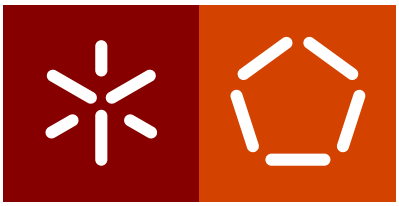

Universidade do Minho
Escola de Engenharia
Departamento de Engenharia Eletrónica
Laboratórios e Práticas Integradas 2

Integrator Project: Final Report

Radio Frequency Camera Assisted Rover (RFCAR)

Group 7

Nuno Rodrigues	A85207
Hugo Carvalho	A85156
Hugo Ferreira	A80665
João Faria	A85632
João de Carvalho	A83564
José Mendes	A85951
José Pires	A50178

Supervised by:
Professor Doutor Vitor Silva

July 11, 2020

Contents

List of Figures

List of Tables

List of Listings

Symbols

Symbol	Description	Unit
ω	angular velocity	rad/s
π	ratio of circumference of circle to its diameter	

1. Introduction

The present work, within the scope of the curricular unit of Laboratórios e Práticas Integradas II, consists in the project of the development of a product with strong digital basis, namely, a remote controlled vehicle used to assist exploration and maintenance in critical or unaccessible areas to human operators.

In this chapter are presented the project's motivation and goals, the product concept, the project planning and the document organisation.

1.1. Motivation and Goals

The main goal of this project is to develop a remote controlled vehicle with the following characteristics:

1. Remotely operated: the vehicle must be remotely operated to enable its usage in critical or unaccessible areas to human operators;
2. Provide visual feedback to the user: to be a valuable asset in the exploration and maintenance domains, the vehicle must provide visual feedback to the user of its surroundings.
3. Safe: the vehicle must be safe to use and prevent its damage and of its surroundings
4. Robust: the vehicle must be able to sustain harsh environmental conditions and provide redundant mechanisms to avoid control loss.
5. Affordable: so it can be an economically viable product.

1.2. Product concept

The envisioned product consists of a remote controlled vehicle used to assist exploration and maintenance domains, hereby, denominated as Radio Frequency Camera Assisted Rover (RFCAR). To satisfy such requirements, the vehicle must contain a remotely operated camera that provides a live video feed to the user.

Additionally, the vehicle must include an odometric system that assists the driving and avoids unintentional collisions when remote control is compromised, e.g., when connection is lost. The vehicle provides means for exploration and conditions assessment in critical or unaccessible areas to human operators, such as fluid pipelines and other hazardous locations.

1.3. 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 tasks of Analysis, Design, Implementation and Tests are performed in two distinct iterations as corresponding to the Waterfall project methodology.

Due to unpredictable circumstances, limiting the mobility of team staff and goods, the implementation stage will not be done at full extent, but rather at a simulation stage. Thus, to overcome these constraints, the project focus is shifted to the simulation stage, where an extensive framework as to built to model the system operation, test it, and providing valuable feedback for the dependents modules. As an example, the modules previously connected just by an RS232 link, must now include upstream a web module (TCP/IP) — the data is now effectively sent through the internet, and must be unpacked and delivery serially as expected if only the RS232 link was used.

The tasks are described as follows:

- Project Kick-off: in the project kick-off, the group is formed and the tutor is chosen. A brainstorming about conceivable devices takes place, whose viability is then assessed, resulting in the product concept definition (Milestone 0).

- State of the Art: in this stage, the working principle of the device is studied based on similar products and the system components and its characteristics are identified.

- Analysis: In the first stage — Analysis 1 — contains the analysis results of the state of the art. It should yield the specifications document, containing the requisites and restrictions to the project/product, on a quantifiable basis as required to initiate the design; for example, the vehicle's desired speed should be, at maximum, $2\ \mathrm{m/s}$. The second stage — Analysis 2 — contains the analysis of the first iteration of the development cycle.

- Design: it is done in two segments: modules design — where the modules are designed; integration design — where the interconnections between modules is designed. It can be subdivided into conceptual design and solution design.

  - In the conceptual design, several problem solutions are identified, quantifying its relevance for the project through a measuring scale, inserted into an evaluation matrix, for example, QFD.
  - In the solution design, the selected solution is developed. It must include the solution modelling, e.g.:
    * Control system: analytically and using simulation;
    * Transducer design: circuit design and simulation;
    * Power system: power supply, motors actuation and respective circuitry design and simulation;
    * Communications middleware: communication protocols evaluation and selection;
    * Software layers: for all required modules, and considering its interconnections, at distinct levels:
      · front end layer: user interface software, providing a easy and convenient way for the user to control and manage the system.
      · framework layer: software required to emulate/simulate and test the required system behaviour, providing seamless interfaces for the dependents modules
      · back end layer: software running behind the scenes, handling user commands received, system monitoring and control.

- Implementation: product implementation which is done by modular integration. In the first stage, the implementation is done in a prototyping environment — the assisting framework developed, yielding version alpha; in the second stage it must include the coding on the final target modules, yielding prototype beta.

- Tests: modular tests and integrated tests are performed. Tests are generally considered as those performed over any physical component or prototype. Here, it is used as a broader term, to reflect the tests conducted into the system and the several prototypes.

- Functional Verification/Validation: System verification may be performed to validate overall function, but not for quantifiable measurement, due to the latencies involved. Regarding validation, specially for an external agent, thus, it should be limited to user interface validation.

- Delivery: — project closure encompassing:
  1. Final prototype

2. Support documentation: how to replicate, instruction manual.
3. Final report
4. Public presentation

1.4. Report organisation

This report is organised as follows:

- In Chapter 2 lays out the theoretical foundations for project development, namely the project development methodologies and associated tools, and the communications technologies.
- In Chapter 3 are identified the key requirements and constraints the system being developed must meet from the end-user perspective (requirements) and, by defining well-established boundaries within the project resources (time, budget, technologies and know-how), the list of spefications is obtained.
- After defining the product specifications, the solutions space is explored in Chapter 4, providing the rationale for viable solutions and guiding the designer towards a best-compromise solution, yielding the preliminary design and the foreseen tests to the specifications.
- The preliminary design is further refined in Chapter 5 and decomposed into tractable blocks (subsystems) which can be designed independently and assigned to different design teams, allowing the transition to the implementation phase.
- Next, in Chapter 6, the design solution is implemented into the target platforms.
- Then, in Chapter 7, the implementation is tested at the subsystem level (unit testing) and system level (integrated testing), analysing and comparing the attained performance with the expected one.
- After product testing, in Chapter 8, the specifications must be verified and validated by an external agent. subsystem level (unit testing) and system level (integrated testing), analysing and comparing the attained performance with the expected one.
- Chapter 9 gives a summary of this report as well as prospect for future work.
- Lastly, the appendices (see Section 9) contain detailed information about project planning and development.

2. Theoretical foundations

In this chapter some background is provided for the main subjects. The fundamental technical concepts are presented as they proved its usefulness along the project, namely the project development methodologies and associated tools, and communications in detail.

2.1. Project methodologies

The methodologies used for the project development are briefly described next.

2.1.1. Waterfall

For the domain-specific design of software the waterfall methodology is used. The waterfall model (fig. 2.1) 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 [?]. It envisions the optimal method as a linear sequence of phases, starting from requirement elicitation to system testing and product shipment [?] 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. 2.2 (withdrawn from [?]).

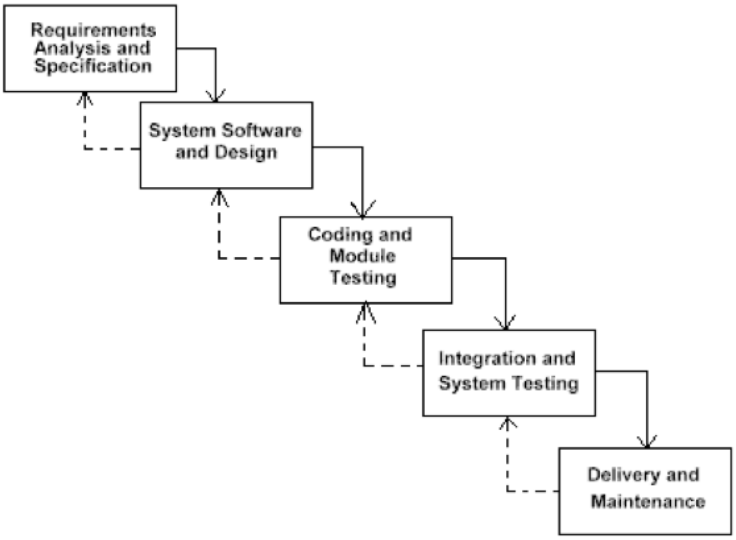

Figure 2.1.: Waterfall model diagram

2.1.2. 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 [?].

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) [?]. 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. 2.2) [?]:

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.

2.2. Communications

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

2.2.1. Bluetooth

Bluetooth is a ubiquituous radio-frequency technology (2.4 GHz) for wireless communication, generally at short distances. Bluetooth is managed by the Bluetooth Special Interest Group (SIG) and the current version of the standard is 5.0. Starting from version 4.0, also known as Bluetooth Low—Energy (BLE), power consumption was minimised, making it suitable for embedded and Internet of Things (IOT) applications. Bluetooth is a full protocol stack, illustrated in Fig. 2.3, comprising the controller, the host device and the applications interacting with the device. The Host Controller Interface (Internet Protocol) layer enables the host device to interface the controller, required for low-level operations such as asynchronous device

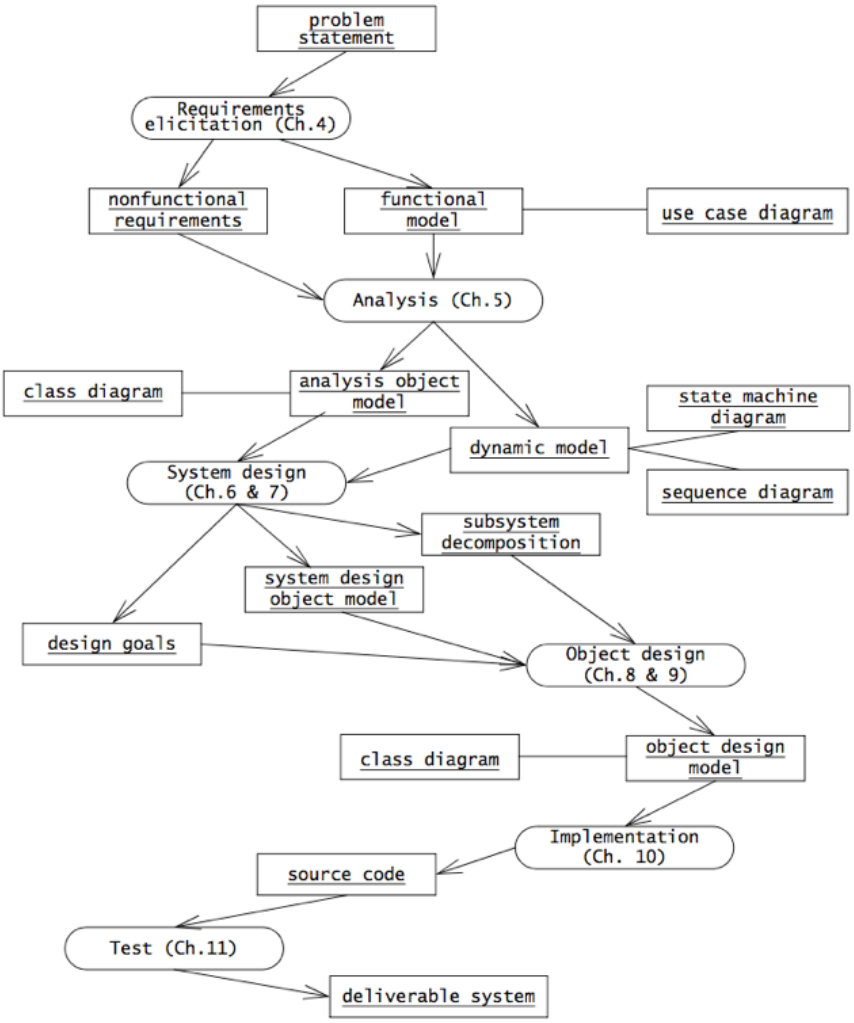

Figure 2.2.: An overview of the object-oriented software engineering development and their products. This diagram depicts only logical dependencies among work products (withdrawn from [?])

discovery or reading radio signal intensity. The host provides profiles to the external applications, easing the interaction betwen device and applications. Conceptually, Bluetooth is very similar to Transmission Control Protocol/Internet Protocol (TCP/IP) stack. Both are part of the network programming class and share the same principles of communication and data exchange between devices. The difference is that Bluetooth was designed for short distance communication, whereas the internet programming does not share this concern. This affects how two devices detect each other initially and how they establish the initial connection. From that moment on, the procedure is similar to the TCP/IP stack (see Fig. 2.4 and Fig. 2.5).

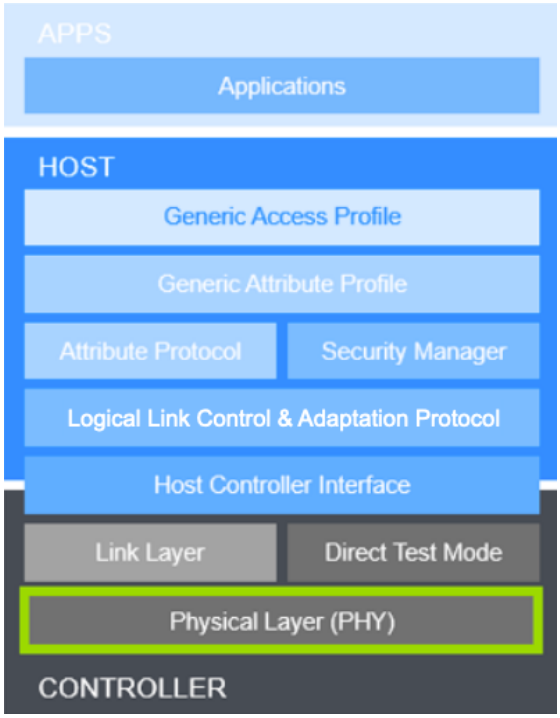

Figure 2.3.: Bluetooth 5.0 protocol stack

2.2.1.1. Establishing a connection

Establishing a connection depends if the device in analysis is trying to establish an outgoing or incoming connection. Basically, for the former case the device sends the first data packet to start the communication, and for the latter the device receives the first data packet. The devices that initiate outgoing connections must choose a target device and a transport protocol, before establishing the connection and exchange data. The devices that initiate incoming connections must select a transport protocol and then listen for incoming connections, before establishing the connection and exchange data [?].

Figures 2.4 and 2.5 illustrate this concept, comparing network, internet and bluetooth programming for outgoing and incoming connections, respectively. For outgoing connections, only the two first steps (choosing a target device, transport protocol and port number) are different; after the connection is established, the process is similar. For incoing connections the procedures are even more similar, with the major difference that port numbers are dynamically assigned for Bluetooth programming. The procedures for programming outgoing and incoming connections are presented next.

Outgoing connection programming procedure:

1. Choose a target device
2. Choose a transport protocol and port number

3. Establish a connection
4. Exchange data
5. Disconnect

Incoming connection programming procedure:

1. Choose a transport protocol and port number
2. Reserve local resources and go into listening mode
3. Wait and accept incoming connections
4. Exchange data
5. Disconnect

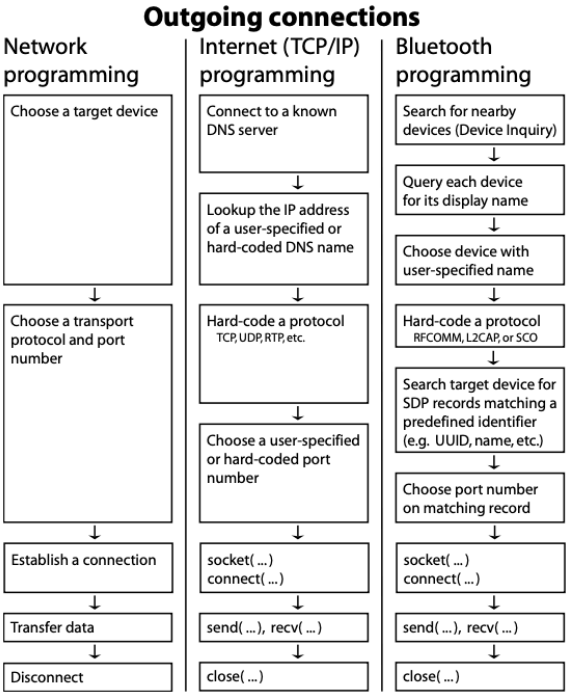

Figure 2.4.: Comparison of Network, Internet and Bluetooth programming for outgoing connections (withdrawn from [?])

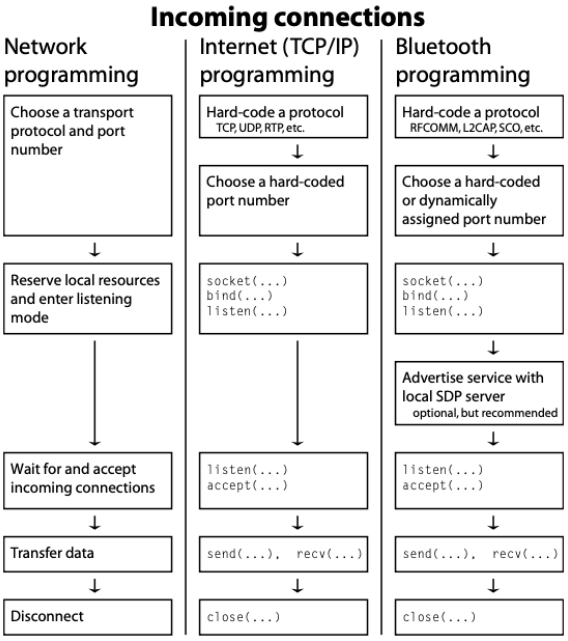

Figure 2.5.: Comparison of Network, Internet and Bluetooth programming for incoming connections (withdrawn from [?])

