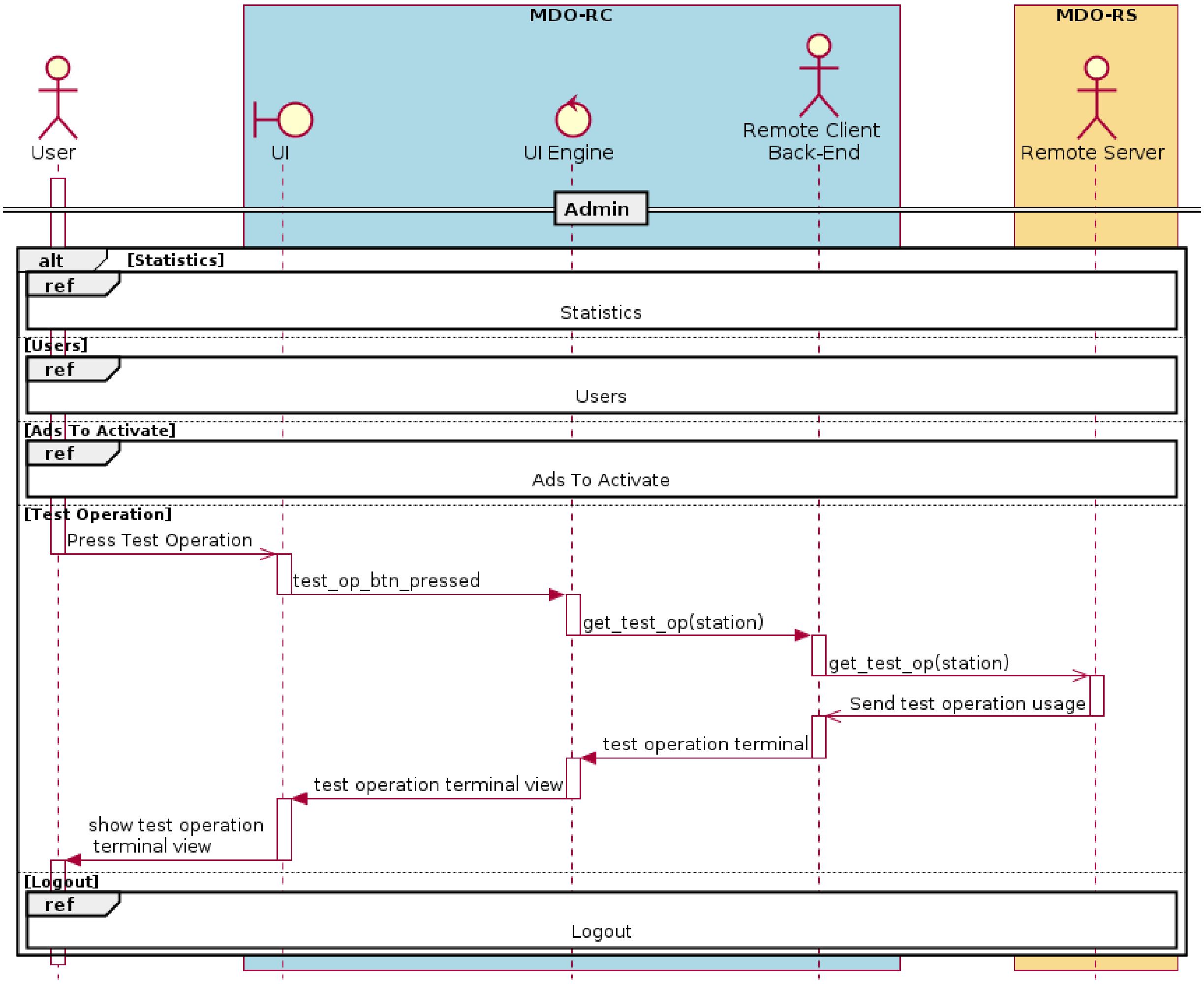


Figure 2.15.: Sequence Diagram: remote client — admin test operation

User mock-ups

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

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

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

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

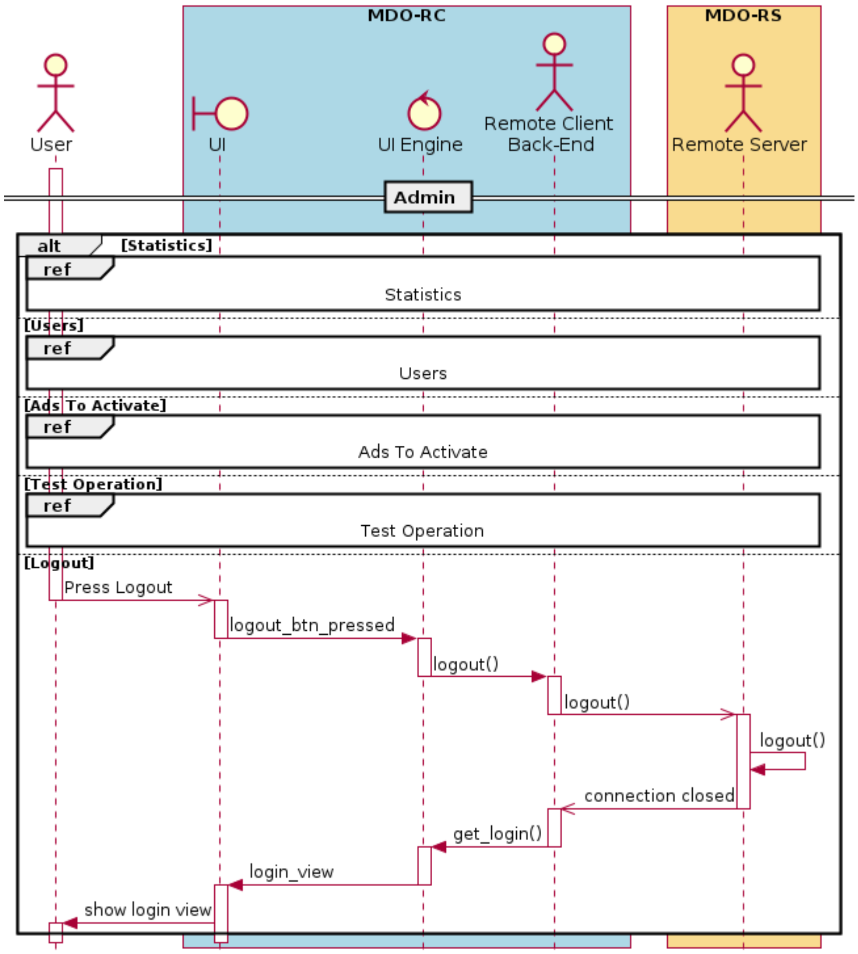


Figure 2.16.: Sequence Diagram: remote client — admin logout

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

Events

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

Use cases

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

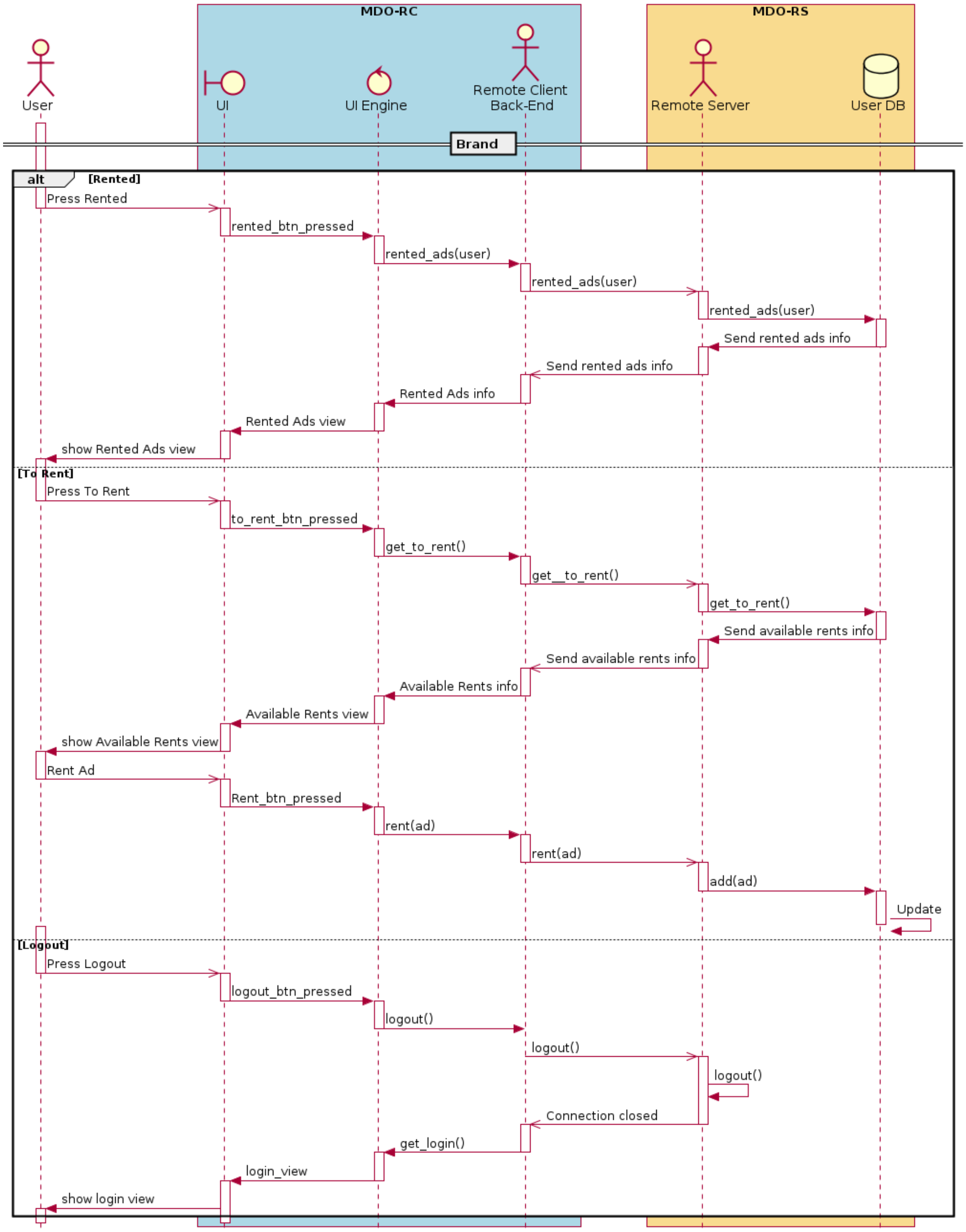


Figure 2.17.: Sequence Diagram: remote client - brand

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

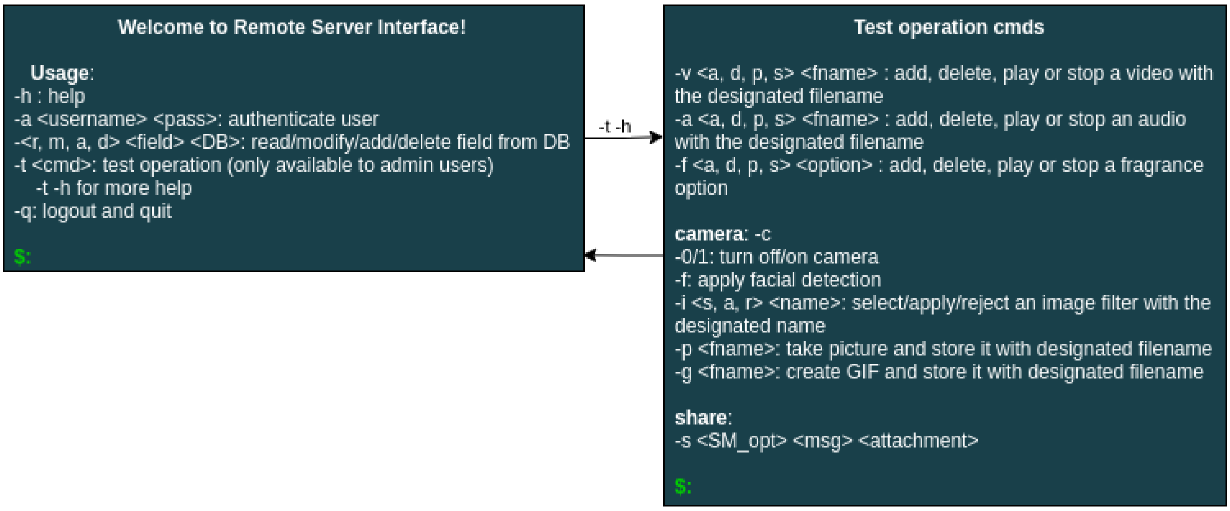


Figure 2.18.: User mock-ups: Remote Server

Table 2.2.: Events: Remote Server

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

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

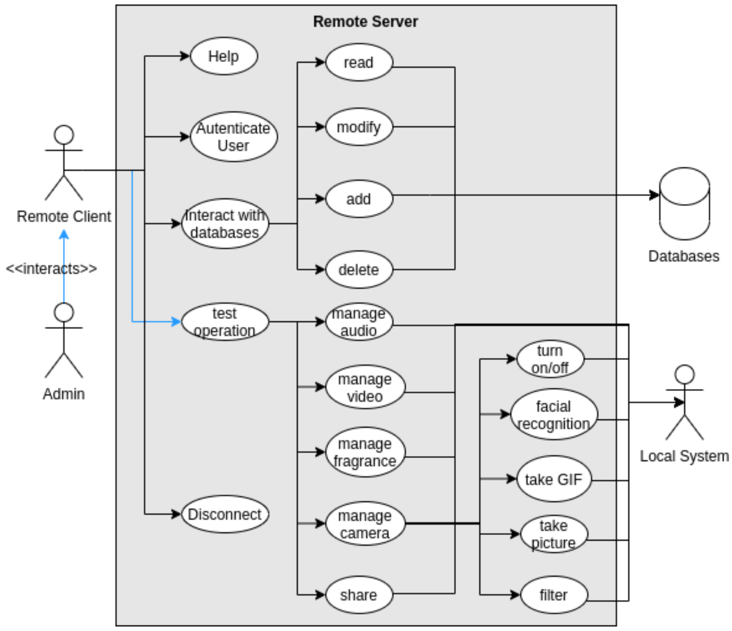


Figure 2.19.: Use cases: remote server