2.2.1.2. Selecting a target device

Every Bluetooth chip manufactured has a 48-bit unique address — BT address — identical to the Machine Address Code (MAC) for Ethernet protocol, acting as the basic addressing unit for Bluetooth programming.

The Bluetooth device must know the target device address to communicate. However, the client application does not need to know a priori the target address: the end-user provides a user-friendly name — device name — and the client translate this to the physical address when searching for nearby devices. The device name may not be unique, but the address must be.

In the device name lookup process, the device searches for the nearby devices by inquiring each device individually and compiling a list with their addresses, which is generally slow. A Bluetooth device does not announce its presence to other devices; it must start a device discovery process — Device Inquiry — to detect them. In the Device Inquiry process the device broadcast a discovery message and waits for replies. Each reply consists of the physical address of the device and of an integer identifier the class of the device (e.g., smartphone, headset, etc.). More detailed information, such as the device name, may be obtained by contacting each discovered device individually. Additionally, for privacy and power consumption reasons, the device may choose not to respond to device inquiries or connections attempts.

2.2.1.3. Transport protocols

Different applications have different needs, thus the need for different transport protocols. The two factors that distinguish these protocols are guarantees and semantics. The guarantees of a protocol state how hard it tries to deliver a packet sent by the application, yielding: robust protocols, like Transmission Control Protocol (TCP), which ensures that all sent packets are delivered or terminates the connection; best-effort protocols, like User Datagram Protocol (UDP), which makes a reasonable attempt at delivering transmitted packets, but ignores its failure. The semantics of a protocol concerns if it distinguishes between datagrams beginning and end, and can be either packet-based (UDP) or streams-based (TCP).

Bluetooth contains four essential transport protocols, namely, by relevance order:

1. Radio Frequency Communications (RFCOMM): reliable and stream-based protocol, which emulates well serail ports. It allows only 30 open ports per device and it is the most frequently used protocol.

2. Logical Link Control and Adaptation Protocol (L2CAP): packet-based protocol which can be configured for several levels of reliability, imposing delivery order, unlike UDP. It encapsulates the RFCOMM connection.

3. Asynchronous Connection-Oriented Logical (ACL): It is not explicitly used, but it encapsulates L2CAP connections. Two Bluetooth devices may have, at most, one ACL connection between them, which is used to transport all RFCOMM and L2CAP traffic.

4. Synchronous Connection-Oriented (SCO): best-effort and packet-based protocol, which is used exclusively to transmit voice-quality audio at precisely 64 Kb/s.

The summary of Bluetooth transport protocols is illustrated in Fig. 2.6, where is visibly the connection encapsulation. Two Bluetooth devices may have, at most, one ACL and SCO connection between them, whereas the number of RFCOMM and L2CAP active connection is limited only by the number of available ports.

2.2.1.4. Port Numbers

A port is used to enable multiple applications on the same device to use the same transport protocol. Bluetooth uses a slightly different terminology: for L2CAP, ports are called Protocol Service Multiplexers (PSM) (range of 1–32767); for RFCOMM, ports are called channels (range of 1–30).

Some protocols have a set of reserved/well-known ports, intended for specific usage. L2CAP reserves ports 1–1023 for standardized usage (e.g., Service Device Protocol (SDP) uses port 1), whereas RFCOMM does not have reserved ports, given its small number.

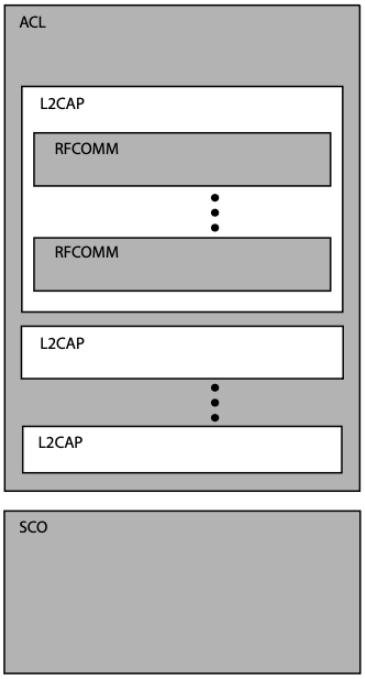

Figure 2.6.: Most relevant Bluetooth transport protocols (withdrawn from [?])

2.2.1.5. Service discovery

For a client application to initiate a outgoing connection it must know in which port the server application is listening. If the application is standard, then a reserved port is used. The traditional approach for Internet programming is hardcoding the same port number at both client and server: when the connection is established, the server listens on that part and the client connects to it. However, the static port definition is cumbersome and prevents two server applications from using the same port.

Bluetooth tries to solve this problem by introducing SDP, as illustrated in Fig. 2.7:

1. Each device keeps an SDP server listening on a well-known port.
2. When the server application is executed, it stores a description of itself and a port number in the SDP server on the local device.
3. Then, when a remote cliente application connects for the first time to the device, it provides a description of the service it is searching for, and the SDP server returns a list of all corresponding services with the respective associated port numbers.

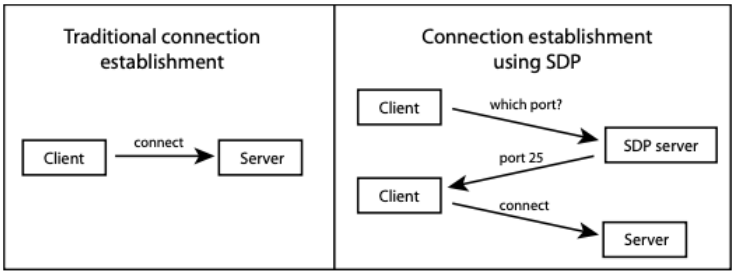

Figure 2.7.: Comparison between traditional connection establishment (internet) and using SDP (Bluetooth) (withdrawn from [?])

2.2.1.6. Host Controller Interface — HCI

The Internet Protocol defines how a host (e.g., a computer) interacts and communicates with the local Bluetooth adapter (the controller). All communications between these two agents are encapsulated in Internet Protocol packets, of four different kinds:

1. Command: sent from the host to the local adapter to control it, which can be used for starting a device discovery, connecting to a remote device, adjusting the connection parameters, amongst others.

2. Event: generated by the local adapter and sent to the host when an event of interest occurs, for example, device detected, connection established, etc.

3. ACL data: encapsulates data to send ou received from a remote Bluetooth device. In this sense, the Internet Protocol is a transport protocol for all transport protocols (ACL,L2CAP, and RFCOMM). When packets arrive to the local adapter, the Internet Protocol headers are removed and the pure ACL packet is transmitted through air.

4. SCO

2.2.1.7. Development Stacks for Bluetooth

A development stack refers to a collection of device drivers, development libraries and tools provided to enable software developers to create Bluetooth applications. In most operating systems there is a Bluetooth dominant stack, easing development, as there is a high level of incompatibility between Bluetooth development stacks. Since Bluetooth is a communications technology, the transport protocols supported by each stack are of particular interest. Occasionaly, wrappers around the developed libraries for a stack are created to provide a higher abstraction programming interface for Bluetooth, in languages like Python or Java. The most relevant development stacks and wrappers are depicted in Fig. 2.8.

PyBlueZ (GNU/Linux)	RFCOMM	L2CAP	SCO	HCI
PyBlueZ (Windows XP)	RFCOMM	L2CAP	SCO	HCI
BlueZ (GNU/Linux)	RFCOMM	L2CAP	SCO	HCI
Microsoft (Windows XP)	RFCOMM	L2CAP	SCO	HCI
Widcomm (Windows XP)	RFCOMM	L2CAP	SCO	HCI
Java – JSR82 (cross-platform)	RFCOMM	L2CAP	SCO	HCI
Series 60 Python (Symbian OS)	RFCOMM	L2CAP	SCO	HCI
Series 60 C++ (Symbian OS)	RFCOMM	L2CAP	SCO	HCI
OS X	RFCOMM	L2CAP	SCO	HCI

Figure 2.8.: Most relevant development Bluetooth stacks and wrappers and its supported protocols (withdrawn from [?])

The Bluetooth development stack selected was BlueZ, since its a powerful open source stack, included in all major GNU/Linux distributions and with extensive Application Programming Interface (API) s, supporting a extensive set of protocols which enables to fully explore the Bluetooth local resources. The RFCOMM, L2CAP, and SCO protocols are acessed using the standard sockets interface and the Internet Protocol is more conveniently used through the provided wrappers around the several Internet Protocol commands and events.

2.2.2. 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.

2.2.2.1. TCP/IP

The most commonly used protocols for Internet communications, including Wi-Fi, are TCP and Internet Protocol (IP), usually associated together, being part of the OSI model (Fig. 2.9), 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 [?], while the UDP provides data transportation without guaranteed data delivery or acknowledgments. The TCP can be thought of a reliable version of 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 [?].

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.

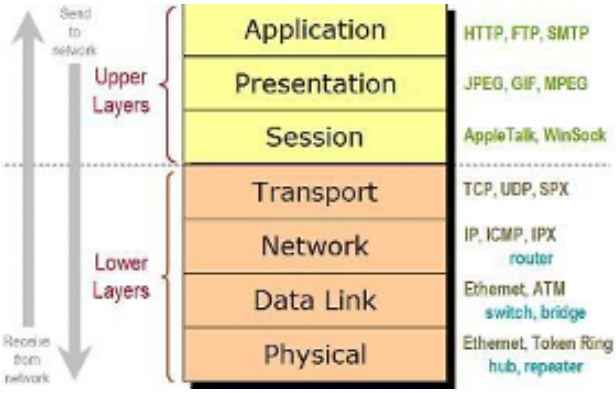

Figure 2.9.: OSI model

2.2.3. General Packet Radio Service — GPRS

General Packet Radio Service (GPRS) is a packet based wireless communication service that offers data rates from 9.05 up to 171.2 Kbps and continuous connection to the Internet for mobile phone and computer users [?]. GPRS is based on Global System for Mobile Communications (GSM) communications and complements existing services such as circuit switched cellular phone connections and the Short Message Service (SMS). However, GPRS is packet oriented (like the Internet), enabling packet data to be sent to or from a mobile device, closing many of the gaps in the GSM standard, namely [?]:

- enable access to company LAN and the Internet;
- provide reasonably high data transmission rates;
- enable the subscriber to be reachable at all times — not only for telephone calls but also for information such as new emails or latest news:

- offer flexible access, either for many subscribers at low data rates or few subscribers at high data rates, optimizing network usage;
- offer low cost access to new services

A GSM network is not able to transmit data in packet switched mode, as required by GPRS. Thus, two additional modules were required: Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN), yielding the GSM/GPRS network depicted in Fig. 2.10. On the user side there is a mobile device known as the Mobile Station (MS), consisting of a Mobile Equipment (ME) and the Subscriber Identity Module (SIM). For GPRS, the ME needs to be equipped with packet transmission capabilities, constituting three different classes of GPRS equipment [?]:

- Class A: equipment that can handle voice calls and packet data transfers at the same time;
- Class B: equipment that can handle voice or packet data traffic (but not simultaneously) and can put a packet transfer on hold to receive a phone call;
- Class C: equipment that can handle both voice and data, but has to disconnect from one mode explicitly in order to enable the other.

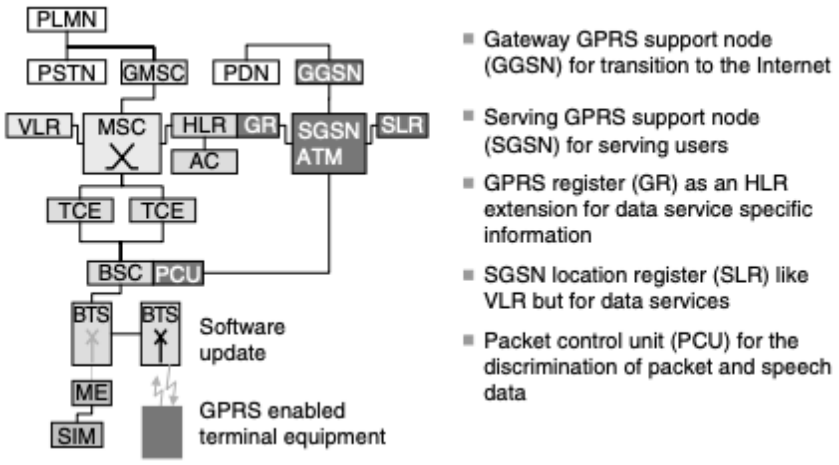

Figure 2.10.: GSM/GPRS network (withdrawn from [?])

The predominant protocol in the world of data networks is the IP, enabling the networking of different network architectures and the standardization of applications. In the mobile network GPRS the subscriber has a logical IP connection with an external data network, representing an actual member of this IP network. Packets can then be transferred between the MS and a server in the IP network, with the GPRS standard describing how they are transmitted on the radio interface and through the whole GPRS network.

Prior to the data transfer three important procedures must be performed [?] (see Fig. 2.11):

1. glsgprs Attach: the MS must be attached in the GPRS network. This is a logical procedure between the MS and the SGSN which takes note of the position, i.e. the 'routing area', of the MS. Storing and updating the position of the MS is particularly important for downlink (DL) transmissions because this information enables the GPRS network to locate the MS.

2. Activation of a Packet Data Protocol (PDP) context: A connection between the MS and the GGSN must be set up, so each node in the GPRS network knows how it has to forward the IP packets of this MS.

3. Data transfer: the path between the MS and the external data network is prepared, so IP packets can be sent through the GPRS network towards the destination address.

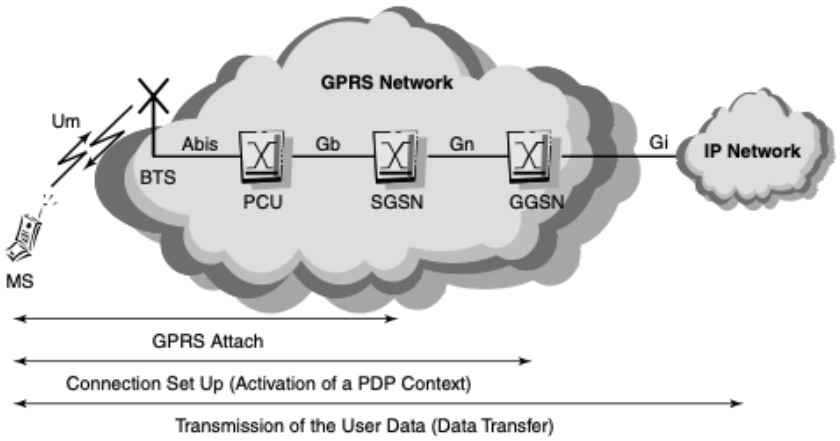

Figure 2.11.: GPRS procedures (withdrawn from [?])

Thus, the user required hardware for implementation of a GSM/GPRS network is the ME, namely, a GSM/GPRS modem and a SIM card with subscription to a mobile operator. A GSM/GPRS modem is generally driven through the RS-232 bus, using the Hayes command set (ATtention (AT) commands) to interface it.

2.2.4. 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 [?], 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 [?]. Bluetooth, as aforementioned, also uses sockets for multiprocess communication, given the device address, transport protocol and port number [?].

In a broader sense, a socket can be described as a method of Inter-Process Communication (IPC) that allows data to be exchanged between applications, either on the same host (computer) or on different hosts connected by a network [?], 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 [?]. It implements the Client-Server paradigm and implement several (standard) functions to access the operating system network resources, through system calls, in Linux [?].

There are two generic ways to use sockets: for outgoing connections — client socket — and for incoming connections — server socket. Fig. 2.12 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.

2.2.5. Client/server model

The client/server model is the most common form of network architecture used in data communications today [?]. 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 [?]. 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 [?].

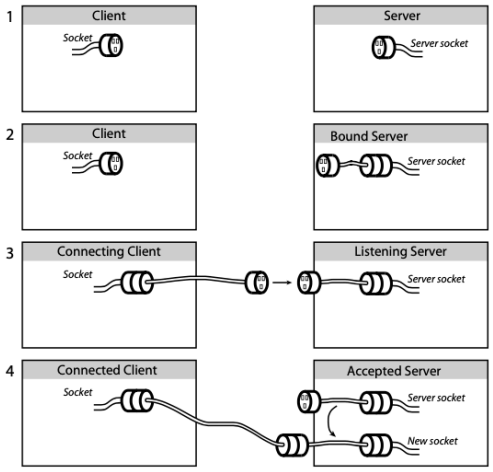

Figure 2.12.: Steps to obtain a connected socket (withdrawn from [?])

An example of a simple client-server model using the Socket API, through system calls, is presented in Fig. 2.13. The operation of sockets can be explained as follows [?]:

- 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.
- 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.