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

Dynamic operation

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

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

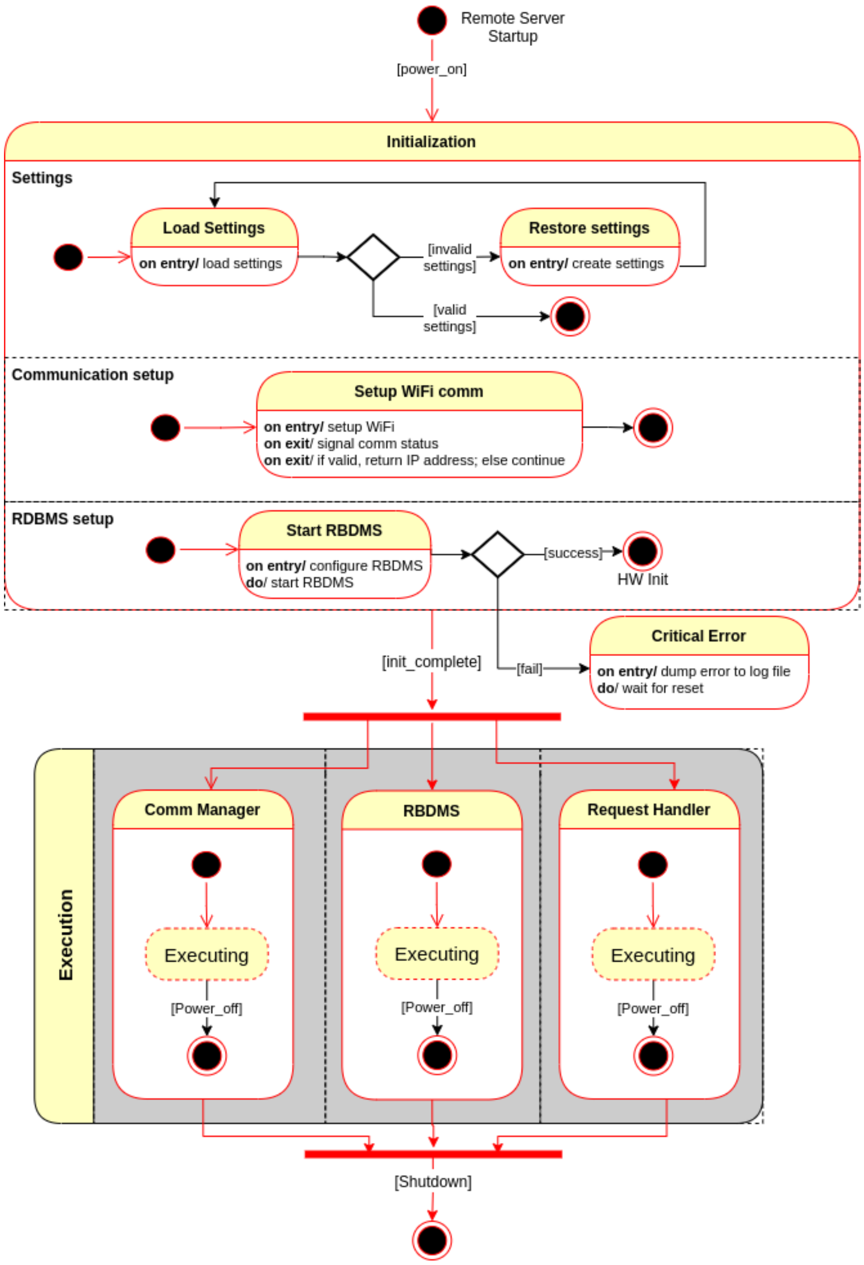


Figure 2.20.: State machine diagram: Remote server

Communication Manager

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

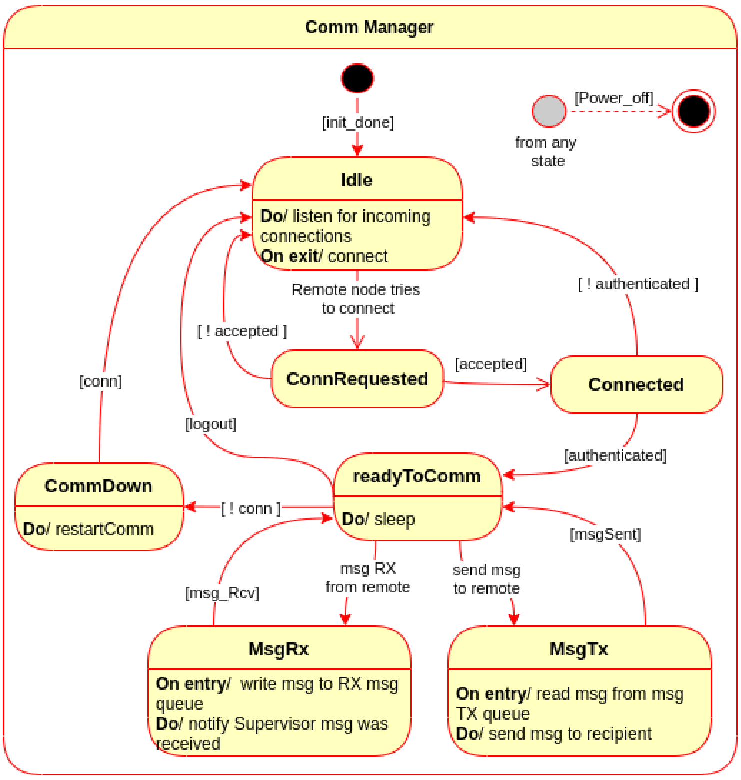


Figure 2.21.: State machine diagram: Remote Server — Comm Manager

Database Manager

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

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

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

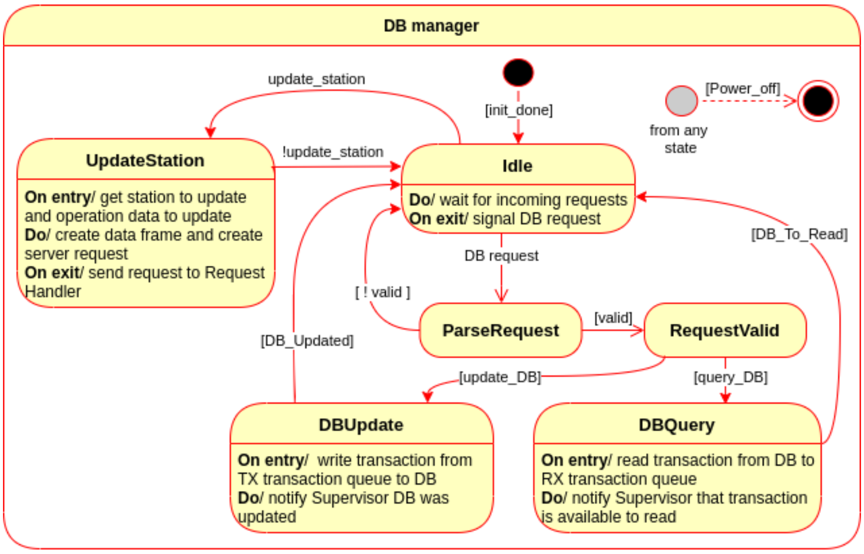


Figure 2.22.: State machine diagram: Remote Server — DB Manager

Request Handler

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

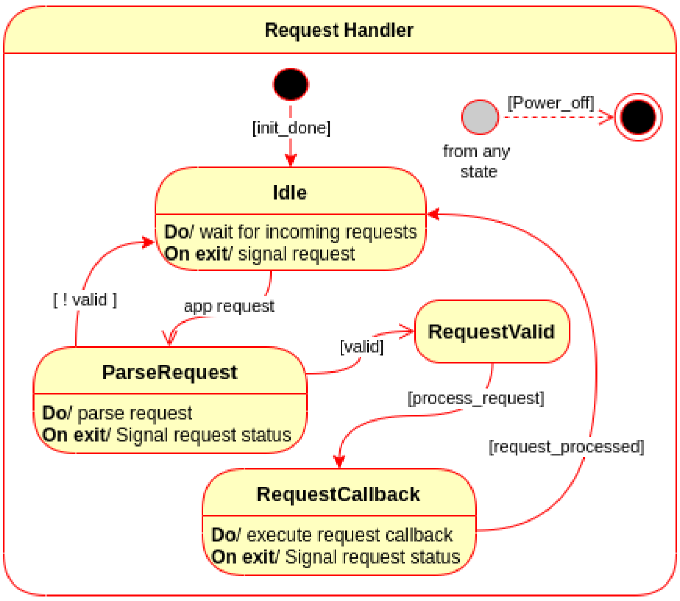


Figure 2.23.: State machine diagram: Remote Server — Request Handler

Flow of events

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

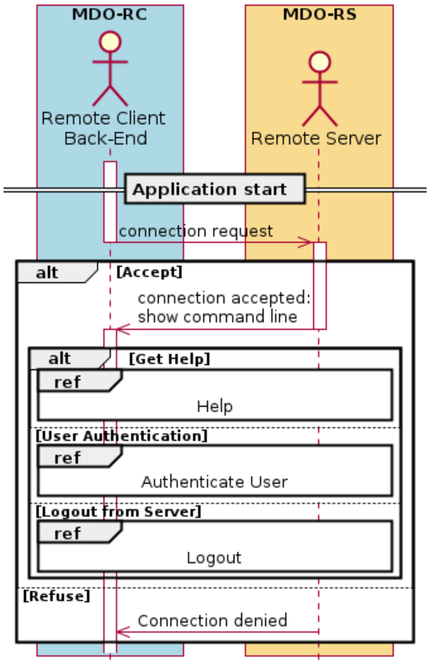


Figure 2.24.: Sequence diagram: Remote Server

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

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

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

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

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

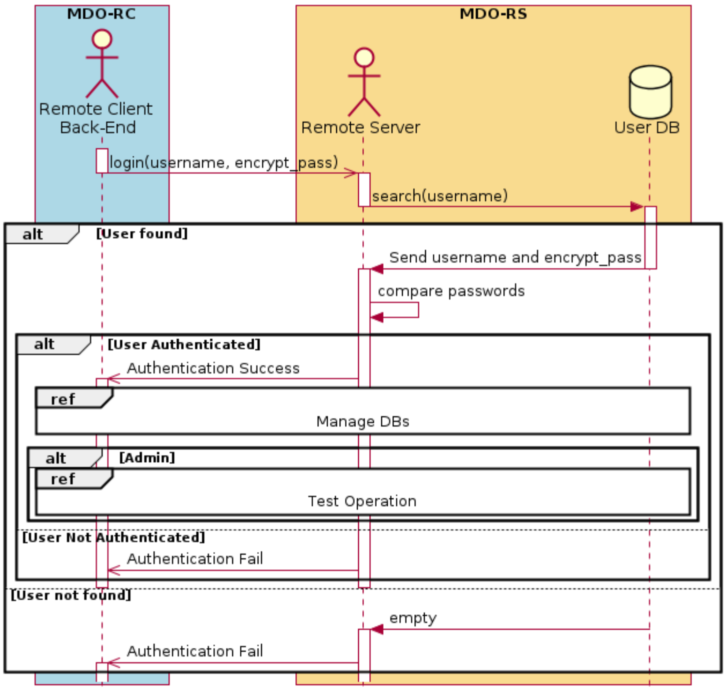


Figure 2.25.: Sequence diagram: Remote Server — Authentication

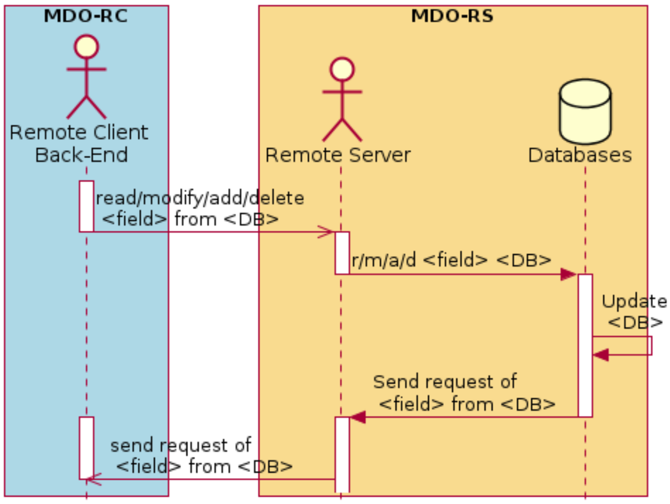


Figure 2.26.: Sequence diagram: Remote Server — Manage Databases

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

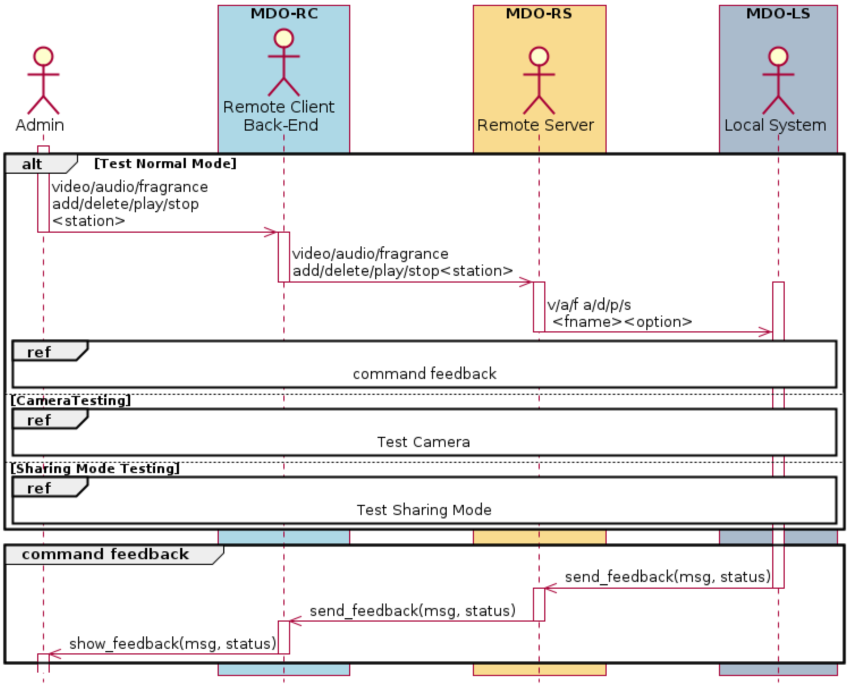


Figure 2.27.: Sequence diagram: Remote Server — Test Operation

2.4.3. Local system

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

User mock-ups

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

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

The workflow can be as follows:

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

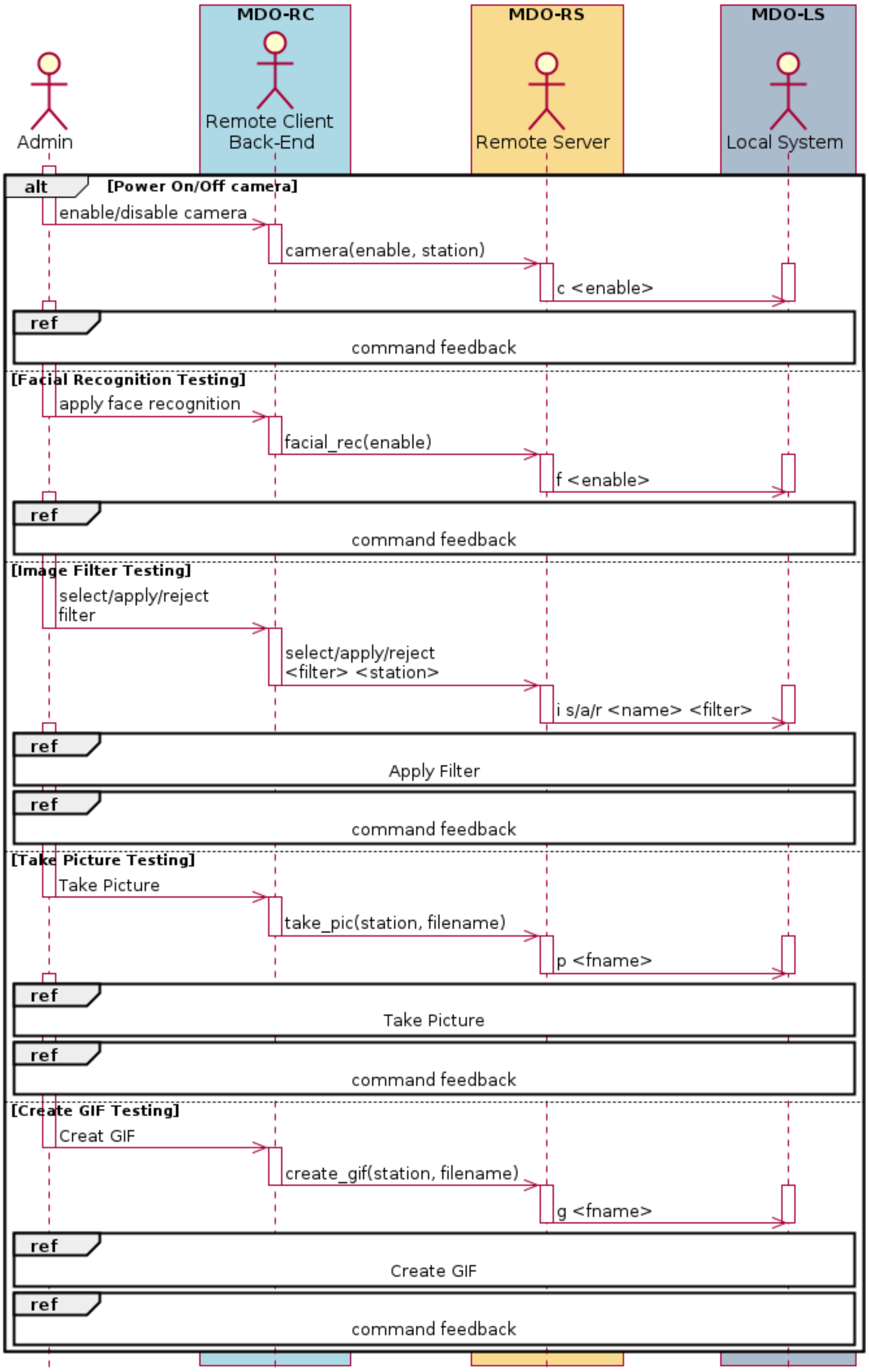


Figure 2.28.: Sequence diagram: Remote Server — Test Operation Camera

mode, keeping the filter on.