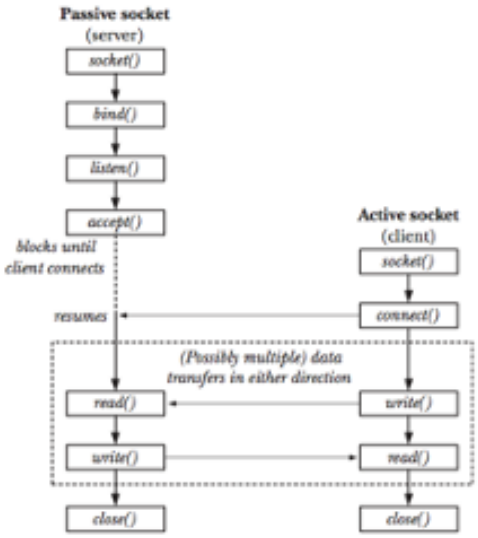

Figure 2.13.: Overview of UNIX system calls with sockets implementing a server/client paradigm (withdrawn from [?])

2.3. Concurrency

Concurrency is used to refer to things that appear to happen at the same time, but which may occur serially [?], 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 in POSIX, 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 [?]:

- 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.
- Synchronisation: this allows the management of shared resources between the concurrent execution contexts.

2.4. Android

The mobile Operating System (OS) chosen for this project was Android. This section gives an introduction to some android concepts to take into account in section 5.4.

2.4.1. Activity

An Android activity is a view that allows user interaction. Generally, an app can have multiple activities but always starts with the main activity. Each activity can start another to perform various tasks.

2.4.1.1. Activity Lifecycle

The activities in the system are managed as activity stacks. The activities are organized in a stack, the back stack, in the order by which they were opened, meaning that when a new activity is started it is placed onto the top of the stack. The prior activity won't come into the foreground unless the current activity exits, e.g., when the user presses a back, home or similar return button. Usually, the lifecycle of an android activity has this four key stages:

- Created: In this stage the activity is beaing created.
- Resumed:The activity is in the foreground thus now visible.
- Paused: Another activity is in foreground but this one is still visible.
- Stopped: The activity is running in the background and is no longer visible.

To control this lifecycle, one needs to implement the callback methods that refer to each specific stage by overwriting the pre-existing ones. The lifecycle of an activity is depicted in Fig. 2.14.

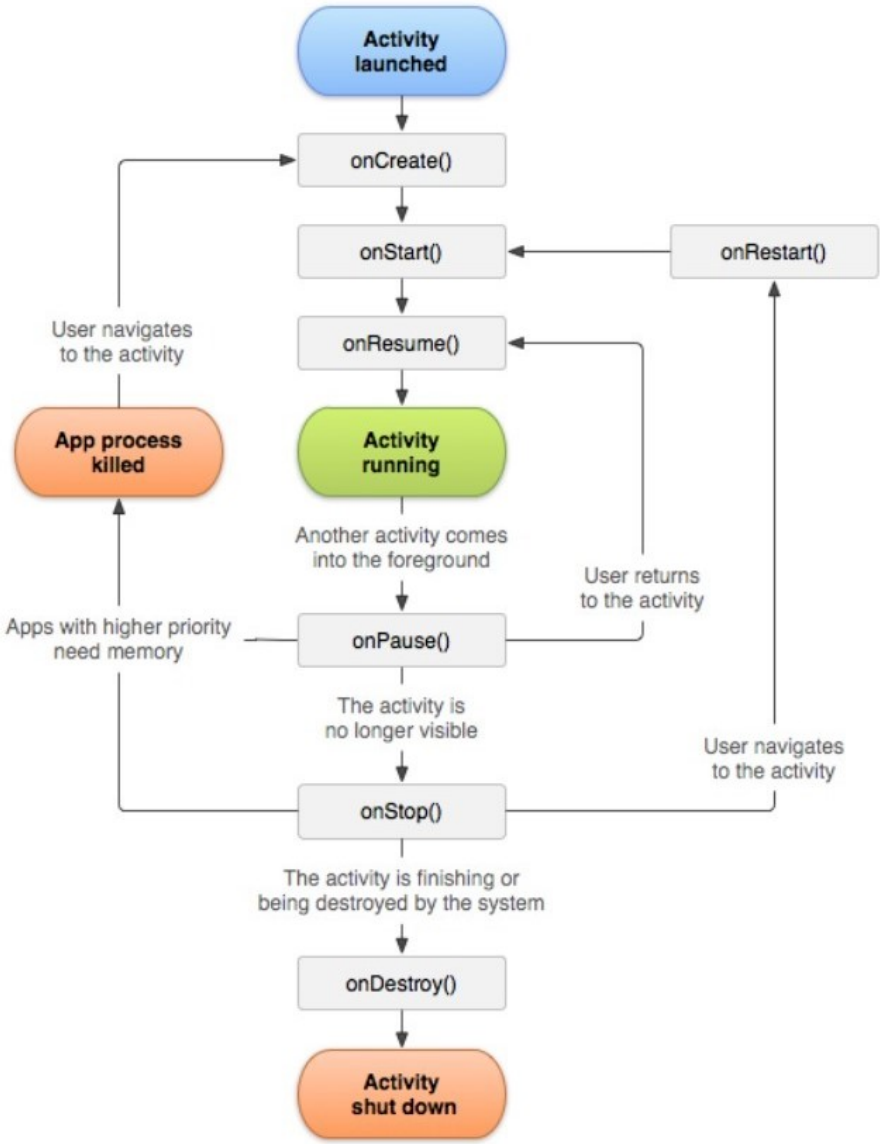

Figure 2.14.: Android activity lifecycle view

3. Requirements Elicitation and Specifications Definition

In this stage 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. The end-user perspective is generally abstract, thus requiring a methodic approach to obtain well-defined product requirements, i.e., product specifications. The product specifications are the result of a compromise between end-user requirements and its feasibility within the available project resources (time, budget, and technologies available). As the specifications are well-defined, they serve as design guidelines for the development team and can be tested later on to assess its feasibility and, ultimately, the quality of the product.

3.1. Foreseen specifications

In this section the foreseen product specifications of the system to be developed are provided. Such specifications were obtained through the intersection of customer, functional requirements and project restrictions.

3.1.1. Quality Function Deployment — QFD

The customer requirements are usually abstract and can collide with the functional requirements, compromising the fulfilment of the project. Thus, it raises the need of a methodology which converts abstract requirements into a series of concrete engineering specifications.

An efficient quality assessment methodology is the use of a Quality House (Quality Function Deployment (QFD)). In this method, the desired requirements are laid out as rows and the engineering specifications/restrictions as columns. In the intersections lies a symbol representing the strength (weak, moderate or strong — Figure 3.2) of the relationship requirement-specification. This symbol is one of the many tools that allow the quantification of relations existing between the customer requirements and engineering specifications.

For instance, the ‘engine power’ specification and the ‘fast’ requirement have a very strong correlation (9) since the power of the engine is directly responsible for the speed of the car.

Along with the requirements, the importance given to each is also specified, ranging from 1 (lowest importance) to 5 (highest importance) these, along with the number at each intersection, will be used to calculate the importance of each specification and thus assign priorities for the Design Team.

Lastly, the triangle shape (the ‘roof’ within the house metaphor) serves as another way of measuring relationships, this time between each specification: such is achieved by placing a symbol (ranging from very negative to very positive, see Figure 3.1) in the diagonal intersection of two specifications. I.e., the battery life will have a very negative correlation with the battery temperature, due to the fact that the increase of the temperature will cause a decrease in life time. As such a ‘very negative’ correlation was placed in the diagonal intersection betwixt ‘Battery Life’ and ‘Battery Temperature’.

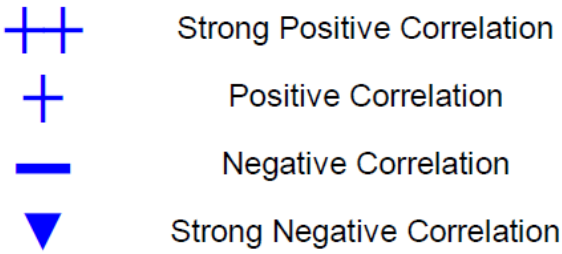

Figure 3.1.: Quality House — Specification Correlation Strength Symbols

Figure 3.3 shows the ‘Quality House’ for the RF CAR containing:

- Customer Requirements: Vehicle Integrity; Obstacle Avoidance; Reliable Feedback; Fast Response; Fast; Budget Friendly; Low Consumption; Small.
- Functional Requirements or Restrictions: Autonomy; Battery Temperature; Minimum Distance to Obstacle; Maximum Velocity; Motor Expectancy; Cost of Production; Motor Power; Ramp-Up Speed Time; Frame Rate; Camera Range; Resolution; Communication Range; Communication Speed; Dimensions; Mass.
- Intersection Values (referencing the strength of the requirement-specification correlation) — see Figure 3.2.
- Analytical Data, depicting, in a quantifiable manner, the aims of the project and the relevance of each entity:
  - Target or Limit Value: The metrics the design team will be based on, white spaces are left for either further discussion and refinement.

- Difficulty: Allows a subjective input to be added so that 'importance' can be changed to balance unforeseen circumstances.
- Importance and Relative Weight: The main conclusion for which the QFD was used, it assigns the priorities for the design team in an objective manner.

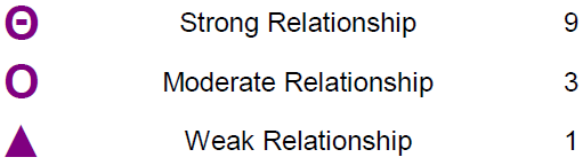

Figure 3.2.: Quality House – Relationship Strength Symbols

With the QFD, the prioritized ranks and specification targets were obtained and diffused within the Design Team with a straightforward guideline. For instance, the low cost requirement should be prioritized over all other specifications, followed by the maximum speed, Ramp-Up Speed Time and so on. On the other hand, the engine expectancy is of little to no consequence (note that the importance added up to a mere 3%), followed by the camera-related specifications. This could be regarded as a point of discussion, which should be prioritized? The functionality of the car or the the feedback provided by the camera?

With the last point in mind, the QFD has the advantage of promoting further discussion, simply by changing the importance of a requirement the priority ranking will change, ergo the priorities can be altered, easily and efficiently, if deemed appropriate.

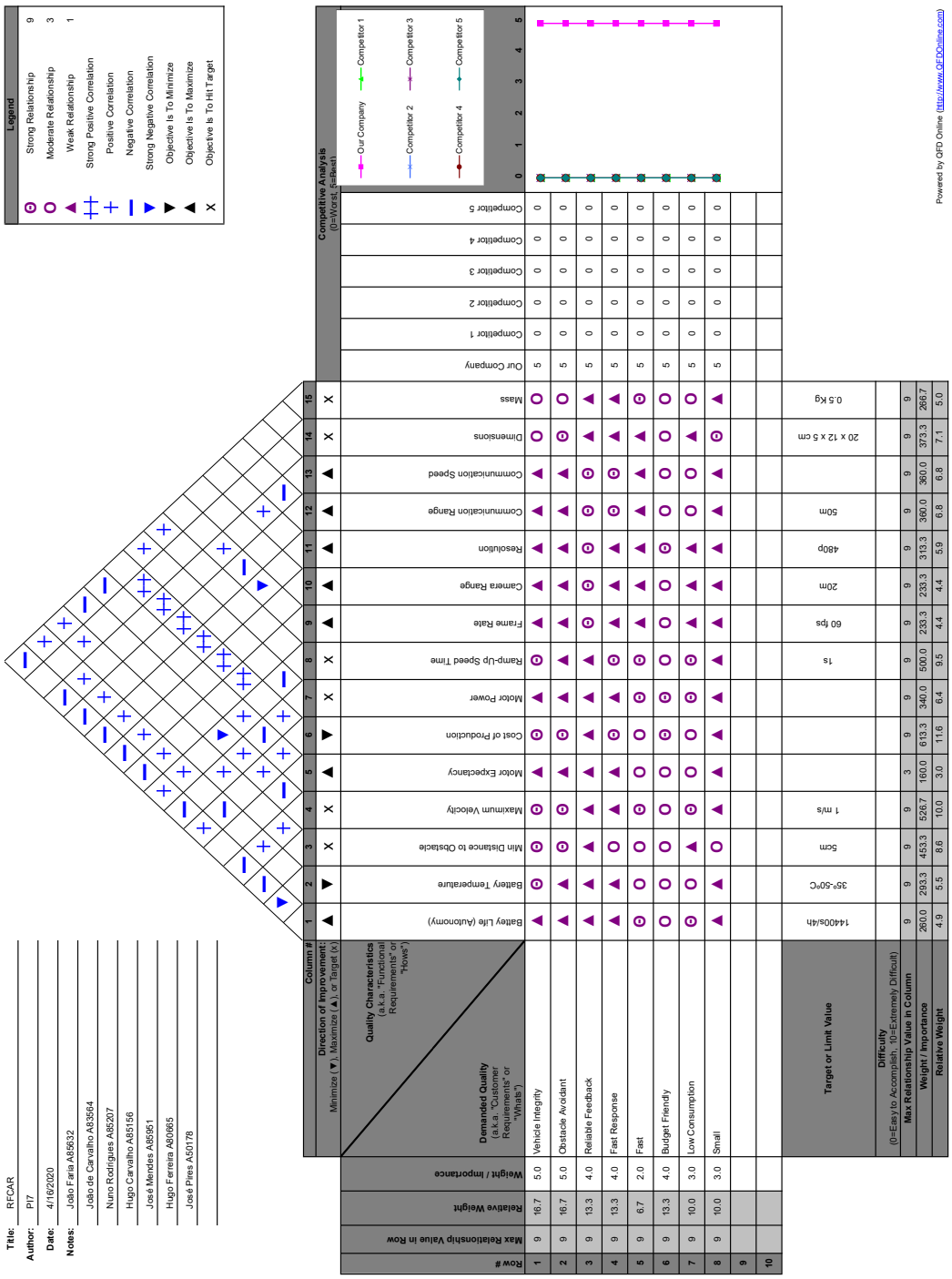

Figure 3.3.: Project Study – RFCar Quality House

3.1.2. Vehicle Autonomy

The vehicle is operated in wireless mode, thus, a portable power source must be included. The autonomy refers to the vehicle operating hours since the battery is fully charged and safely discharged and should be observed for the following scenarios:

- No load and vehicle operating at maximum speed;
- No load and vehicle operating at mean speed;
- Maximum load and vehicle operating at maximum speed;
- Maximum load and vehicle operating at mean speed.

3.1.3. Speed

The vehicle must be operated within a safe range of speed, while also not increasing excessively the power consumption. Thus, these speed boundaries should be tested in the absence of an external load and in the presence of the maximum load.

3.1.4. Safety

Vehicle self integrity protection is a requirement in product design, especially considering the vehicle is to be remotely operated. The safety in the operation can be analysed in two ways, and considers the preservation of people and goods. For the former, it is important to assure safe interaction as well as user operation – the vehicle may encounter several obstacles along its path, but it must not inflict any damage. For the latter, the vehicle under operating conditions must not inflict any damage to goods. Thus, in the presence of conflicting user commands violating the safety of people and goods, the local system should override them, taking corrective measures to prevent it. The same holds true if the communication between user and system is lost.

3.1.5. Image acquisition

The vehicle is equipped with a camera to assist in its navigation, thus, requiring it to be fed to the user's platform appropriately. To do so, several functionalities details need to be addressed efficiently. It was selected the most relevant three and these include the frame rate, the resolution and the image range.

3.1.5.1. Frame rate

Frame rate refers to the frequency at which independent still images appear on the screen. A better image motion is the result of a higher frame rate but the processing overhead increases as well, so a compromise must be achieved between the quality of the image and the increased processing overhead required. The minimum frame rate defined must be such that allows a clear view of the navigation.

3.1.5.2. Range

How far can the camera capture images without being distorted or unseen by the user. The range must be such that allows the user to see the obstacles when the car is heading to them and provide enough time to change the direction.

3.1.5.3. Resolution

The amount of detail that the camera can capture. It is measured in pixels. The quality of the acquired image is proportional to the number of pixels but a increased resolution requires a greater data transfer and processing overhead, thus, a compromise must be achieved. The minimum resolution must be such that provides the least amount of information required for the user.

3.1.6. Communication

For the implementation of the communication, several stages must be considered: Reliability, redundancy and communication range.

3.1.6.1. Reliability

A communication is reliable if it guarantees measures to deliver the data conveyed in the communication link. As reliability imposes these measures, it also increases memory footprint, which must be considered depending on the case. For the devised product, an user command must be acknowledged to be processed, otherwise, the user must be informed; on the other hand, loosing frames from the video feed is not so critical — user can still observe conveniently the eld of vision if the frame rate is within acceptable boundaries.

3.1.6.2. Redundancy

The communication protocols are not flawless and the car relies on them to be controlled. If the communication is lost, the car cannot be controlled. A possible solution for this issue is using redundancy in the communication protocols (e.g Wi-Fi and GPRS), so if one protocol fails, the car will still be controlled using the other.

3.1.6.3. Range

The communication protocols have a limited range of operation, and, as such, regarding the environment on which the car is used the range can be changed. The range established the maximum distance allowed between user and system for communication purposes.

3.1.7. Responsiveness

The movement of the car will be determined by the tilt movement of the smartphone. Sensibility refers to the responsiveness of the car on the minimum smartphone tilt movement. The sensibility must be in an range of values in which small unintentional movements will not be enough to change the state of the car and it does not take big smartphone tilts for the car to move.

3.1.8. Closed loop error

The speed, direction and safe distance to avoid collisions must be continuously monitored to ensure proper vehicle operation. The closed loop error must then be checked mainly in three situations as a response to an user command:

- speed: the user issued an command with a given mean speed, which should be compared with the steady-state mean speed of the vehicle.
- direction: the user issued an command with a given direction, which should be compared to the vehicle direction.
- safe distance to avoid collisions: the user issued an command with a given direction and speed which can cause it to crash. The local control must influence, to prevent collision, and the final distance to the obstacles must be assessed and compared to the defined one.

3.1.9. Summary

Table 3.1 lists the foreseen product specifications.

Table 3.1.: Specifications

	Values	Explanation
Autonomy	4 h	Time interval between battery fully charged and safely discharged
Speed Range	0.1 to 1 m/s	Speed at which the car can operate
Frame Rate	60 fps	Frequency at which independent still images appear on the screen
Camera Range	20 m	How far can the camera capture images without loosing resolution
Camera resolution	480p	Amount of detail that the camera can capture
Communication Range	50 m	Maximum distance between the car and the smarphone without losing connection
Speed Error	5 %	Maximum difference between desired and real speed
Direction Error	5%	Maximum difference between desired and real direction
Distance Error	5 %	Maximum difference between desired and real distance to the obstacle
Dimensions	20x12x5 cm	Dimensions of the car
Weight	0.5 kg	Weight of the car

4. Analysis

After defining the product specifications, it is possible to start exploring the solutions space within the project's scope, providing the rationale for viable solutions and guiding the designer towards a best-compromise solution. In this chapter are presented the preliminary design and the foreseen tests to the specifications.

4.1. Initial design

Following an analysis of the product's family tree (remote controlled cars), the state of the art and the QFD matrix in fig. 3.3, an initial design of the product itself can be produced (fig. 4.1). The selected approach was top-down, in the sense that the requirements and specifications were addressed and that resulted in a general diagram of the product concept. Some macro-level decisions were made in this stage to narrow the problem's solutions pool, as follows:

- The car itself should be battery-powered, as it is a free-moving object that is intended to work in environments where trailing cables could interfere with its regular movement.
- The device used to control the car should ideally be one already owned by the user, with an integrated screen (e.g. smartphone), as it would make it more affordable and have a more straightforward interface.
- The protocols for communication between the controlling device and the Rover should be chosen from within the pool of those readily available to smartphones (e.g. Wi-Fi, GPRS, Bluetooth) to keep the price of the overall product down and make it as practical as possible.
- The control and communication unit for the Rover should be divided into two modules: one which can measure and process sensor inputs and control the actuators in real-time, as well as communicate with the user-operated device through a low-latency connection. And another one which can interface directly with the camera module and manage data transmission to the user at the applicational level of the TCP/IP protocol stack, with enough throughput for the specified video resolution and framerate.

The latter should also exchange sensor reading values and commands with the user-controlled device, introducing redundancy to the controls and thus allowing for more reliable operation.

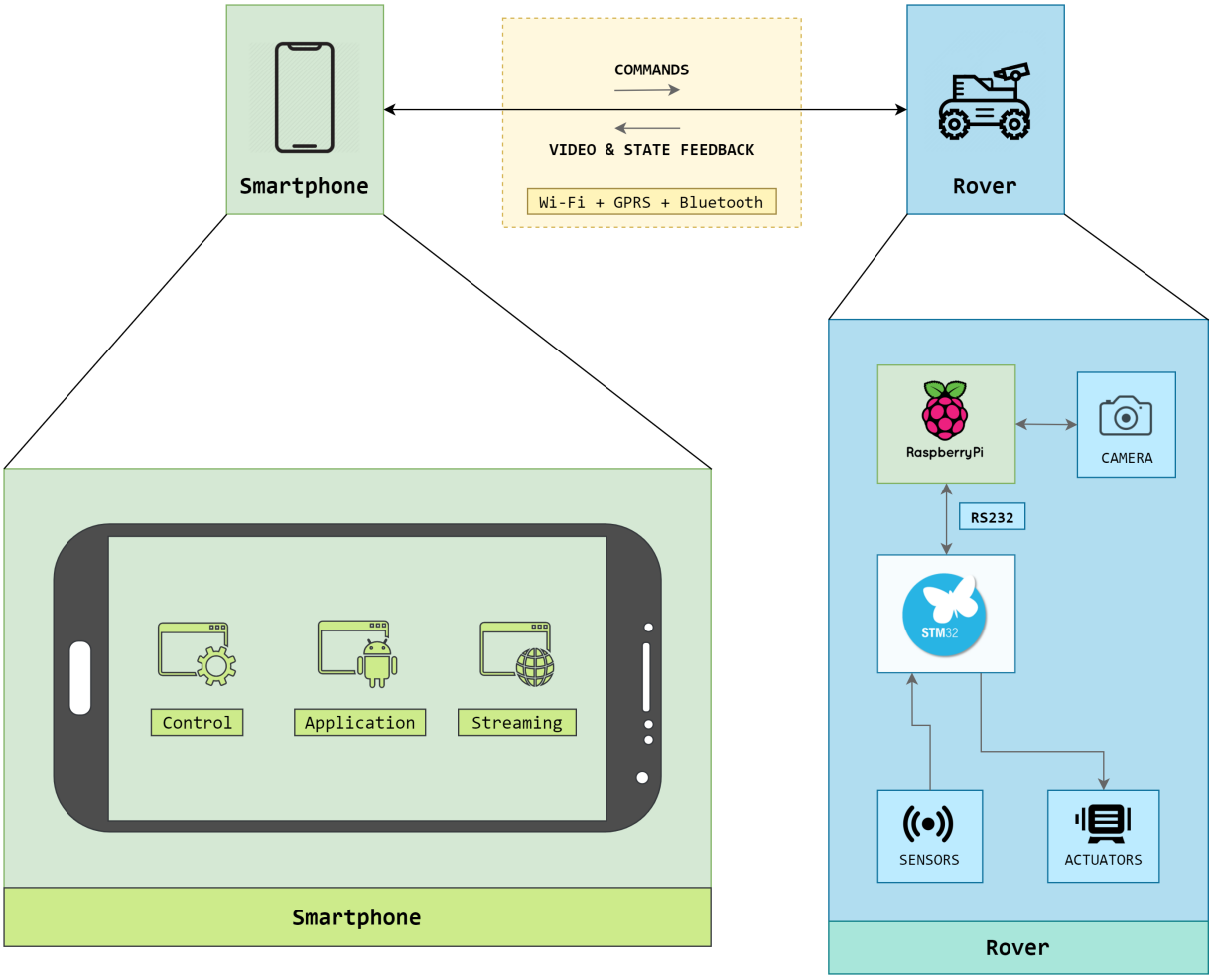

Figure 4.1.: Initial design: Block diagram view

Thus, summarising, the initial design yields the system illustrated in fig. 4.1, comprised of:

- Raspberry Pi: Interfaces with the camera directly, streaming that information to the smartphone. It also receives user commands and sends sensorial information it receives from the STM32 module back to the Smartphone, all through redundant Wi-Fi and GPRS connections;
- STM32: Receives user commands and sends back sensorial information through a Bluetooth connection, information it also uses to control the actuators;
- Actuators: DC Motors that control the movement of the Rover and headlights for nocturnal or low light conditions;

- Sensors: Odometric sensors that support the detection of obstacles and luminosity sensors;
- Camera: Device connected to the Raspberry Pi that allows the live stream of the car´s surrounding environment;
- Smartphone: Grant visual feedback from the live feed of the camera also allowing the user to control the movement of the vehicle intuitively;

For simulation purposes and in conformance with the extraordinary conditions imposed by the recently enacted confinement measures, the need rose to create a virtual environment to simulate the various subsystems of the Rover. This solution allows for integrated testing without the need to deploy it to the hardware, as illustrated in fig. 4.2.

The first of said subsystems is the Physical Environment Virtual Subsystem, which simulates the Rover and the physical environment, also receiving actuator inputs from the Navigation Virtual Subsystem, for which it generates the sensor readings. The latter also exchanges that information, as well as the status of the Rover and the commands from the user with the Smartphone module through a Bluetooth connection, which separates it from the non-simulated environment.

The Remote Vision Virtual Subsystem interfaces with the computer's Web Camera, which is meant to simulate the onboard camera of the Rover. Moreover, it can also communicate through Wi-Fi and GPRS, thus ensuring that communication is kept despite any failure from the Navigation Virtual Subsystem, as well as having shared access to the sensor reading values for monitoring of certain key parameters, depending on their refresh rate.

The RS232 connection between both controlling subsystems ensures proper synchronism and cooperation between them.

4.2. Foreseen specifications tests

The functional testing is generally regarded as those performed over any physical component or prototype. Here, however, a broader sense is used, to reflect the tests conducted into the system and the several prototypes, under the abnormal present circumstances. Moreover, as indicated in the design, the current development strategy encompasses the virtualization of all hardware components, enclosed in a single virtual environment.

Thus, it does not make sense to perform hardware related tests such as velocity measurements, autonomy, safety, etc. As such, the focus is shifted towards software and control verification, encompassing the following tests: functionality, image acquisition, communication, and control algorithms correctness.

The tests are divided into verification and validation tests.

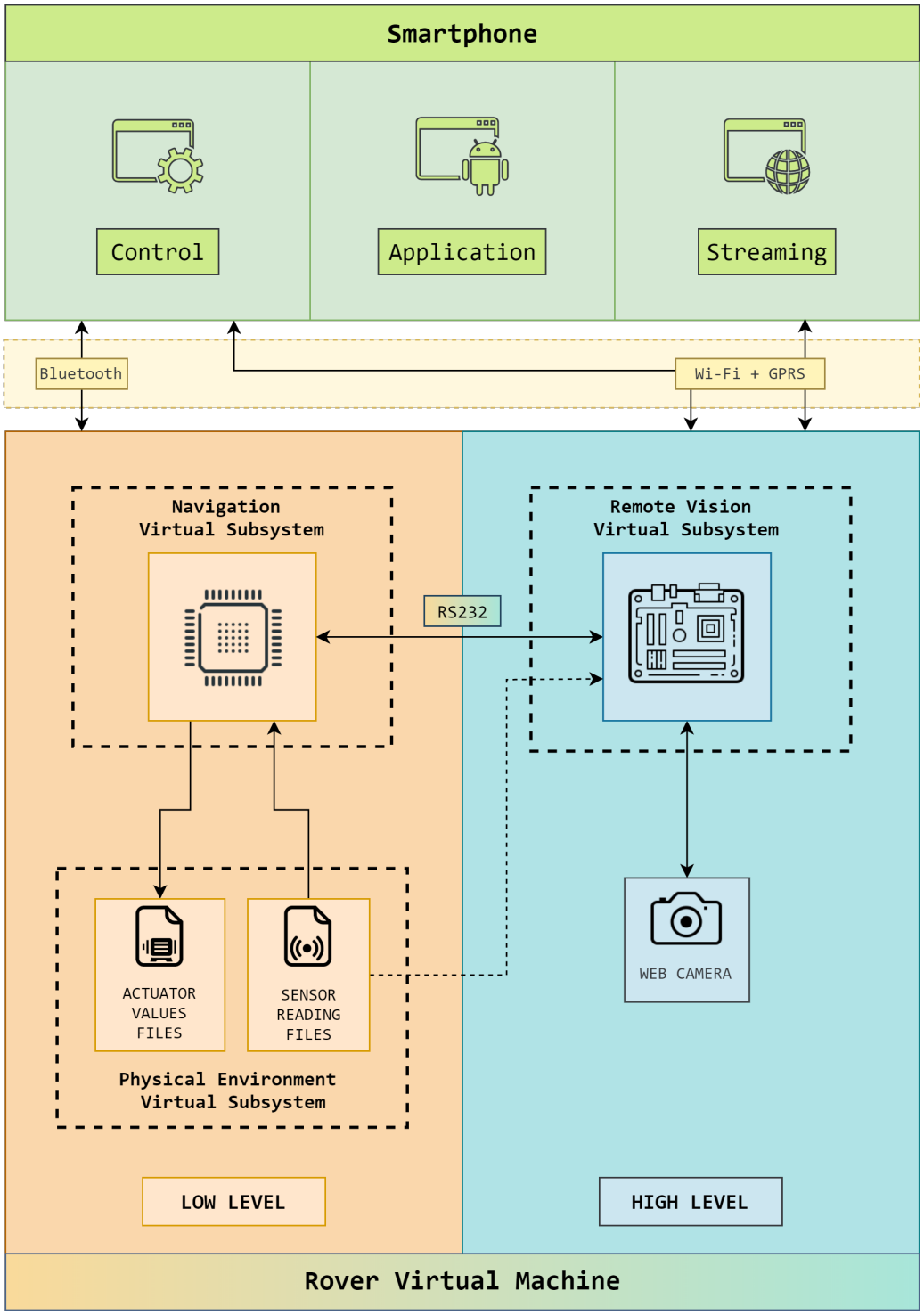

Figure 4.2.: Initial design: Virtual environment block diagram view

4.2.1. Verification tests

The verification tests are tests performed internally by the design team to check the compliance of the foreseen specifications. These tests are done after the prototype alpha is concluded.

4.2.1.1. Functionality

The remotely operated vehicle is composed of several modules distributed along several different platforms, some of which distanced from each other. In doing so, the proposed sets of functionalities should be tested in the integrated system, by tracking and analysing the user commands issued along the way until it finally reaches the vehicle (in the virtual environment), assessing if it is correctly processed. For example, if the user issues the vehicle to move to a given place (via smartphone interaction), the message sent to the vehicle must be signalled in each endpoint hit, and the vehicle should move to that place, symbolically detected by the modification of its virtual coordinates.

4.2.1.2. Image acquisition

The vehicle is equipped with remote vision to assist the user in its navigation, thus, requiring the following variables to be tested: frame rate, range, and resolution. In the current scenario, the virtual environment should provide access to a integrated camera, being fairly common nowadays in every modular component, thus, enabling easy testing.

4.2.1.2.1. Frame rate

To test frame rate, the user screen must be updated with the number of frames received from the camera per second and checked against the defined boundaries.

4.2.1.2.2. Range

To test camera's range, an object must be captured at increasing distances, until the object resolution fades or is unclear.

4.2.1.2.3. Resolution

The minimum resolution should be tested as providing the least amount of information required for the user, while minimizing data transfer and processing overhead.

4.2.1.3. Communication

The communication tests are performed in compliance to the specifications provided in Section 3.1.6, namely reliability and redundancy.

4.2.1.3.1. Reliability

To test communication reliability, the most critical communication link is chosen, namely the wireless communication between user's platform (smartphone) and vehicle's platform (virtual environment). Then, the communication link and protocol selected are tested by monitoring the ratio of dropped packets versus total packets, using an appropriate tool on both directions for transmission and reception.

4.2.1.3.2. Redundancy

To test communication redundancy, one communication channel should be turned off, verifying if the communication between nodes is still possible through another communication channel. For example, in the communication between host (user's platform) and remote (virtual environment) guest systems, the priority communication is performed between host and high-level subsystem. However, if this is turned off, the host must also be able to communicate with the remote system via low-level subsystem.

4.2.1.4. Correctness of the control algorithms

As previously mentioned, the speed and position must be continuously monitored to ensure proper vehicle operation. Under the current circumstances, where the sensors and actuators are virtualized, the control loops can be externally stimulated through input files containing the relevant data. Then, the behaviour of the system can be analysed and verified for some cornercase situations, assessing the control algorithms correctness.

4.2.2. Validation tests

The validation tests should be performed by the client using the product's manual, so it is expected that a user without prior experience with the product should be able to use it correctly and safely. On the current circumstances, validation tests should be limited to user interface validation.

For this purpose, an external agent will be provided with the software application and the respective installation and usage manuals, and the feedback will be collected and processed to further improve the product.

5. Design

In the design phase, the solution is developed in the various domains and a thorough specification is written, paving the way for implementation. In this chapter is presented the design for the various domains and subsystems identified.

5.1. Navigation Virtual Subsystem

The Navigation subsystem hosts the core of the system functionality-wise, which is the control routine. This means that it should strive to not only make accurate readings and calculations but also be as efficient as possible in managing those processes in order to introduce very little delay and meet timing requirements.

To meet these requirements as best as possible it should be capable of :

- Gathering information from the physical domain at equally distant instants kT_s and output an electrical representation of the command variable at equally distant instants kT_o;
- Acquiring commands from the Smartphone and Remote Vision Subsystems, identifying matches that will allow it to validate those commands and feeding them into the control rule in a useful format;
- Providing real-time feedback to the user about its status.

The first task should be idealizing the control system itself, understanding what inputs are needed to control the machine and then how it could be used to manipulate the wheels of the car. After that, the rest of system should be designed to fit the needs of the control rules and algorithms and use them to react as fast and consistently as possible within its own constraints and those of the other subsystems.

5.1.1. Control

The objective of the control module is to be capable of, with an input reference velocity and steering angle, outputting the appropriate command variables to achieve said references. The first step in its design is to

conceptualize the car's model; since the interest lies in the motion of the car, the kinematic model will be the one to be studied.