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

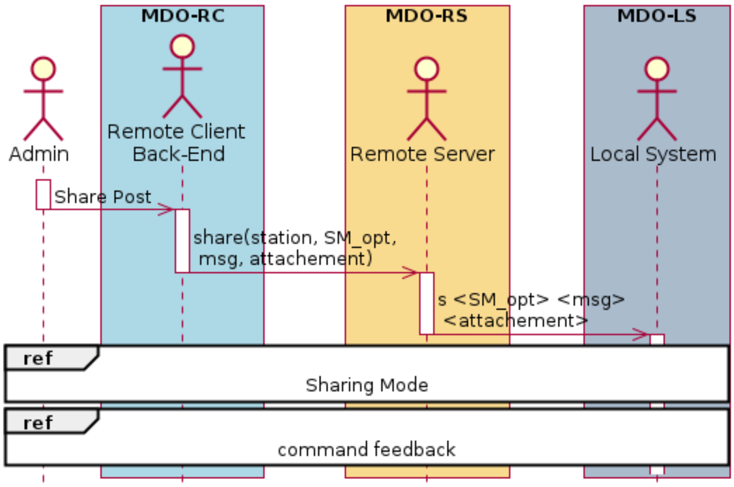


Figure 2.29.: Sequence diagram: Remote Server — Test Operation Share

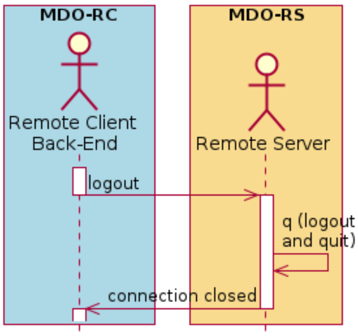


Figure 2.30.: Sequence diagram: Remote Server — logout

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

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

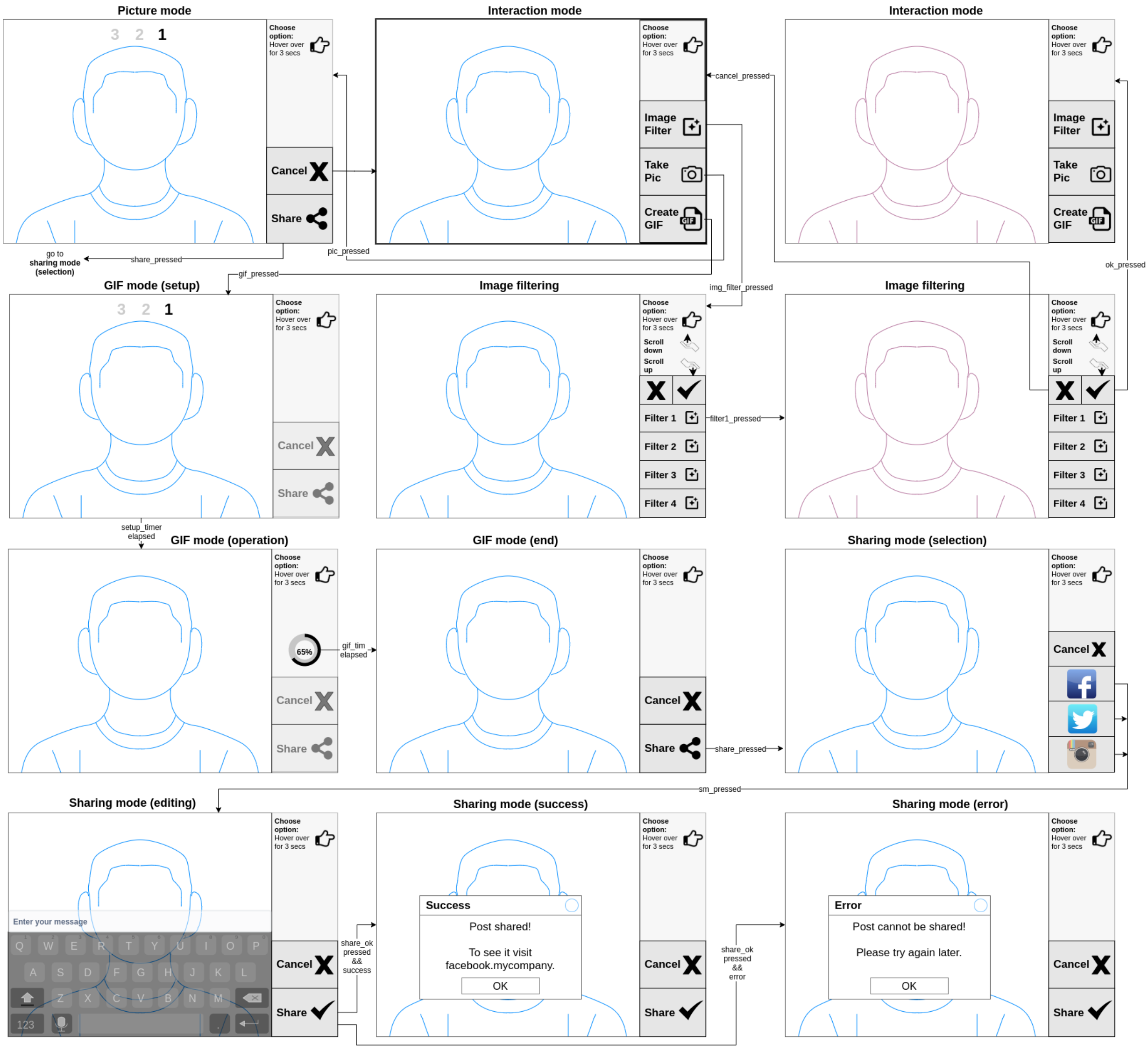


Figure 2.31.: User mock-ups: local system

Events

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

Table 2.3.: Events: local system

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

Use cases

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

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

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

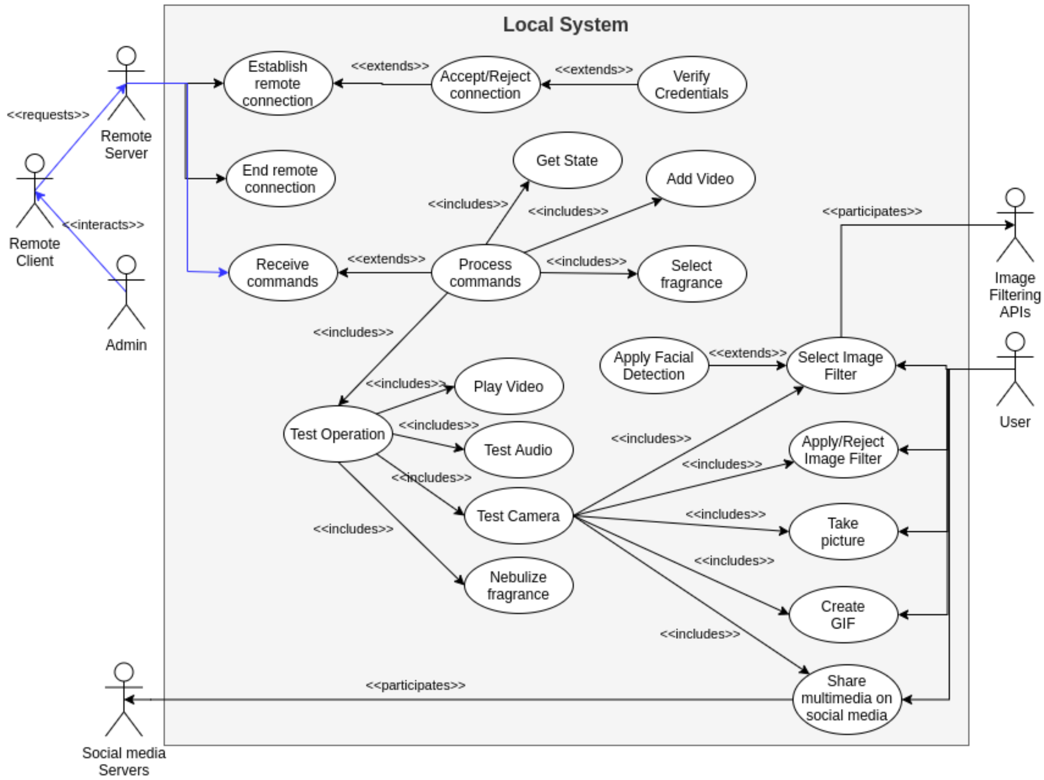


Figure 2.32.: Use cases diagram: local system

Dynamic operation

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

There are two main states:

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

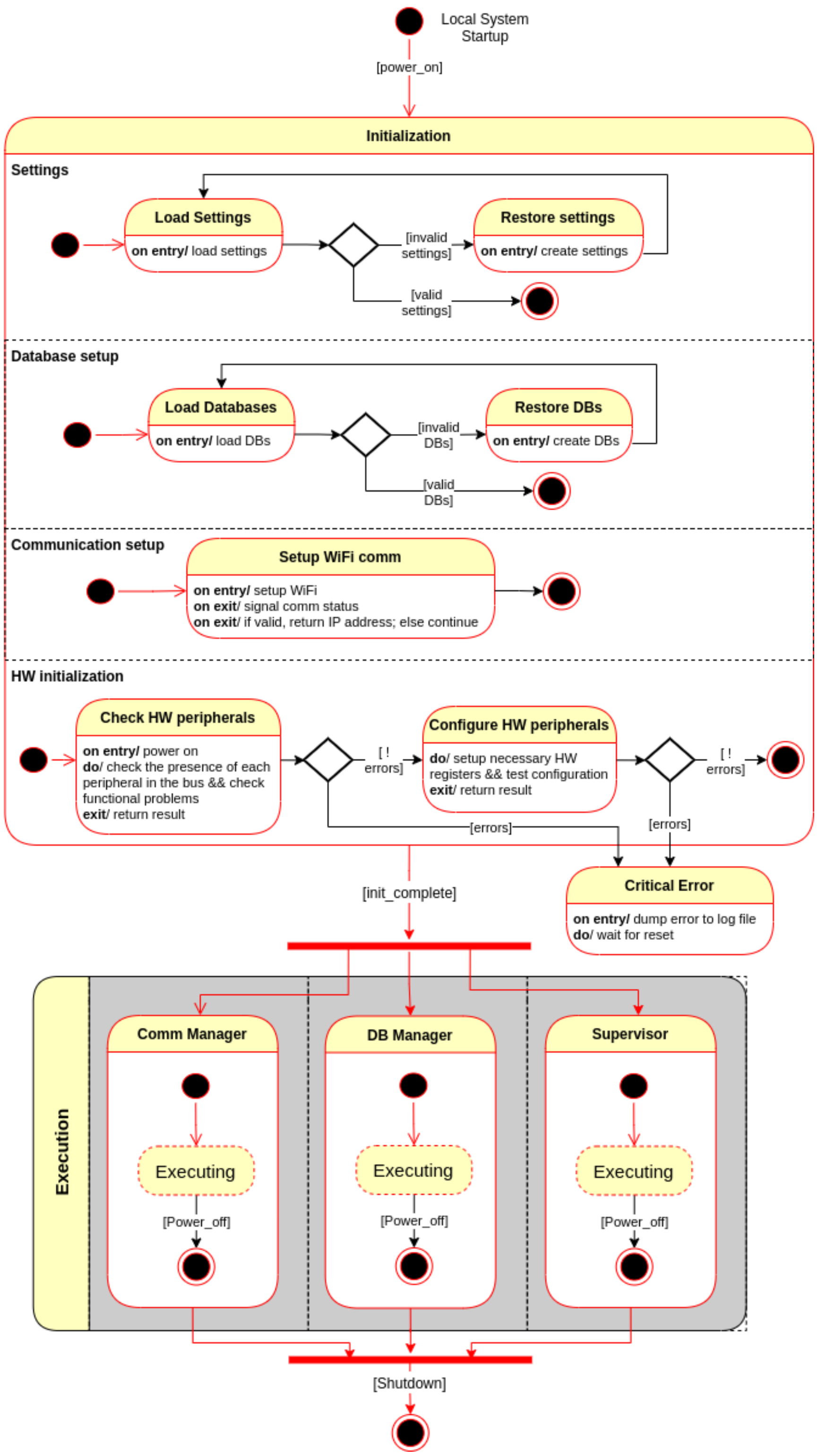


Figure 2.33.: State machine diagram: local system

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

Communication Manager

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

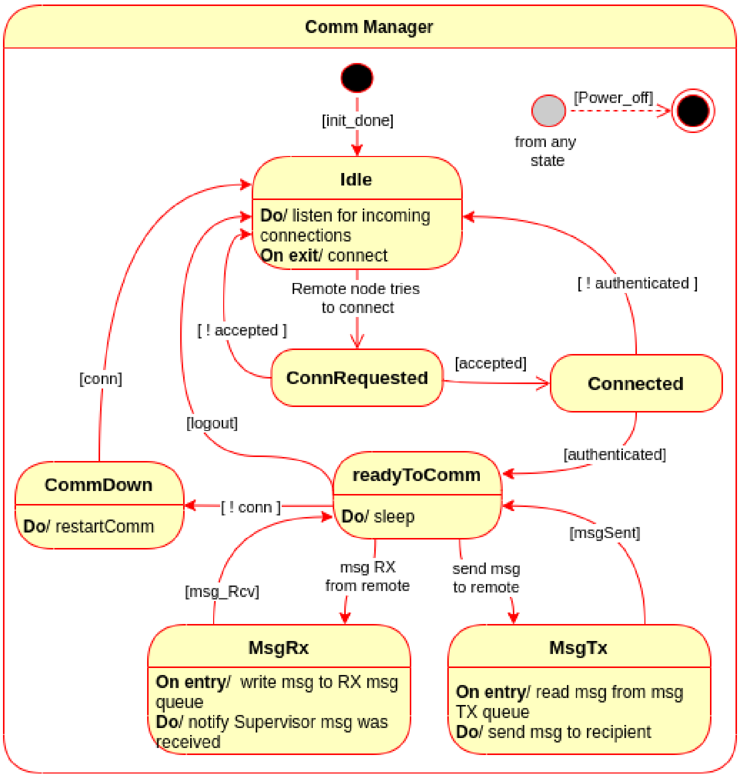


Figure 2.34.: State machine diagram: local system — Comm Manager

Database Manager

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

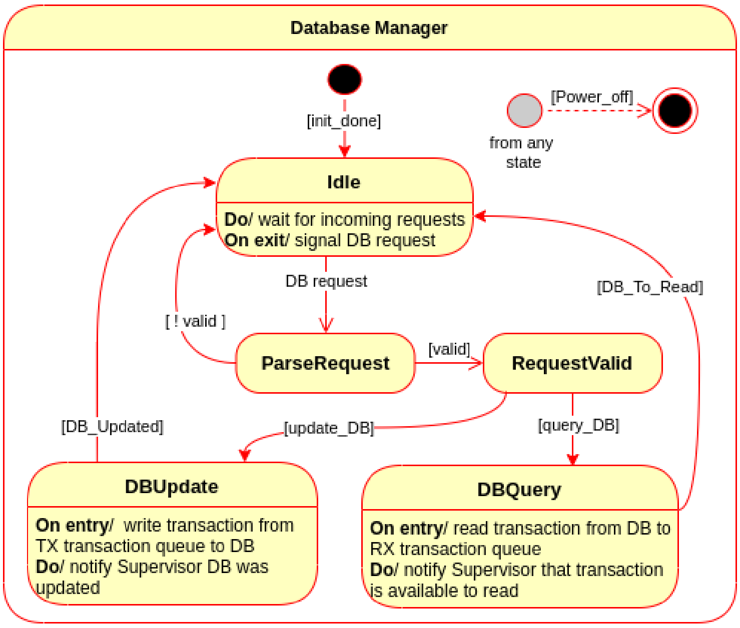


Figure 2.35.: State machine diagram: local system — DB Manager

Supervisor

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

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

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

Flow of events

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

Normal mode

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

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

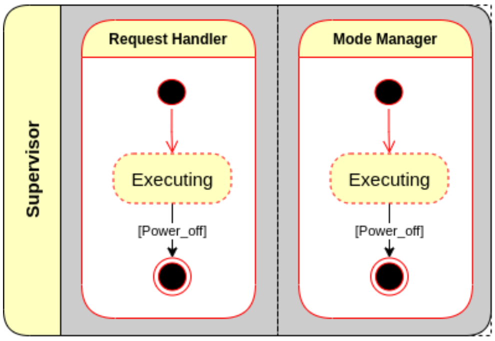
- Video playback: while its time to play the advertisements, a video is played from the video list. When it finishes, it moves the next video in the playback queue.
- Fragrance diffusion: while there are timestamps for fragrance diffusion, diffuse fragrance between start and stop times and sleep on the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

Interaction mode

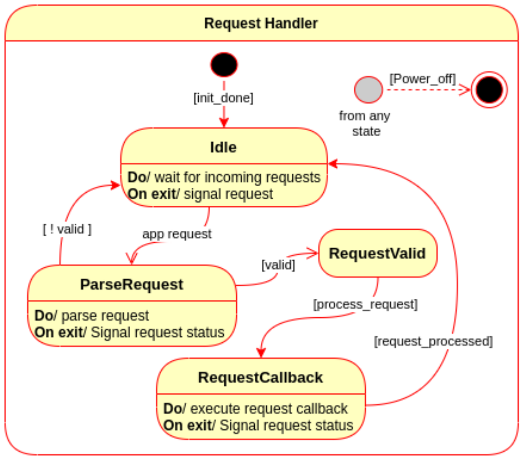
Fig. 2.38 depicts the **Interaction mode**'s sequence diagram. The blue area delimits the **MDO-L** system, comprising the **Gesture Recognition Engine**, the **UI engine** and the **Local System Back-End**.

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

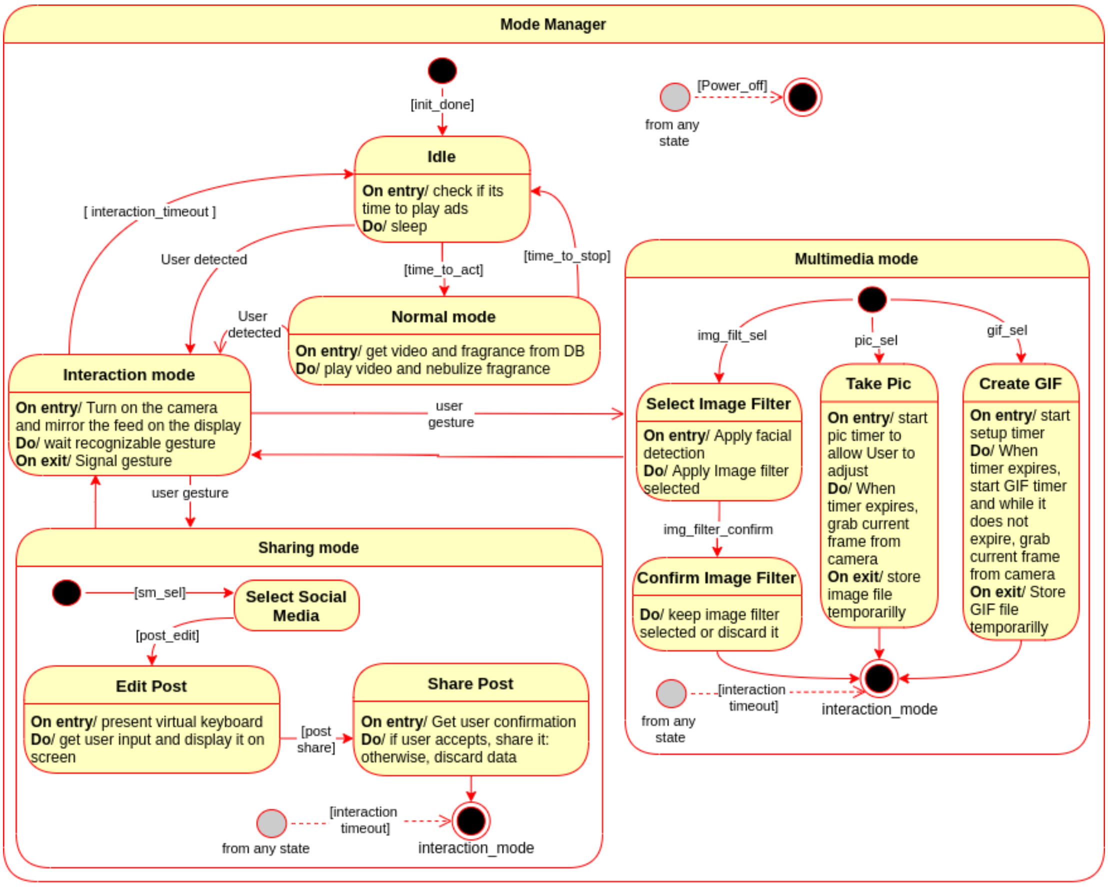
- mirror camera feed: while the **User** is in range and active, the **UI engine** will grab frame from the camera and display it on the window providing visual feedback to the **User**.
- gesture recognition and processing: if a gesture is recognized by the **Gesture Recognition Engine** it is dispatched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



(a) main



(b) Request Handler



(c) Mode Manager

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

Multimedia mode

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

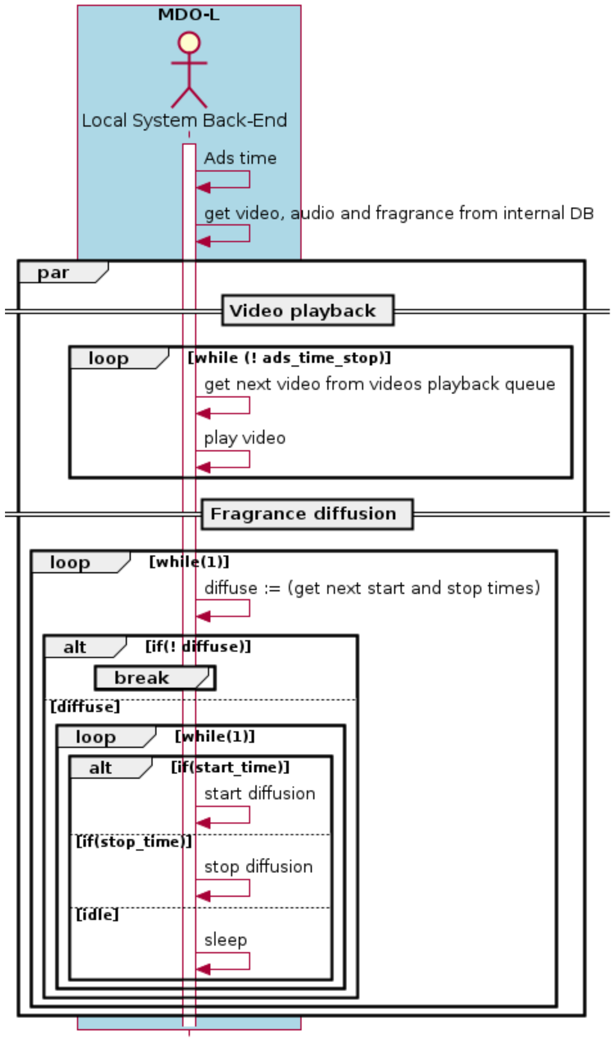


Figure 2.37.: Sequence diagram: local system — Normal mode

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

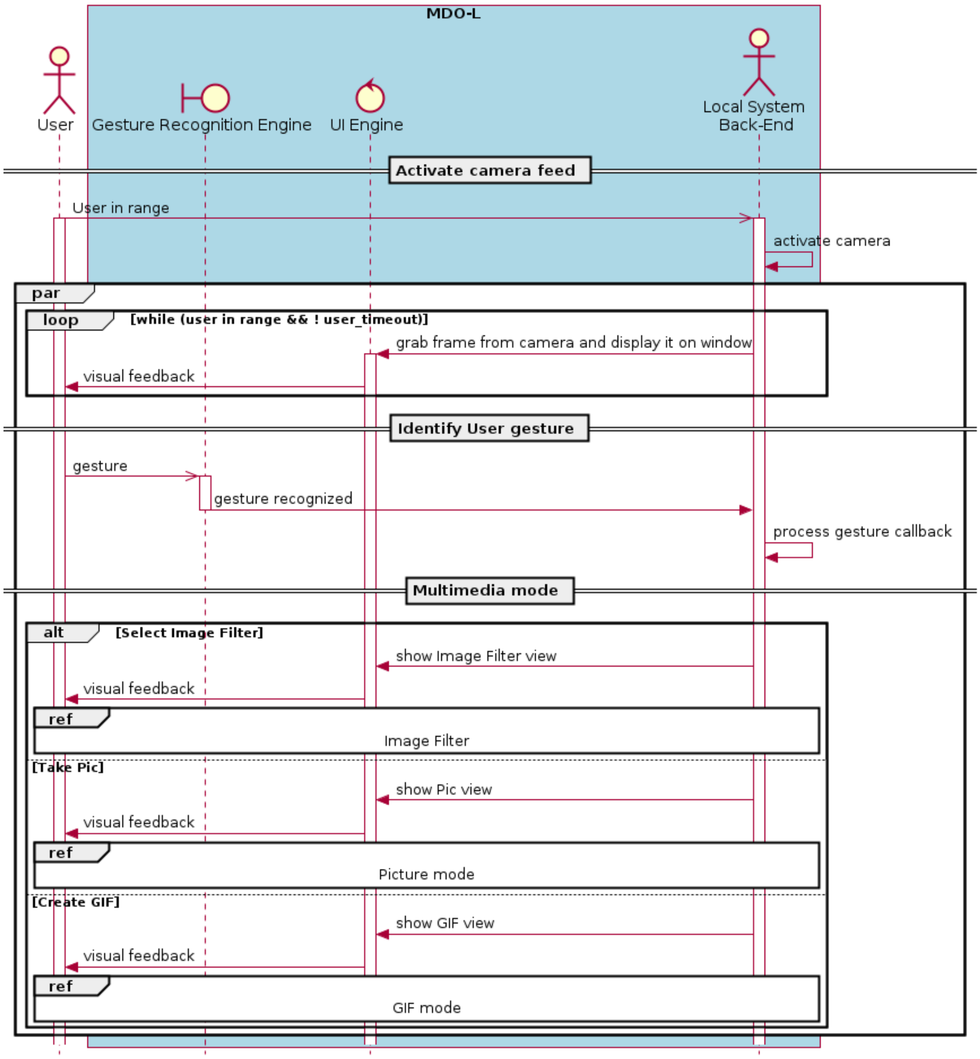


Figure 2.38.: Sequence diagram: local system — Interaction mode

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

Sharing mode

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

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

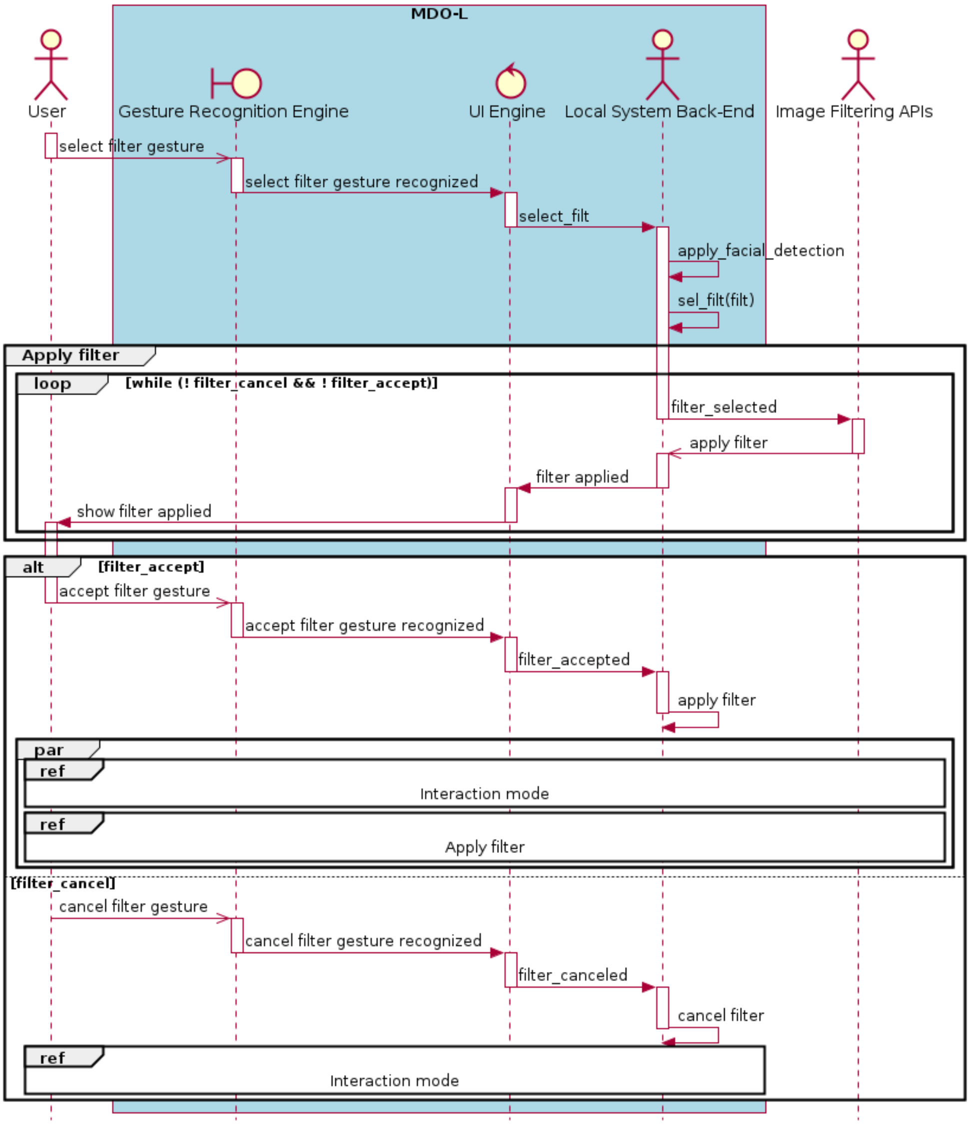


Figure 2.39.: Sequence diagram: local system — Multimedia mode (select image filter)

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

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

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

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

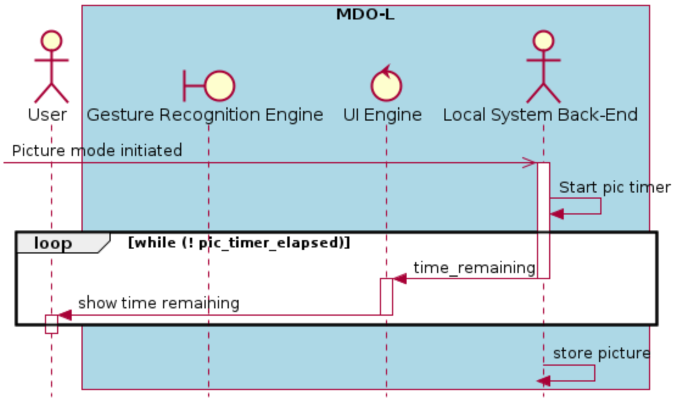


Figure 2.40.: Sequence diagram: local system — Multimedia mode (take picture)

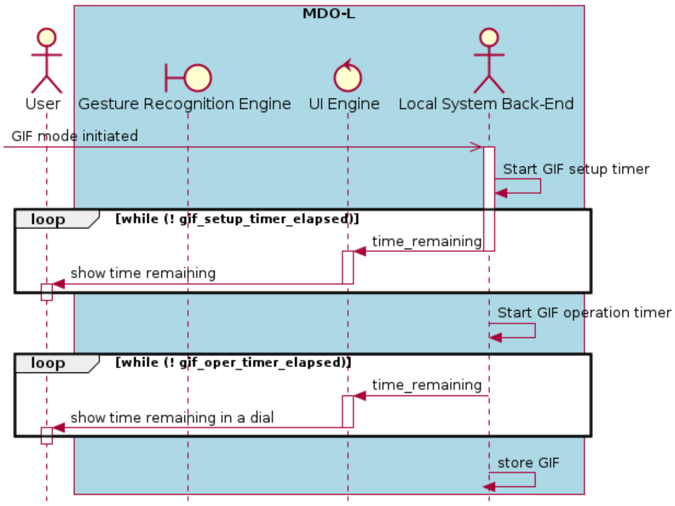


Figure 2.41.: Sequence diagram: local system — Multimedia mode (create GIF)

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

2.5. Budget estimation

Table 2.4 presents the budget estimation for the project.