5.1.1.1. Conception Of Car Model

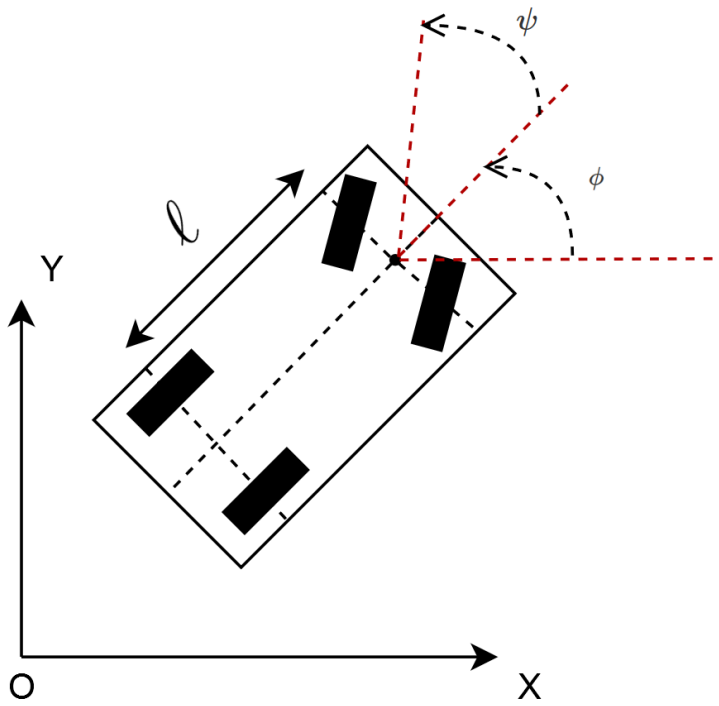

Figure 5.1.: Kinematic Model of Car

Considering the kinematic model of a four-wheel vehicle with a wheelbase length ℓ, a linear velocity v(t) and an angular velocity $\omega(t)$ the following situation will occur: since the rear wheels will remain in the same position no matter where the car is facing towards, they will be facing whatever orientation, ϕ, the car is facing, however, in order for the car to be capable of moving into wherever the user tells it to, the front wheels must turn, ergo a steering angle ψ, must be considered. The resulting direction the car will be going in is $\phi + \psi$. With these considerations, a desired angle of tilt θ and considering that $\phi = 0$, in other words, that whatever direction the car is told to face, it will be relative to its current direction, the following

model is obtained:

$$\dot{x} = v(t)cos(\psi) \tag{5.1}$$

$$\dot{y} = v(t)sin(\psi) \tag{5.2}$$

$$\dot{\psi} = \omega(t) = \frac{v(t)}{\ell}\theta \tag{5.3}$$

However this model is not enough for simulation purposes. The simulation has the objective of granting the designers clear ideas of the response of the systems towards given inputs, therefore, for implementation purposes, the simulation must give feedback on the position of the car, its heading and the linear velocity of the right rear wheel (v_r), and the left rear wheel (v_l) therefore the model will have to changed with these details in mind, which can be achieved considering that:

$$\omega(t) = \frac{v_r(t) - v_l(t)}{\ell} \tag{5.4}$$

$$v(t) = \frac{1}{2}(v_r(t) + v_l(t)) \tag{5.5}$$

Solving the system above for v_r and v_l:

$$v_r(t) = v(t) + \frac{\omega(t)\ell}{2} \tag{5.6}$$

$$v_l(t) = v(t) - \frac{\omega(t)\ell}{2} \tag{5.7}$$

Returning the model in order to $\dot{x}$ and $\dot{y}$:

$$\dot{x} = v(t)cos(\psi) - \frac{\ell}{2}\omega(t)sin(\psi) \tag{5.8}$$

$$\dot{y} = v(t)sin(\psi) + \frac{\ell}{2}\omega(t)cos(\psi) \tag{5.9}$$

$$\dot{\psi} = \omega(t) = \frac{v(t)}{\ell}\theta \tag{5.10}$$

5.1.1.2. Simulation Model

The mathematical model determined in Section 5.1.1.1 was simulated in the Matlab/Simulink environment. Converting the mathematical model into a simulink subsystem yields (Fig. 5.2): After obtaining the model of the car, the control simulations of the system may begin:

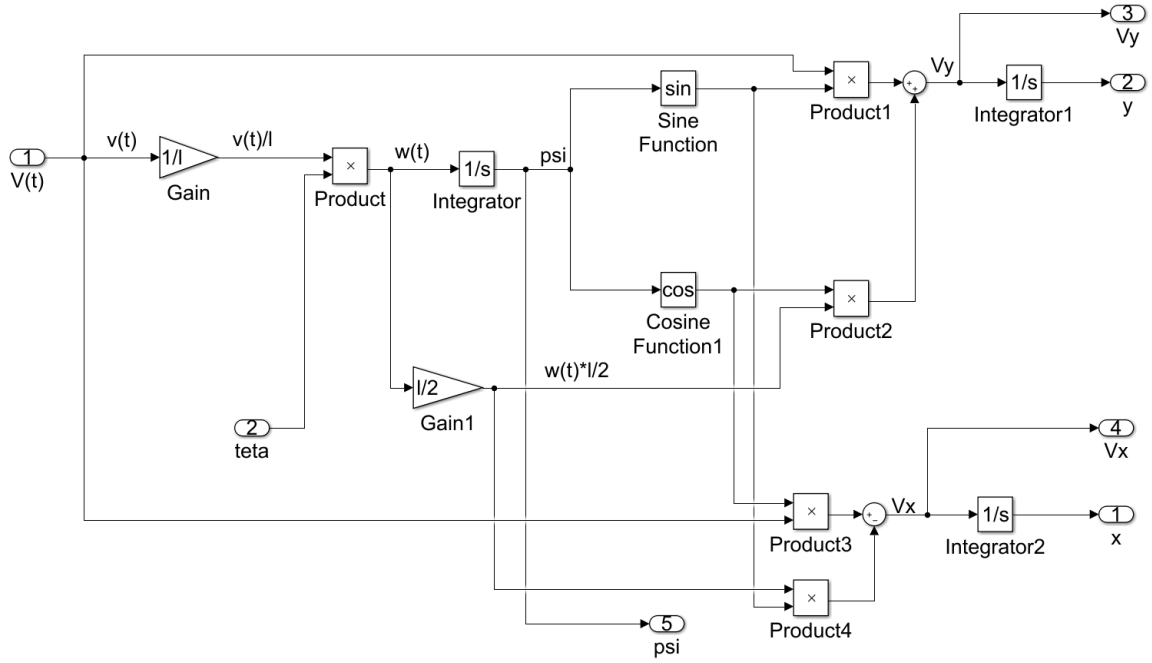

Figure 5.2.: Car Model in a Simulink Subsystem

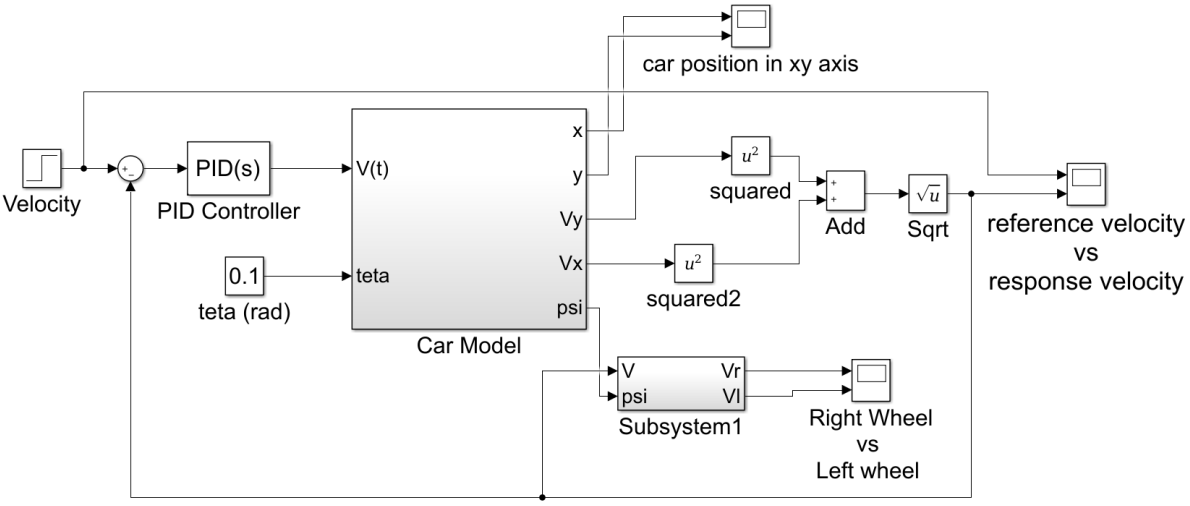

Figure 5.3.: Simulation Schematic

The simulation schematic uses the model in Fig. 5.2 within the 'Car Model' subsystem to simulate the response of the system to a step reference of the desired velocity and a constant reference of the angle to which the car should turn towards. It calculates the norm of the velocity vector, returns it as feedback and also uses it to pass through subsystem1 that will return the different velocities at which each of the rear wheels must turn in order to achieve said angle.

5.1.1.3. Discrete PID

To control the system, a PID controller was used for the reason that it is the most used of all controllers in this type of cases and also leads to better system behavior because of the following statements : the proportional action defines the velocity of approach to the value in steady state; the integral action reduces the error in steady state by integrating the error; the derivative action anticipates the response, deriving the error, allowing to reduce the oscillatory behavior of the system response and its integration in the controller should always be analysed, principally in systems with a lot of noise, to avoid instability in the system.
The PID controller can be implemented either in analog or digital, however, the implementation it is usually digital using a system with a microprocessor associated. This method can also improve the performance by allowing control of actuator saturation(wind-up), even in systems with a lot of noise in the measured variables. Thus, the digital control system performs the sampling of inputs of interest, calculates the value of the control variable and then, if necessary, convert it to an analog value. The sampling introduces a delay in the system and keep the entry value between consecutive samples, leading to the concept of retainer. Normally, the Zero Order Holder (ZOH) is selected to model this effect in the control.

The analog PID controller equation is given by Eq. 5.11:

$$C(t) = C_{est} + K_c\left(e(t) + \frac{1}{\tau_i}\int e(t')dt' + \tau_d\frac{de(t)}{dt}\right) \tag{5.11}$$

- C(t): control action
- C_{est}: control action in steady state;
- K_c: proportional controller gain;
- τ_i: integral time constant;
- τ_d: derivative time constant;
- $r(t) = y_r(t) - y(t)$: error, given by the difference between reference and the output of the system.

The discrete PID controller equation is obtained by discretizing the Eq. 5.11 term to term in moments n and $n-1$, resulting in Eq. 5.12, called position algorithm:

$$C[n] = C_{est} + K_c\left(e[n] + \frac{T}{\tau_i}\sum_{i=0}^{n} e[i] + \tau_d\frac{e[n]-e[n-1]}{T}\right) \tag{5.12}$$

- C[n]: control action
- C_{est}: control action in steady state;
- K_c: proportional controller gain;
- τ_i: integral time constant [s];
- τ_d: derivative time constant [s];
- $e[n] = y_r[n] - y[n]$: error, given by the difference between reference and the output of the system for instant n.
- $e[n-1]$: error for instant n-1.
- T: sampling period [s]

Determing the equivalences for the controller's earnings in the most conventional one:

$$K_p = K_c;\ K_i = K_c\frac{T}{\tau_i};\ K_d = K_c\frac{\tau_d}{T} \tag{5.13}$$

There are other algorithms for PID control, namely the speed algorithm and the modified ones (position and speed). The speed algorithm calculates the variation of the controller output signal in relation to the immediately previous moment, presenting advantages over position algorithm as it does not need initialization and protects against eventual saturation of the controller (wind-up) and against computational failures. The modified algorithms aim to mitigate the derivate jump when there is an abrupt change in the error, by applying the derivate action to the measured variable instead of the error.

To the modified speed algorithm will then come:

$$\begin{aligned} C[n] = C[n-1] + K_c\Bigg((y_m[n-1] - y_m[n]) + \frac{T}{\tau_i}e[n] \\ + \tau_d\frac{-y_m[n] + 2y_m[n-1] - y_m[n-2]}{T}\Bigg) \end{aligned} \tag{5.14}$$

5.1.1.4. Optimal control parameters determination

As the controller in use is a discrete PID, the first step is to find the optimal parameters. For the first set of simulations, the aim is to find the values for the PID gains. As purpose of the car is to explore hazardous areas, the parameter K_d will be set as 0 due to the noise.

Resorting the matlab functionality to automatically tune the PID gains for the system, the outcome is the

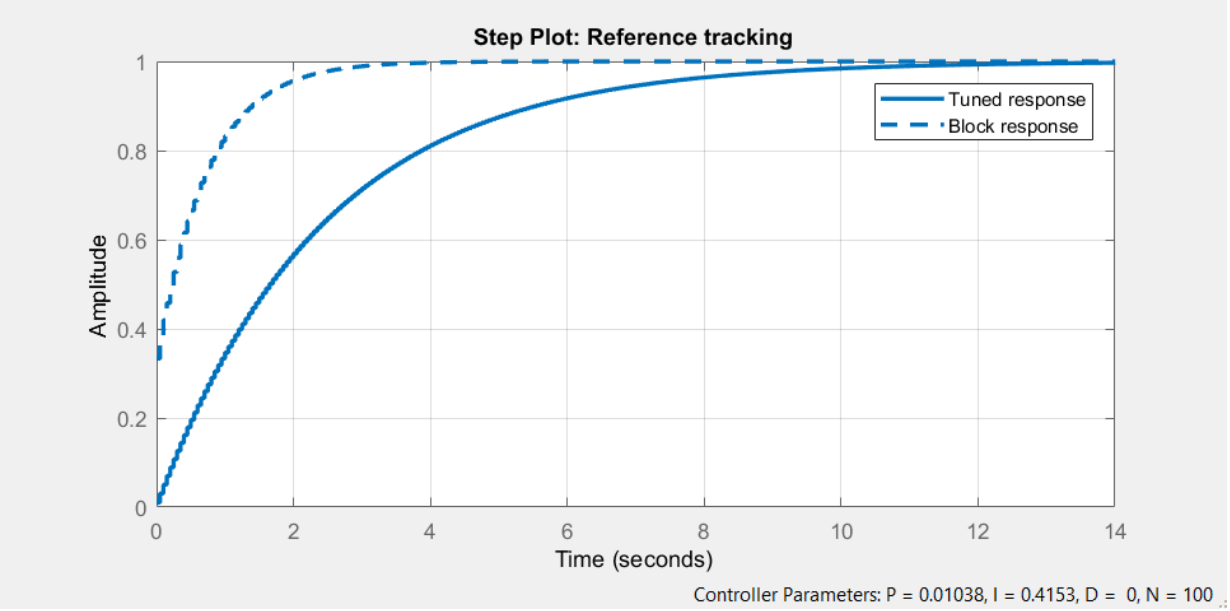

Figure 5.4.: Automatic matlab tune

present in the figure 5.4. It is possible to see that the velocity of the car reaches the reference value in approximately 11 seconds which is too long. The aim is for the car to get to the speed reference as fast as possible, so, in order for that to happen, new values of K_p and K_i have to be found in simulations. For the first simulation, $K_p = 1$, $K_i = 1$, Linear speed = 1 m/s, $\theta = 0$ rad: With the figure 5.5, it is possible to see that the initial value of the velocity of the car is 0.5 m/s and it take about 7 seconds to reach steady state. 0.5m/s as an initial value for the linear velocity is a considerably high value and can cause the car to slide, therefore, the K_p value needs to get lower, for the initial value to get lower as well. In the next simulation 5.6, K_p will be set as 0.5 and K_i as 1. Changing the value of K_p to half of the initial value, it is possible to see that the initial linear speed is now 0.3 m/s, making it harder for the car to slide. It takes nearly 6 seconds for the car to reach steady state, thus the K_i value must be increased. In the next simulation ??, K_p will be set as 0.5 and K_i as 2. As the K_p value was not changed, the initial linear speed value remains the same. As for the time it takes the car to reach steady state, it was reduced to 3 seconds due to the increase of the K_i parameter.

In ideal conditions, the aim would be for the car to reach the desired speed instantly, but due to the real

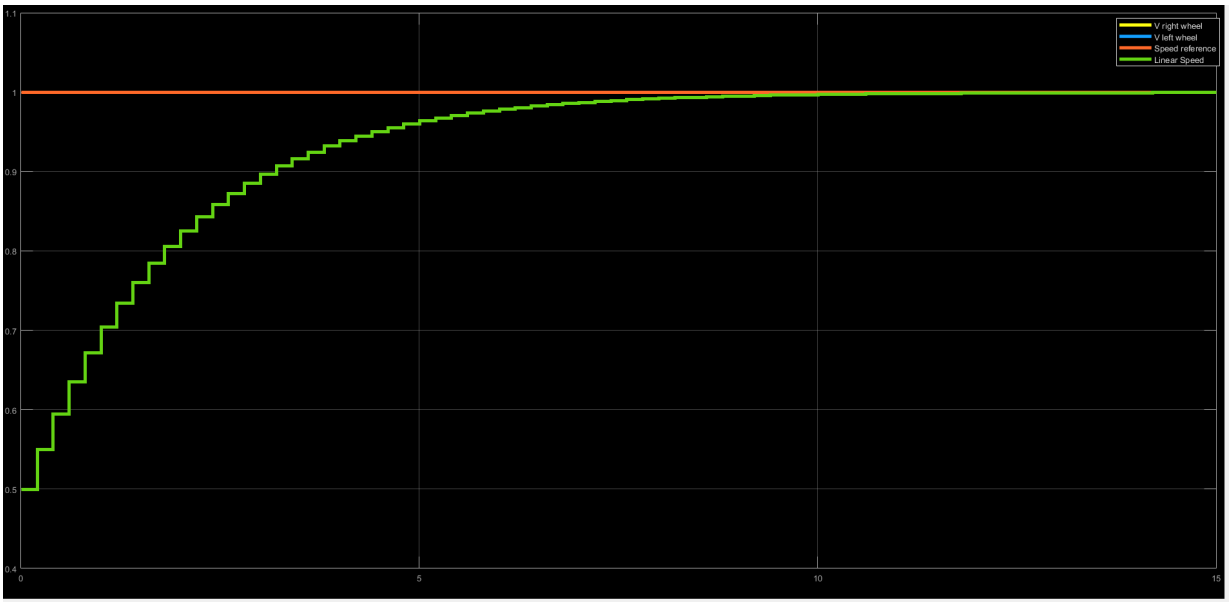

Figure 5.5.: $K_p = 1$, $K_i = 1$

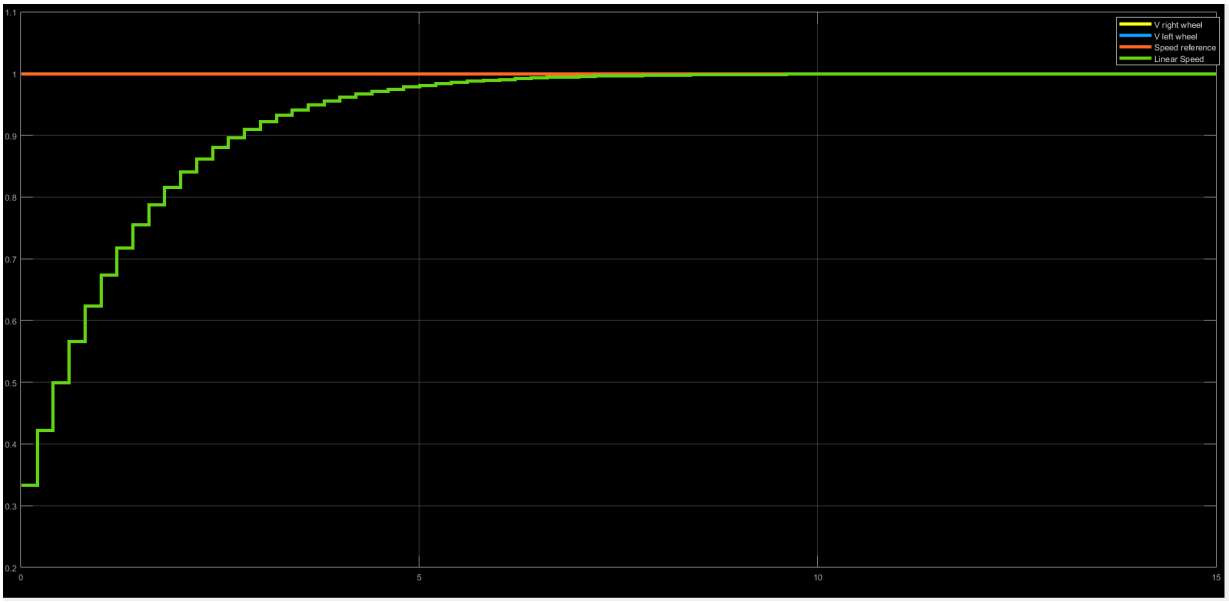

Figure 5.6.: $K_p = 0.5$, $K_i = 1$

limitations, such as wheel sliding, 3 seconds is a a safe value.

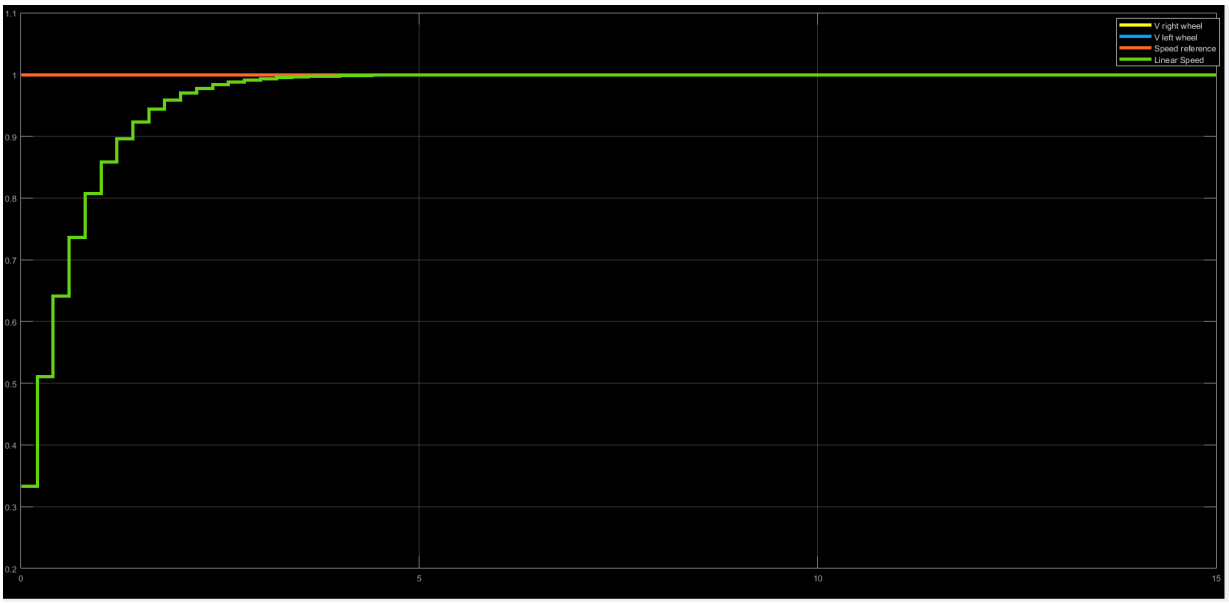

Figure 5.7.: $K_p = 0.5$, $K_i = 2$

5.1.1.5. Sampling time determination

In order to choose the sampling time, one needs to take in consideration that with the decrease of the sampling time, the processing overhead will increase and with the increase of the sampling time, the system response will have abrupt changes affecting the performance of the car. So, in order to accommodate both necessities, the sampling time will be set to 50ms.

5.1.1.6. System response

Having determined the parameters of the controller, the next step is to simulate the response of the system. In order to predict the behavior of the car to a linear speed reference and angle of tilt (teta), some simulations were necessary. The output of the simulation is the plot of the linear speed of the car, the linear speed of both wheels (left and right) and the position of the car. In order to plot the position of the car the Cartesian referential is used.
For the first simulation, the parameters were: Speed reference= 1m/s and $\theta = 0$ rad.
As expected,the figure 5.8 shows that the car linear velocity reached 1m/s. The angle of tilt is equal to 0 which means the car will be moving in a straight line, and as such, both wheels will are moving at the same speed of the car.

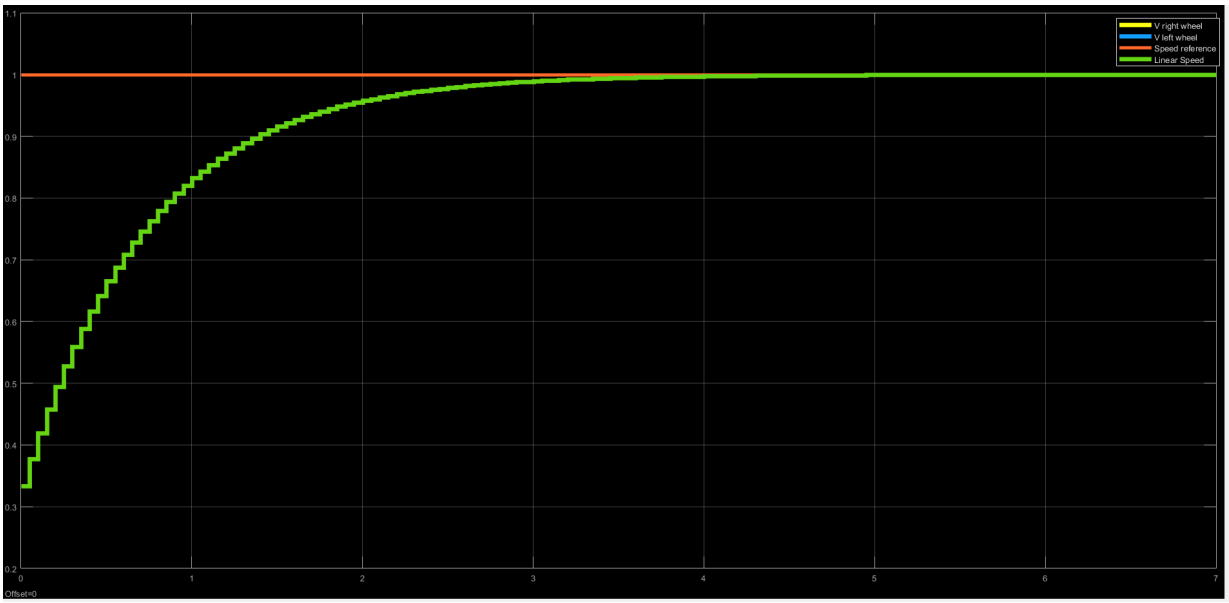

Figure 5.8.: Linear speed v=1m/s,$\theta = 0$ rad

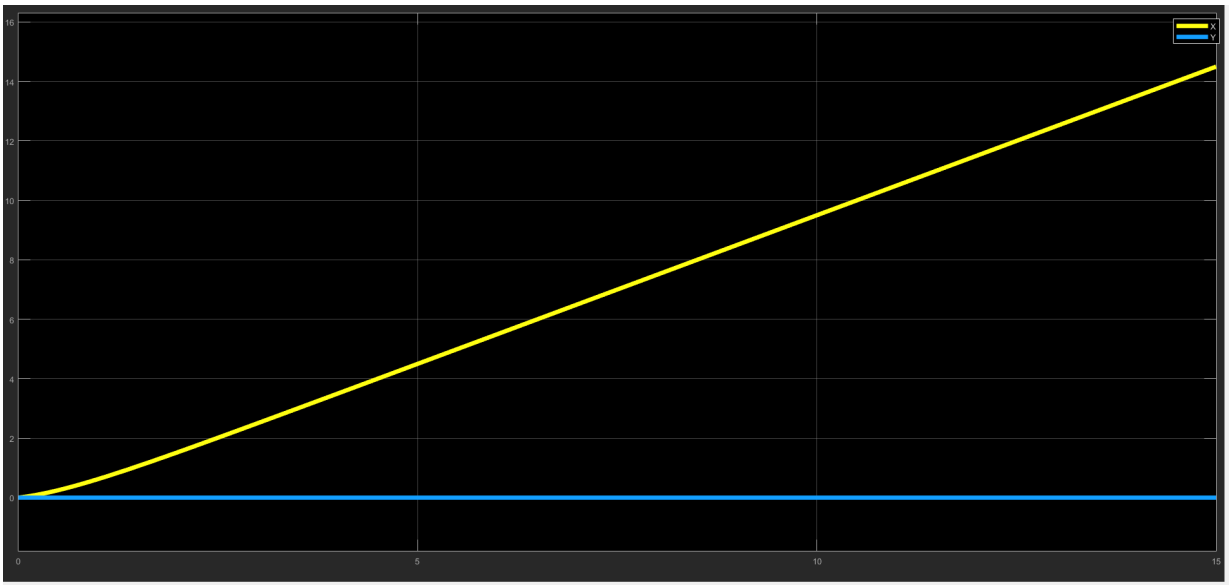

Figure 5.9.: Car position v=1m/s, $\theta = 0$ rad

As the teta is equal to 0 rad, only 1 coordinate of the car is moving, as the figure 5.9 demonstrates. The x coordinate is equal to 0 the entire simulation time, and the y coordinate is increasing with a linear scope equal to the linear velocity of the car. This implies that the car is indeed moving in a straight line.

Changing the teta to 0.1 rad to simulate constant tilt of the smart phone to the right, and maintaining the value of the speed reference in 1 m/s : In the simulation figure 5.10 it can be observed that having a

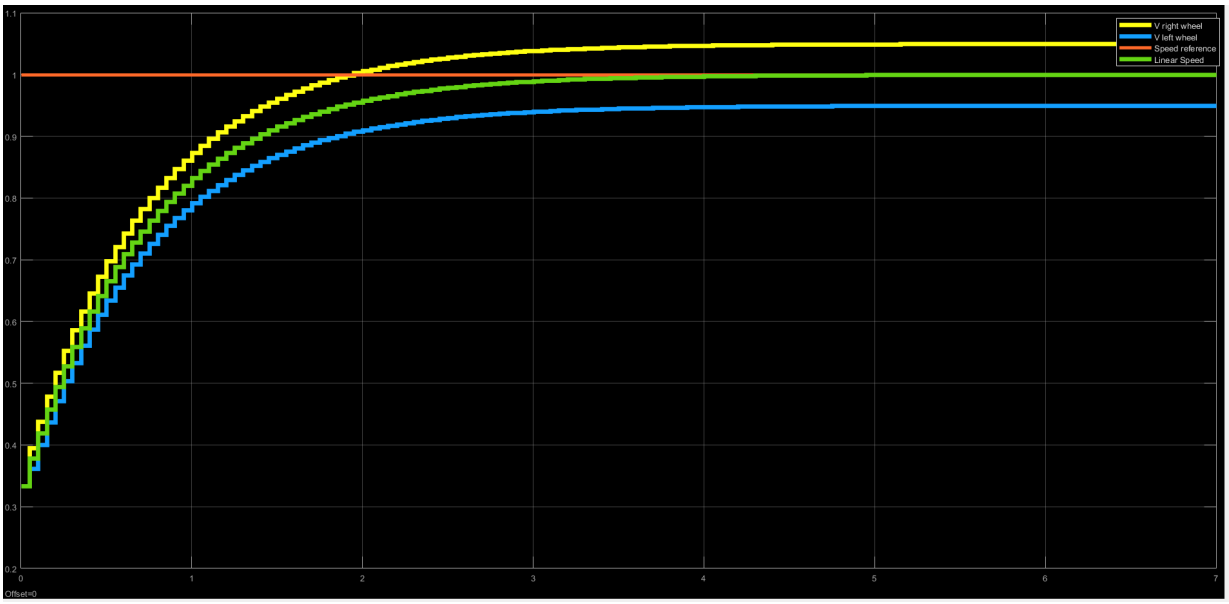

Figure 5.10.: Linear speed v=1m/s,$\theta = 0.1$ rad

angle of tilt not equal to 0, causes the left and right wheels to have different velocities, in order to make the car turn. Running more simulations with different values of teta, the outcome shows that the bigger the module of the value of teta, the bigger the difference between the linear velocities of the wheels. With this

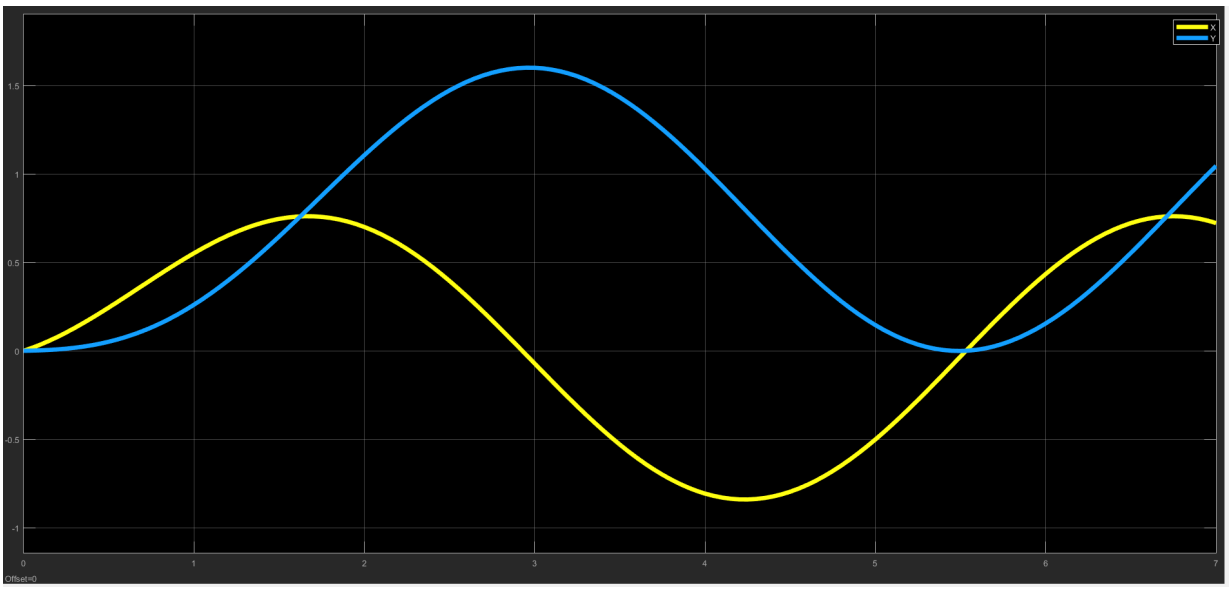

Figure 5.11.: Linear speed v=1m/s, $\theta = 0.1$ rad

figure 5.11 it is possible to observe that both the position of x and y of the car change with time. With a constant angle of tilt, the car will turn constantly in the same direction, eventually making a 360 degrees turn and as the car as small dimensions, it takes a very small time for it to do so, which is what is observed is this simulation.