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

Additionally, several types of costs were assessed, namely:

Table 2.4.: Budget estimation

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

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

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

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

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

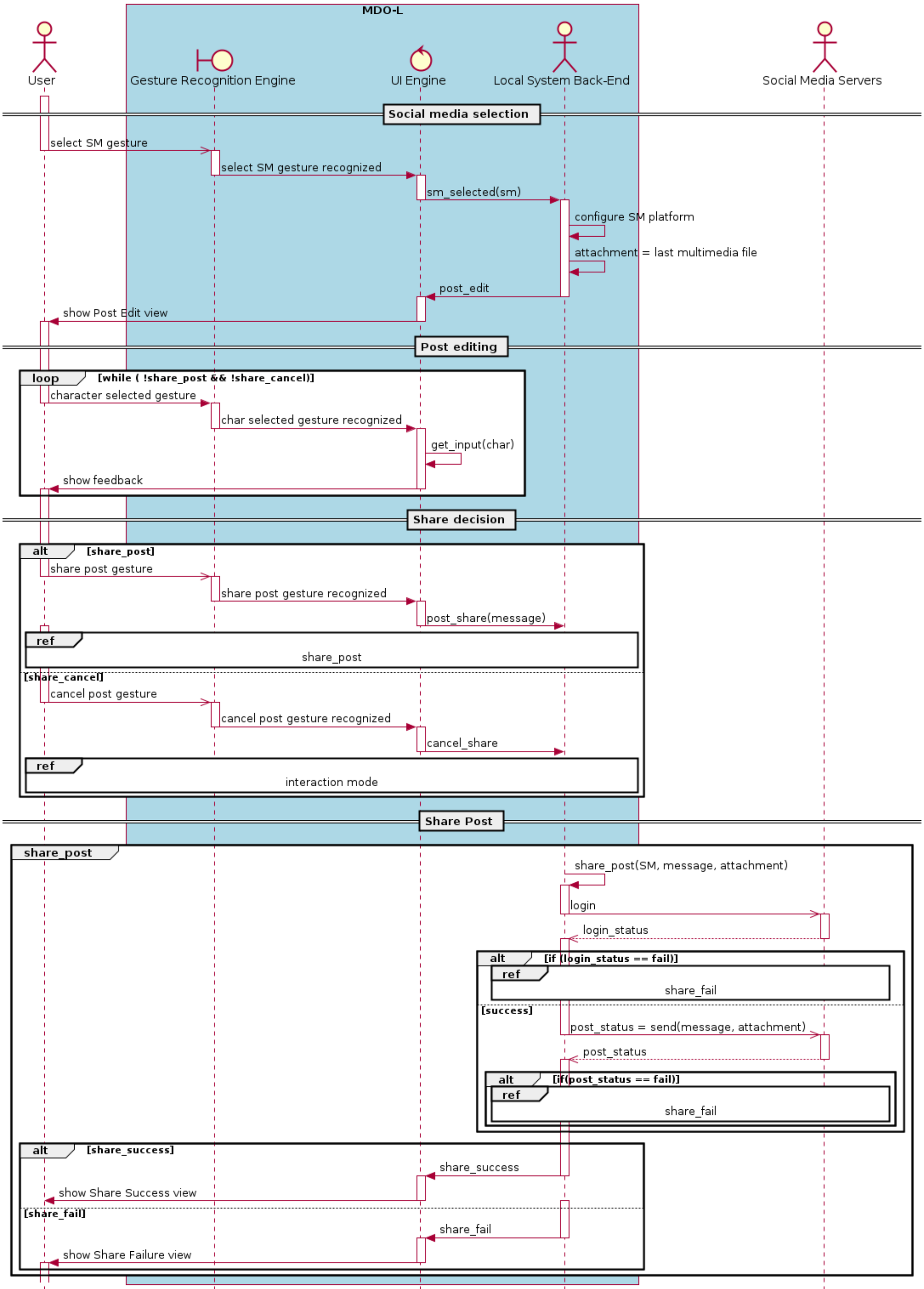


Figure 2.42.: Sequence diagram: local system — Sharing mode

3. Theoretical foundations

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

3.1. Project methodology

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

3.1.1. Waterfall model

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

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

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

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

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

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

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

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

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

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

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

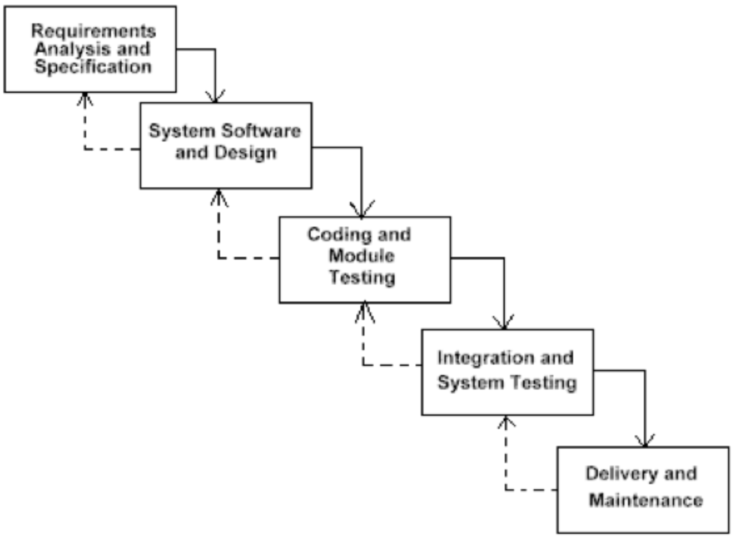


Figure 3.1.: Waterfall model diagram

3.1.2. Unified Modeling Language (UML)

To aid the software development process, a notation is required, to articulate complex ideas succinctly and precisely. The notation chosen was the Unified Modeling Language (UML), as it provides a spectrum of notations for representing different aspects of a system and has been accepted as a standard notation in the software industry [11].

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

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

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

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

3.2. Concurrency

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

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

Every concurrent system provides three important facilities [12]:

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

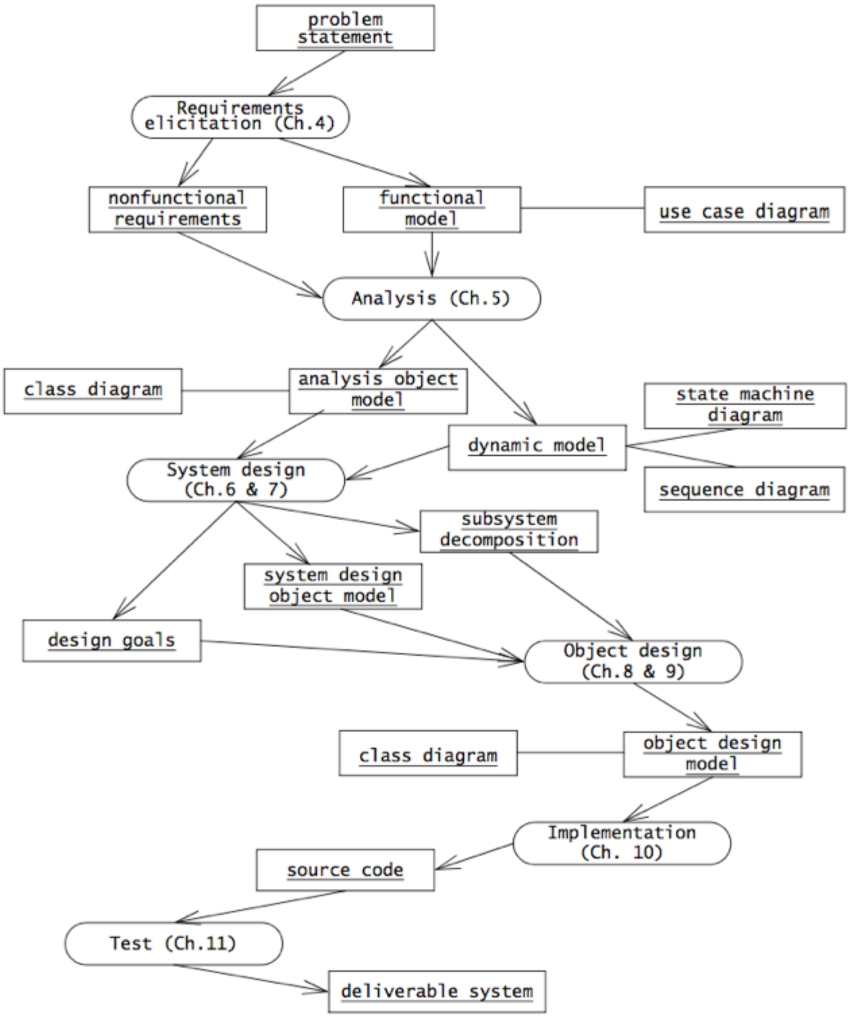


Figure 3.2.: An overview of the object-oriented software engineering development and their products. This diagram depicts only logical dependencies among work products (withdrawn from [11])

3.3. Threads versus Processes

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

The advantages of a multithreaded approach are [13]:

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

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

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

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

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

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

3.3.1. Pthreads API

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

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

Thread creation

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

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

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

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

Thread termination

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

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

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

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

Joining with a terminated thread

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

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

It is important to note that:

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

Detaching a thread

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

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

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

3.4. Communications

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

3.4.1. IEEE 802.11 — Wi-Fi

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

TCP/IP

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

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

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

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

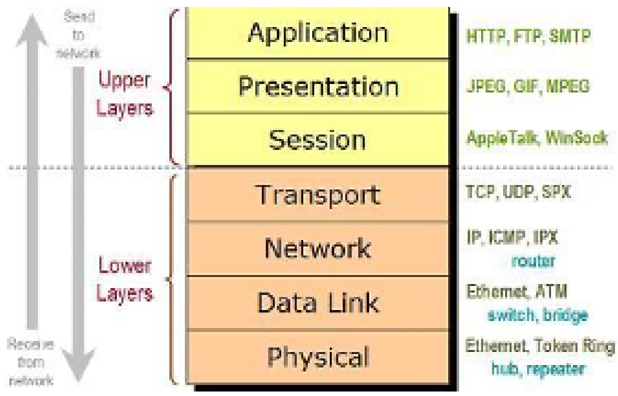


Figure 3.3.: OSI model

3.4.2. Network programming — sockets

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

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

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

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

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

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

3.4.3. Client/server model

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

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

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

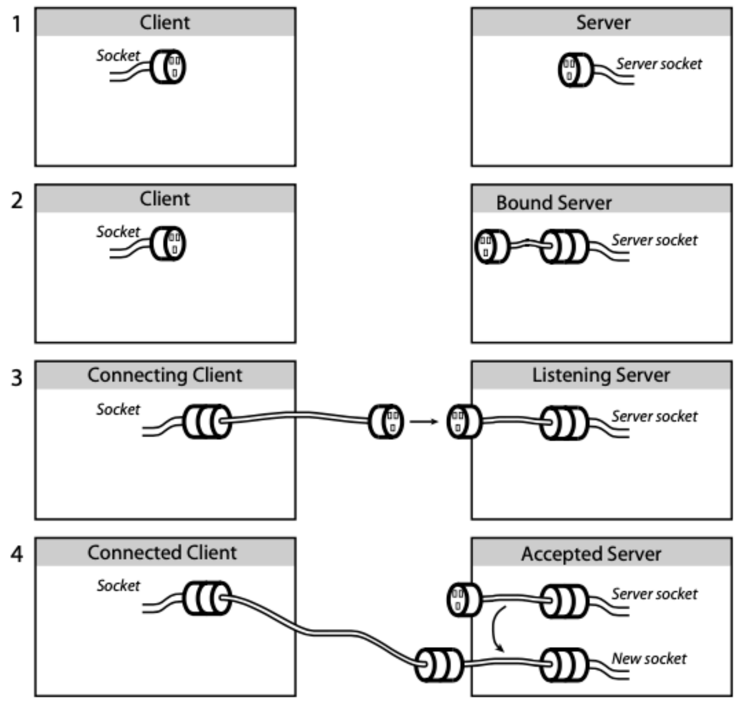


Figure 3.4.: Steps to obtain a connected socket (withdrawn from [16])

- Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place. Two sockets are connected as follows:

  1. One application, assuming the role of server, calls **bind()** to bind the socket to a well-known address, and then calls **listen()** to notify the kernel it is ready to accept incoming connections.
  2. The other application, assuming the role of client, establishes the connection by calling **connect()**, specifying the address of the socket to which the connection is to be made.
  3. The server then accepts the connection using **accept()**. If the **accept()** is performed before the client application calls **connect()**, then the **accept()** blocks.

- Once a connection has been established, data can be transmitted in both directions between the applications (analogous to a bidirectional telephone conversation) until one of them closes the connection using **close()**.

- Communication is performed using the conventional **read()** and **write()** system calls or via a number of socket-specific system calls (such as **send()** and **recv()**) that provide additional functionality. By default, these system calls block if the Input/Output (I/O) operation can't be completed immediately. However, nonblocking I/O is also possible.

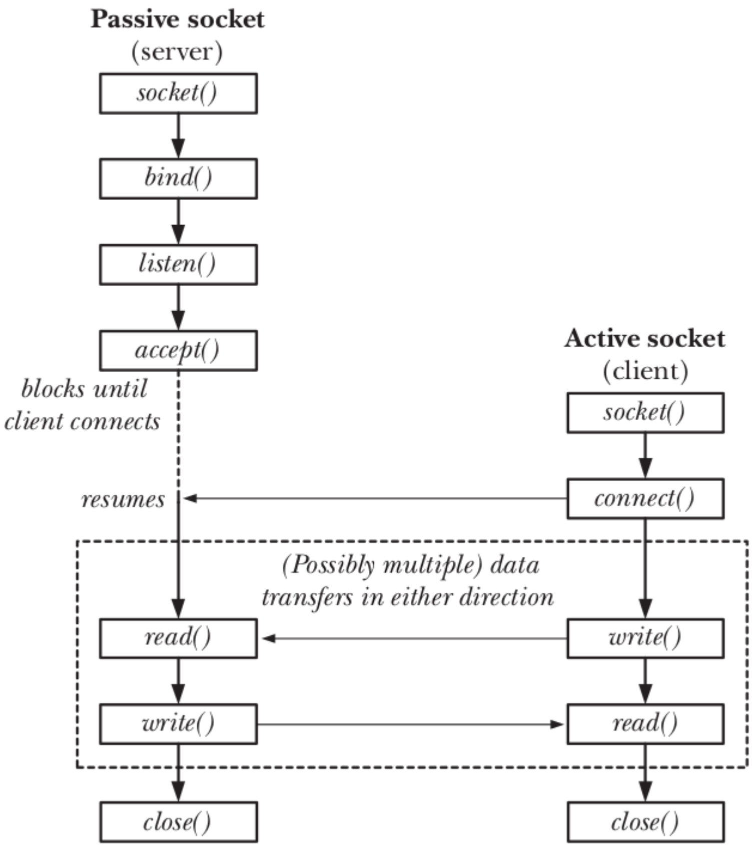


Figure 3.5.: Overview of UNIX system calls with sockets implementing a server/client paradigm (withdrawn from [13])

3.5. Daemons

aaaaaaaa

3.6. Device drivers

aaaaaaaa

3.7. Nebulizing technologies

aaaaaaaa

3.8. Computer vision

3.8.1. OpenCV

3.8.2. Face detection

Face detection has been studied for decades in the computer vision literature. Modern face detection algorithms can be categorized into four categories: cascade based methods [2, 10, 15, 16, 21], part based methods [19, 23, 30], channel feature based methods [25, 24], and neural network based methods [6, 14, 25, 28].

Here we highlight a few notable studies. A detailed survey can be found in [27, 29]. The seminal work by Viola and Jones [21] introduces inte- gral 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 stud- ies follow a similar pipeline. Among those variants, SURF cascade [15] achieves competitive performance. Chen et al. [2] learn face detection and alignment jointly in the same cascade framework and obtain promising detection performance.

One of the well-known part based methods is deformable part models (DPM) [7]. Deformable part models define face as a collection of parts and model the connections of parts through Latent Support Vector Machine. The

3.8.3. Hand gesture recognition

3.9. RDBMS

aaaaaaaa

3.10. Motion detection

aaaaaaaa

3.11. Camera recording and codecs

aaaaaaaa

3.12. Image filtering

aaaaaaaa

3.13. GIF generation

aaaaaaaa

3.14. Social media sharing APIs

aaaaaaaa

3.15. UI framework

aaaaaaaa

3.16. File transfer protocols

aaaaaaaa

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendices

A. Project Planning — Gantt diagram

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

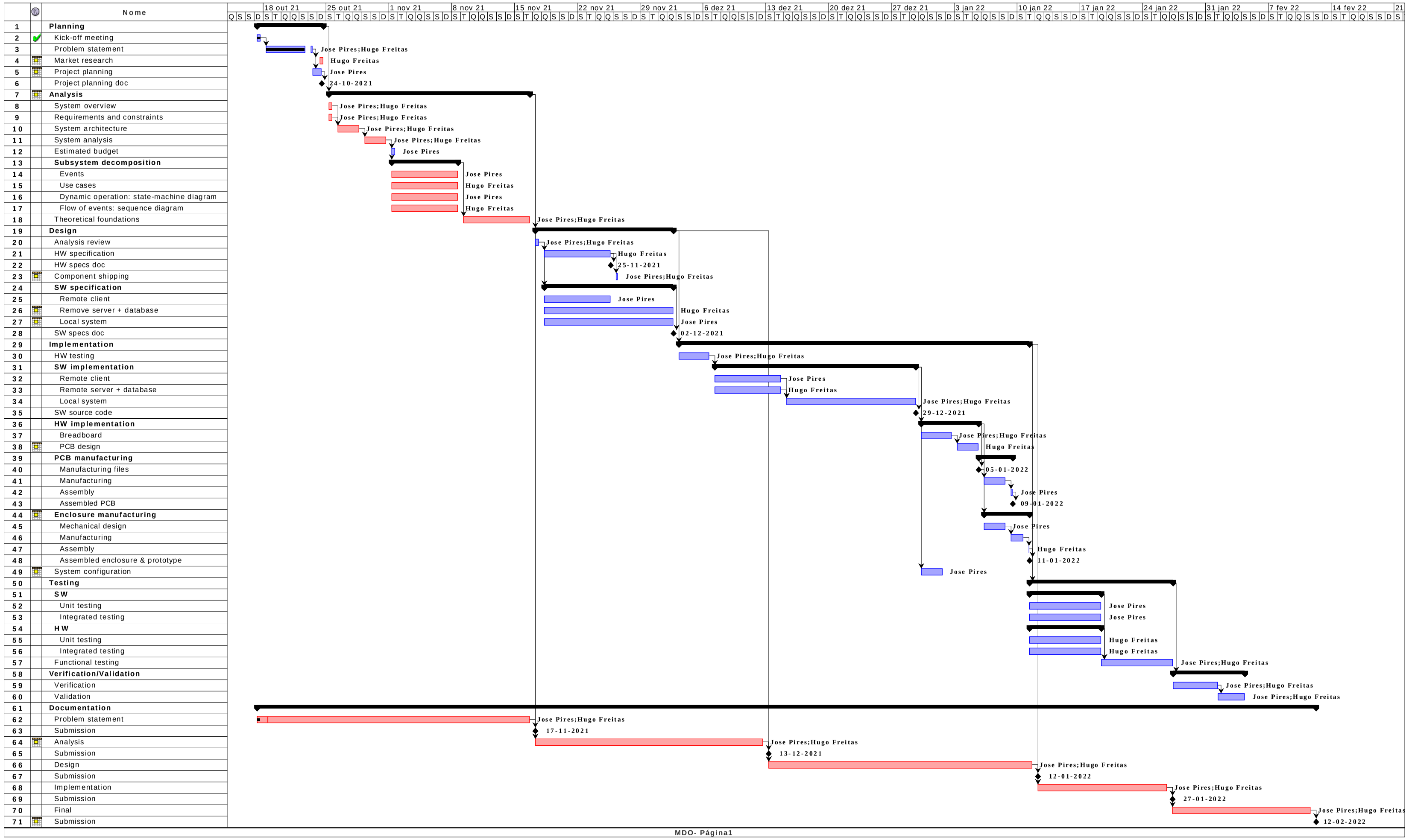


Figure A.1.: Project planning — Gantt diagram