5.1.2. System design

Tackling the objectives laid out in Section 5.1 requires an organized and well thought-out plan of work because there are a lot of smaller systems at play that need to work cohesively and synchronously.

The best way to achieve a good plan is by first separating the problem into packages, and in each package have subpackages, each with a collection of modules dedicated to serve a collective purpose in the data treatment and organization chain.

The system is divided into such packages as follows:

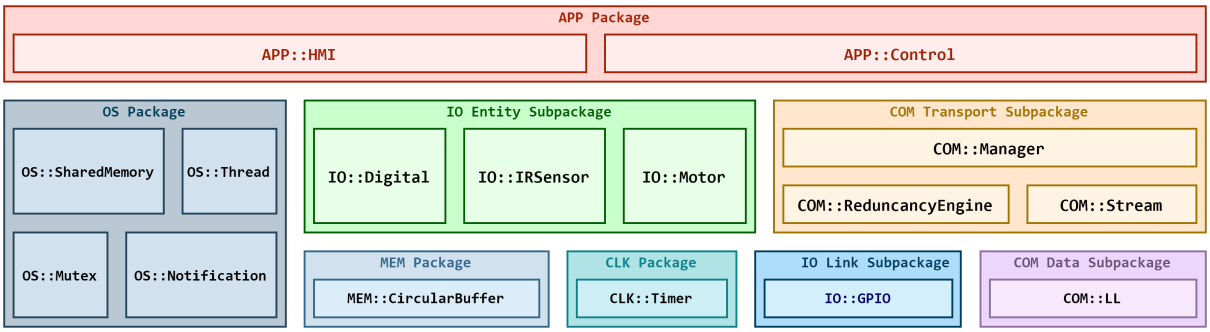

Figure 5.12.: Full Stack Overview

Each subpackage also belongs to a certain layer of software, characterized not by the resources it is associated to but by how close the modules within it are to the hardware. Futhermore, the packages should be distributed between layers in such a way that one seldomly need to use another that doesn't belong to the same layer or the layer directly below.

These layers are, from the bottom to the top:

- The High-level Hardware Abstraction Layer, which consists of the OS (partly), MEM, CLK, IO Link and COM Data packages/subpackages. Modules within this layer are responsible for the lower-level interaction with the system's resources so they are made to be thread-safe. Their implementation is platform-dependent and their interface is platform-agnostic. The main goal for this layer of software is to standardize the hardware, making for clean, maintainable and easily portable code.

- The High-level Software Abstraction Layer, consisting of the OS (partly), IO Entity and COM Transport packages. The modules within these packages should serve as an interface with the lower level layer, creating a more intuitive interaction process and mechanisms for processing information asynchronously. This way, when other modules make use of those interfaces, the information is already parsed and ready to be retrieved.
  Modules in the layers above ara also highly dependent on both abstraction layers to carry out their tasks in time so the ones in this layer should provide robust mechanisms for exception-handling and timing.

- The Main Application Layer, comprised only of the APP package, where the core functionality lies. As stated earlier, modules in this layer are protected from having to access to most lower-level layers but can use those modules and must use when no other abstraction is provided.

5.1.2.1. IO: Input/Output Package

The IO package is comprised of the IO Entity and IO Link subpackages. The modules within these subpackages are the parts of the hardware abstraction layers responsible for standardizing the General Purpose Input/Output resources of the machine.

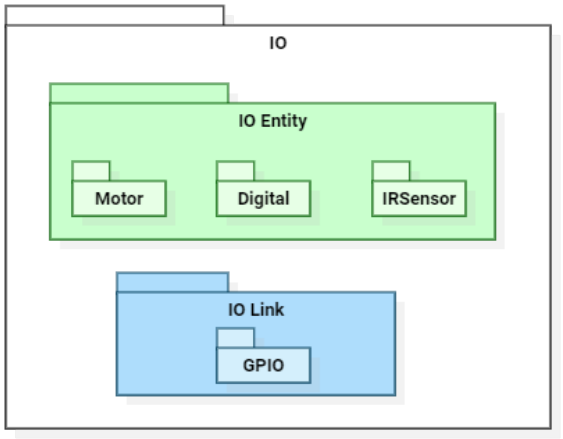

Figure 5.13.: IO Package Diagram

The IO Link subpackage is the interface provides the most generic yet complete package for interacting with the machine. It provides such functionality as automatic resource assignment, automatic buffering or timed output/input.

The IO Entity subpackage serves for specialization of functionality present in IO Link to serve a certain purpose attached to a physical meaning. This means it also provides methods for automatic calculation of real-world values based on the measurements taken or otherwise.

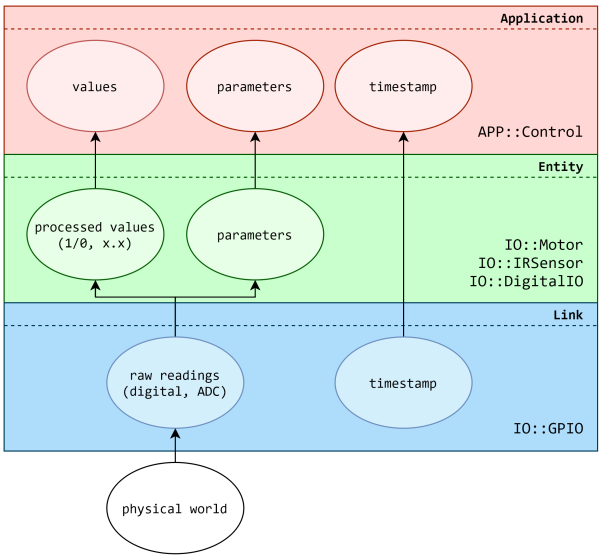

Figure 5.14.: IO subpackage interaction and information propagation diagram

5.1.2.2. COM: Communications Package

The COM package is the sum of the COM Data and COM Transport subpackages. These modules are responsible for standardizing the access to the inter-device communication resources of each machine.

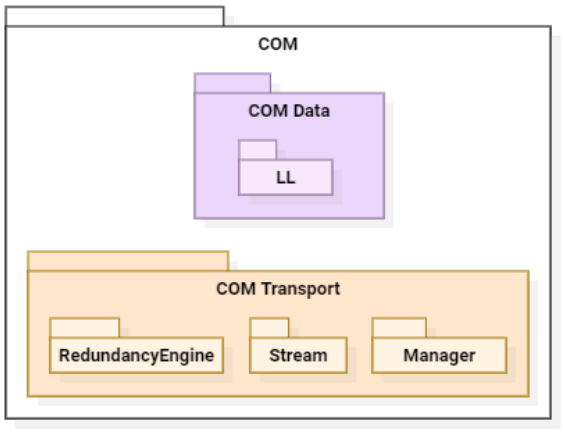

Figure 5.15.: COM Package Diagram

The COM Data subpackage is aimed at providing a platform-agnostic interface for communicating over a serial or Bluetooth connection, while also providing specific functionality for different protocols/roles. The COM Transport subpackage provides tools for managing multiple simultaneous, redundant, multiprotocol and multi-stream connections as well as automatically parsing of data for usage in time-constrained scenarios.

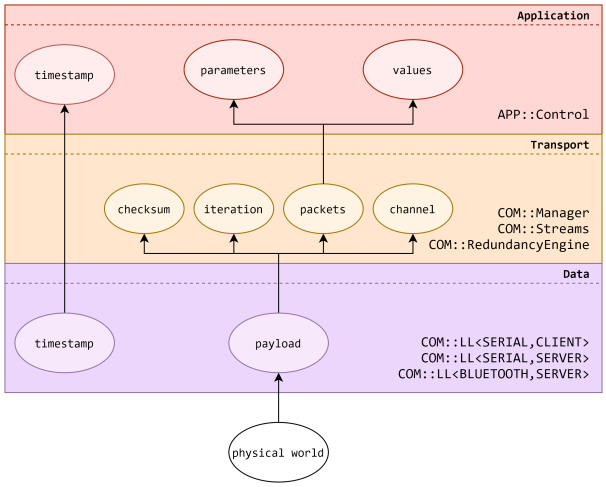

Figure 5.16.: COM subpackage interaction and information propagation diagram

5.1.2.3. OS: Scheduler Package

The modules in the OS package mainly serve the purposes of thread creation and management management, inter-thread synchronization and thread-safe access to memory.

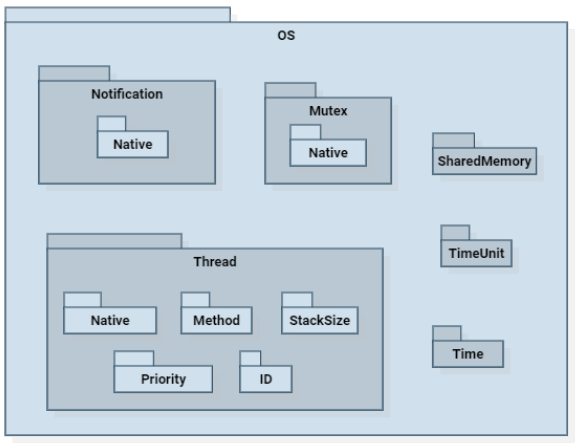

Figure 5.17.: OS Package Diagram

5.1.2.4. MEM: Memory Structures Package

5.1.2.5. CLK: Timing Package

5.1.2.6. APP: Main Application Package

5.2. Physical Environment Virtual Subsystem

5.3. Remote Vision Virtual Subsystem

The Remote Vision Virtual Subsystem (RVVS) is mainly responsible for providing visual feedback to the user of the vehicle's surroundings to assist its navigation. Additionally, it should offload the Navigation Virtual Subsystem (NVS) from other intensive, but no so critical, tasks — like telemetry data — as well as provide redundant paths for communications, especially in the most critical conditions, like off-grid navigation with the inclusion of the GPRS module. Thus, as aforementioned and illustrated in Fig. 4.2, it interfaces three different subsystems: NVS via RS232 communication; smartphone via Wi-Fi or GPRS; and the web camera.

The RVVS design was divided into three phases, corresponding to the functional, object and dynamic models.

5.3.1. Functional model

The functional model describes the functionalities of the system from the actors' perspective, illustrated in Fig. 5.18 by an use case diagram. Two actors were identified: NVS and Smartphone. The Smartphone acts a proxy for user interaction, thus for clarity purposes, the actor was named User. Three main set of features were identified, namely:

1. Communication: deals with the communications interfaces between the various subsystems, further decomposed into the required functionalities for each interface (Comm Functions). The User may communicate with the Remote Vision Virtual Subsystem (RVVS) subsystem via GPRS or Wi-Fi, thus requiring the latter to provide network discovery capabilities, alongside with the conventional connect, send, receive and disconnect functionalities. Additionally, it may also required periodic KEEP ALIVE pings to check if the connection is still on. The NVS communicates via RS232, which does not require network discovery. Lastly, the user may require sensor information, which could potentially be connected to the RVVS subsystem (e.g., in a Inter-Integrated Circuit (I2C) network) for offloading the NVS subsystem.

2. Image Acquisition: responsible for providing visual feedback to the user. It can be configured, started, stopped and captured.

3. Processing: responsible for processing: user commands for image acquisition or forwarding them to the NVS subsystem as a redundant path; telemetry data, such as, overall distance traversed, maximum speed so far, and operation time.

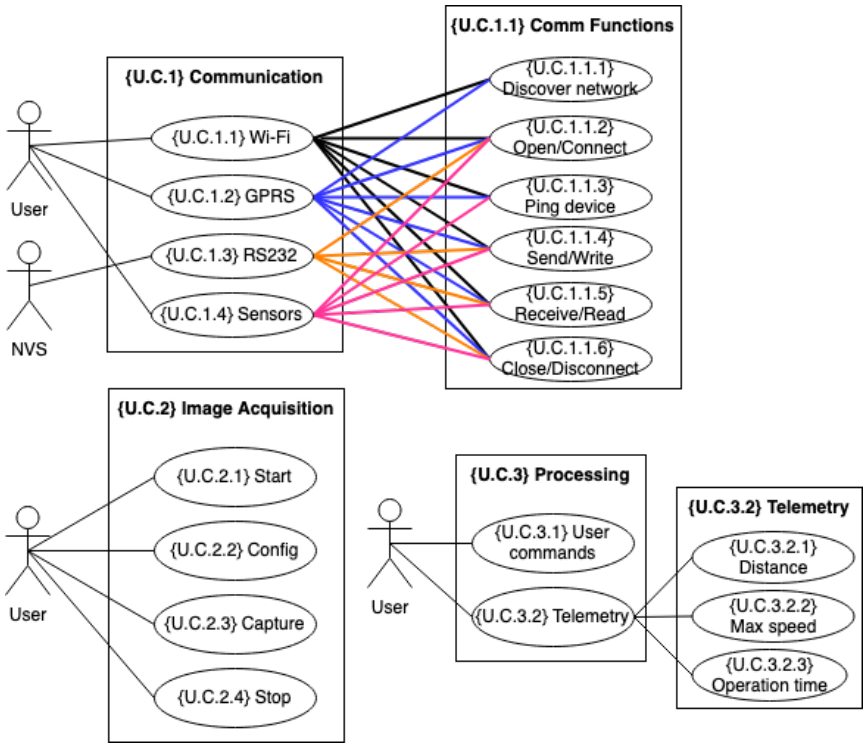

Figure 5.18.: Use case for RVVS subsystem

5.3.2. Dynamic model

The dynamic model describes the internal behaviour of the system, represented in UML by sequence, state-machine, and activity diagrams.

The sequence diagram represents the sequence of events related to a particular use case. It aids to refine use cases and identify software objects (entity, boundary, control), providing a well-established path for the implementation of system's functionalities.

On the other hand, state-machine diagrams provide an overview of the system functionalities and how they interact, i.e., the system's states, transitions, and response to stimulus, internal or external. This

makes it an excellent tool for designing overall system behaviour. Fig. 5.19 identifies the main elements used in the state-machine diagram.

The state-machine diagram for the RVVS subsystem is depicted in Fig. 5.20. On system startup is performed an initialization procedure, loading user-defined and machine settings and initializing the required hardware. After initialization is completed, four main processing units are executed in parallel until system is shutdown, namely:

- Wi-Fi Manager: manages Wi-Fi connection;
- RS232 Manager: manages RS232 connection;
- Main: idle processing unit; parses messages received through the available communications channels and, if a command is detected, pushes it to a command queue for later execution.
- Scheduler: periodically executed processing unit, responsible for executing periodical tasks and spawning tasks as a response to issued commands, with different priority levels.
  - Task: spawned as a response to a command, performs the required associated function and has a low expected lifetime. The commands can be classified as user, NVS, telemetry data, image acquisition, or communications.

Lastly, an example of a more refined state-machine diagram, in this case for communications, is illustrated in Fig. 5.21. In the initial state, a connection request is expected and if accepted, authentication is requested. If successfully authenticated, the Communication Manager is ready to communicate, handling incoming/outgoing data by notifying the relevant entities and performing the associated operations. The RS232 does not require connection and authentication management, thus yielding only the states **readyToCommunicate**, **MsgReceived**, and **MsgToSend**.

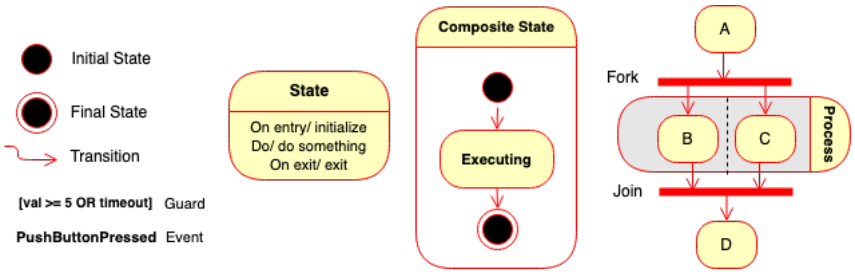

Figure 5.19.: State-machine diagram

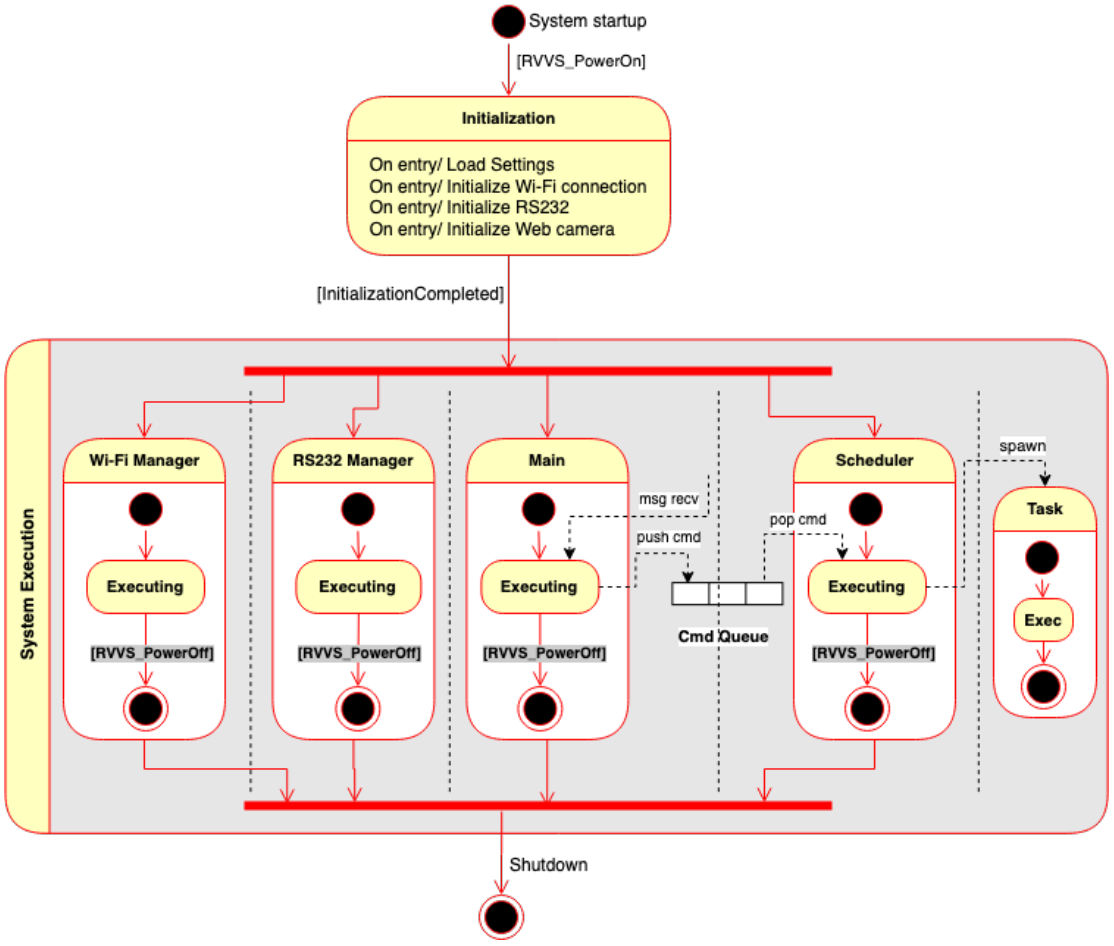

Figure 5.20.: RVVS state-machine diagram

5.3.3. Subsystem decomposition

Following the same design rationale indicated in Section 5.1.2 for the NVS subsystem, the the RVVS system should be decomposed into smaller, more tractable, subsystems for easier development. Furthermore, modularity and reuse, whenever is possible, should be key design guidelines.

The subsystem decomposition for RVVS is illustrated in Fig. 5.21, taking into account these considerations. Thus, **OS**, **COM Transport**, and **COM Data** packages are reused from NVS for the following reasons: RVVS requires concurrent/parallel processing as provided by the **OS** package in a thread-safe way; it shares the communication interface with NVS (RS232), as provided by the **COM** packages. The same idea is extensible to the Wireless Communication (**WCOM** package), with the transport layer comprised of a **Manager**, a **RedundancyEngine** and a **Stream** for packet serialization/deserialization, and a low-level layer dependent of the technology used (Wi-Fi/GPRS). For **Image Acquisition**, a **Manager** handles image acquisition functionalities, with the **Frame** acting as image data container; low-level layer **Img::LL** handles the hardware

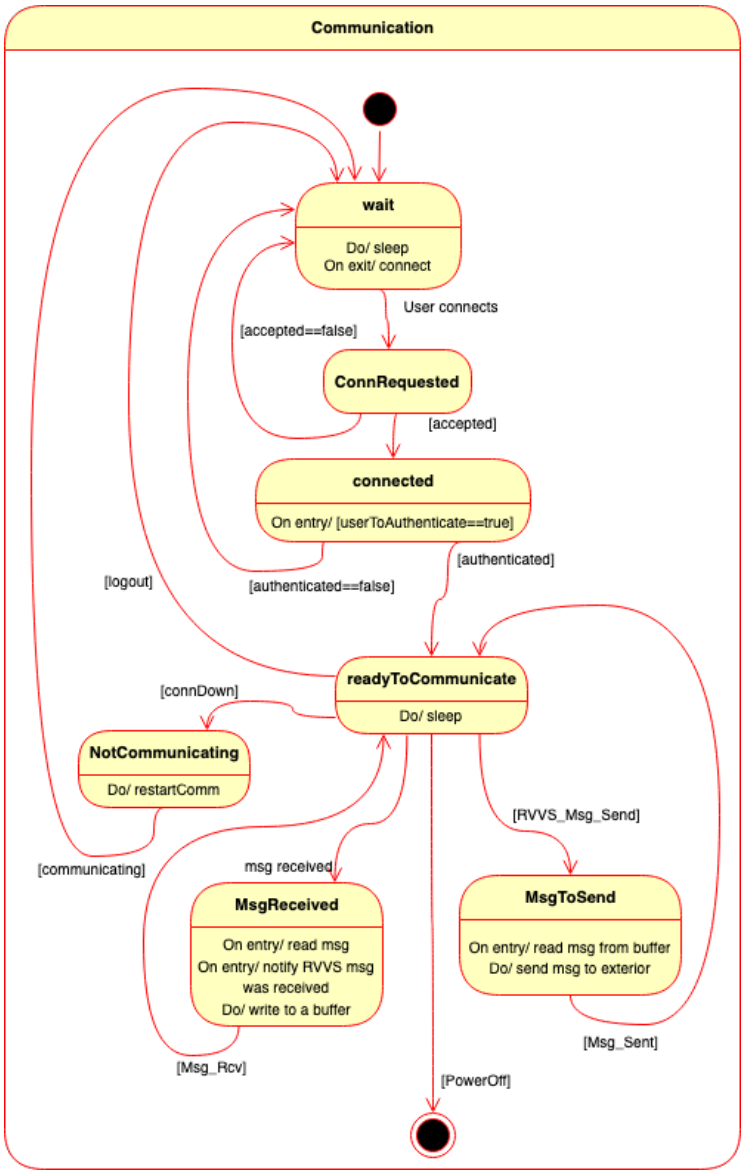

Figure 5.21.: Communication state-machine diagram

interface and low-level details. Completing the system stack is the **Telemetry** package, responsible for managing the telemetry data as required. This package could have a direct interface with the sensors, but, for the time being, it tracks information received by the NVS subsystem.

At the top of the software stack is the actual software executed on top of the system stack, containing the application logic and management. Thus, the **App::Manager** package is a event-driven processing unit, listening for relevant events signalled by the layers below and handling those events, as the one generated by successfully parsing of commands arriving from the available communications channels.

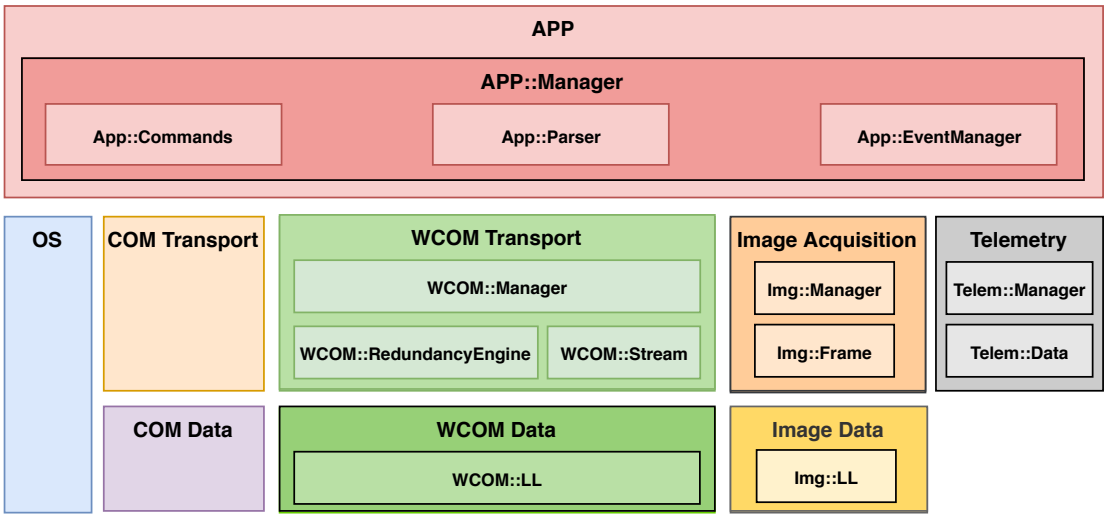

Figure 5.22.: RVVS full stack overview

5.3.4. Object model

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

5.4. Smartphone

The smartphone design was divided into three stages, the functional model, the object model and the dynamic model.

5.4.1. Functional Model

The functional model includes a description of the system in terms of accessible functionalities from the actors' point of view, represented in figure 5.23. In this stage, the actors identified were the user, the NVS and the RVVS. The user must have access to three key features:

- Vehicle control: the ability to control the car by tilting the smartphone and thereby changing the provided angle and speed reference. This feature implies a transmission of these values to the NVS and a UI value update.
- Notification view: pop-ups that allow a grasp of the current state of the system with informative or alert messages.

- Video feed view: Be able to visualize the RVVS‘ video transmission on screen.

On one hand, the NVS should be able to receive the accelerometer data provided while also sending notifications to the app, both via Bluetooth. On the other hand, the RVVS should be capable of transmitting the camera´s video while also sending the same type of notifications to the user within the application. This communication is established via Wifi/GPRS.

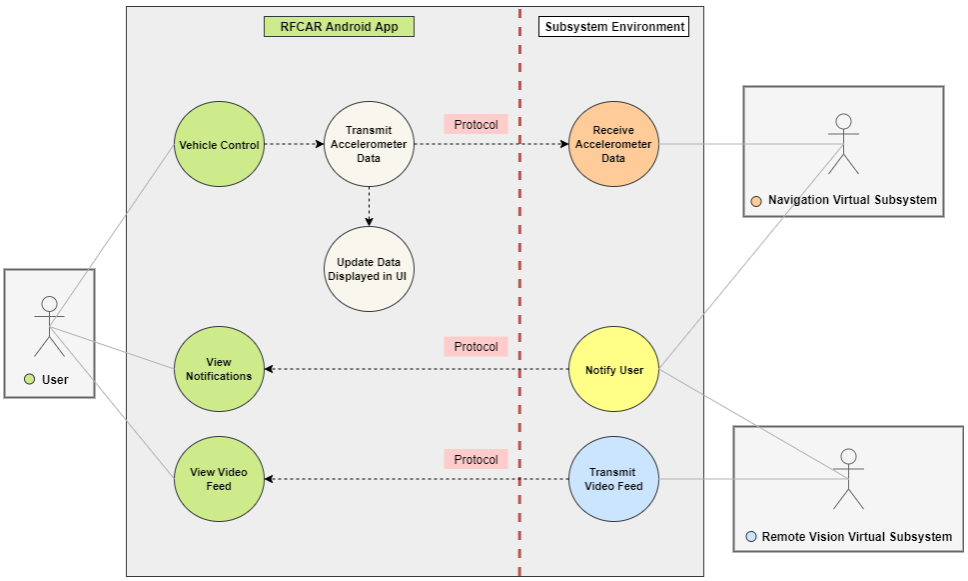

Figure 5.23.: Android app use case diagram.

5.4.2. Object Model

In this step, the objective was to define the app system´s structure with UML class diagrams, concerning its objects, attributes, operations and associations established. The first, of the forementioned diagrams, is represented in figure ??.

5.4.3. Dynamic Model

For the dynamic model, were devised multiple state-machine diagrams to describe the internal behaviour of the application. Figure 5.24 illustrates the overall application behaviour. One can observe that all the intended features are meant to run in parallel, that will surely affect the system implementation in terms of concurrency (section 2.3) and thread management. Initially, the system loads the User Interface (UI) and waits for all connections to be established before moving to the next state. While the feature for vehicle

control is running (figure 5.25) it retrieves the accelerometer data, sends the data to the NVS and displays it on the screen. Additionally, the feature that allows the user to see the video feed transmitted by the RVVS in figure 5.26 also displays it on the app screen. When this Wifi/GPRS connection is suspended by any means, it should exist an immediate reconnection to the RVVS. The notifications presented to the user should indicate the current state of the system, which means displaying informative messages, like successful connections but also alert messages like the ones displayed in figure 5.27.

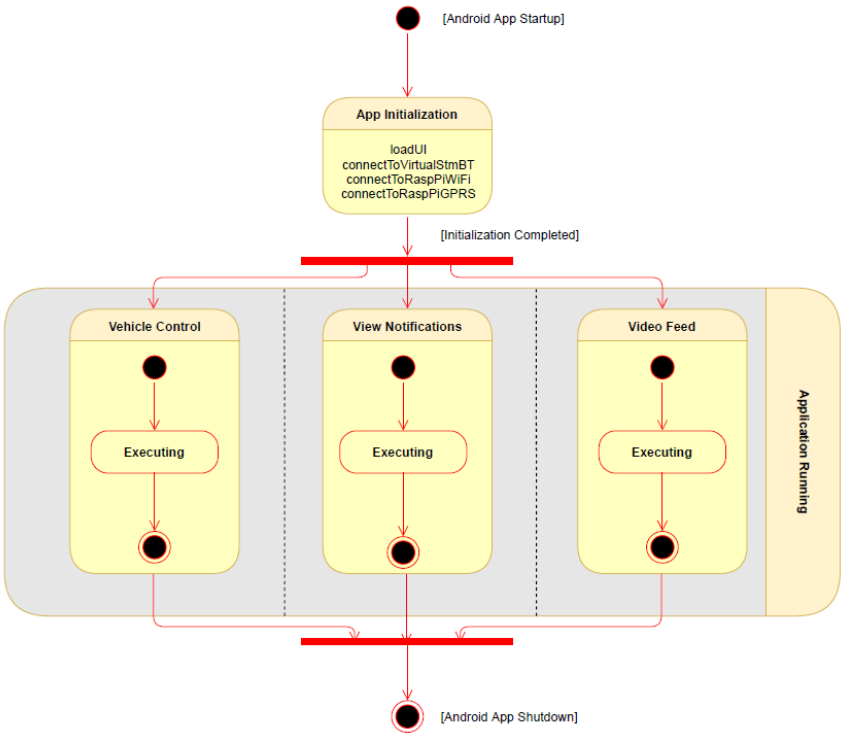

Figure 5.24.: Overall system behaviour diagram.

5.5. Hardware/Software mapping

In the hardware/software mapping activity, the functionality associated to a software component is mapped to the hardware responsible for executing it. It is represented by a UML deployment diagram used to depict the relationship between run-time components and nodes. Components are self-contained entities that provide services to other components or actors, e.g., the **RVVS_App** in Fig. 5.28. A node is a physical device or an execution environment in which components are executed, such as a desktop computer, or **myLinux** in Fig. 5.28, represented by boxes containing component icons. Furthermore, a node can contain another node, for example a device can contain an execution environment such as virtual machine.

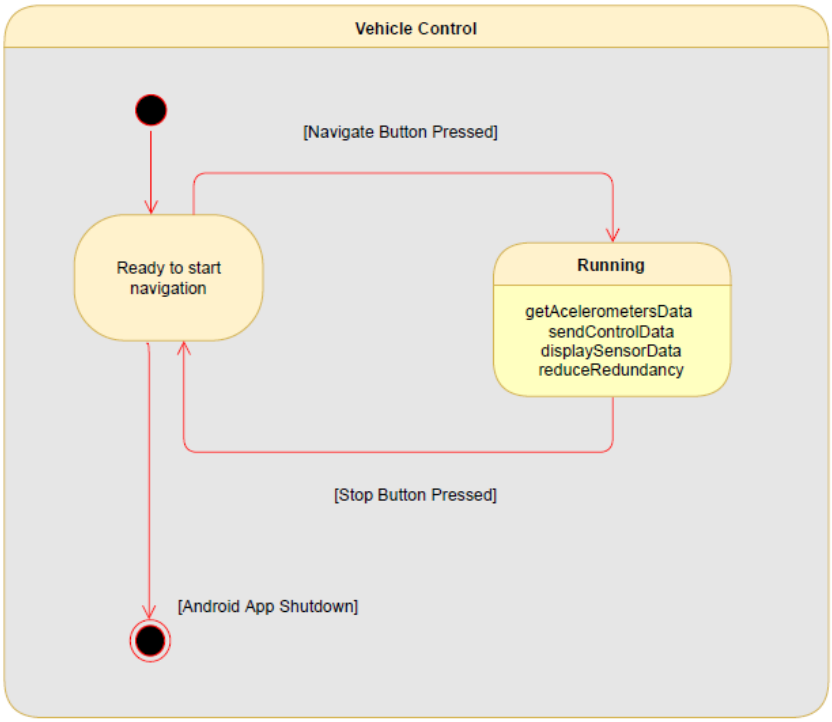

Figure 5.25.: Vehicle control feature diagram.

The UML deployment diagram for the RFCAR system is depicted in Fig. 5.28, where the two nodes — Smartphone and LinuxVM — are represented in turquoise and orange, respectively. The ball-and-socket joint identifies the provided and required interfaces, respectively. Thus, the **RVVS_app** provides wireless communication interfaces that the **Android_app** can use, namely Wi-Fi and GPRS . Furthermore, it highlights the client-server software pattern, with the **RVVS_app** and **NVS_app** serving the requests (servers) that the **Android_app** requests (client). The **Android_app** communicates with the **NVS_app** via Bluetooth, or, if the communications fail, through any of the available communications channels for the **RVVS_app** that forwards that request for **NVS_app**.

Lastly, it should be noted that in a real-world scenario the **NVS_app** and **RVVS_app** are mapped to a different hardware, which is also distinct between them, e.g., the former in an **STM32** and the latter in a **Raspberry Pi**. However, in the virtualized environment, and ideally, they represent two distinct processes that must communicate via an IPC mechanism, e.g., sockets. Nonetheless, to ease and speed up the development it is perfectly acceptable to implement both processes into the same application in the early development phase.

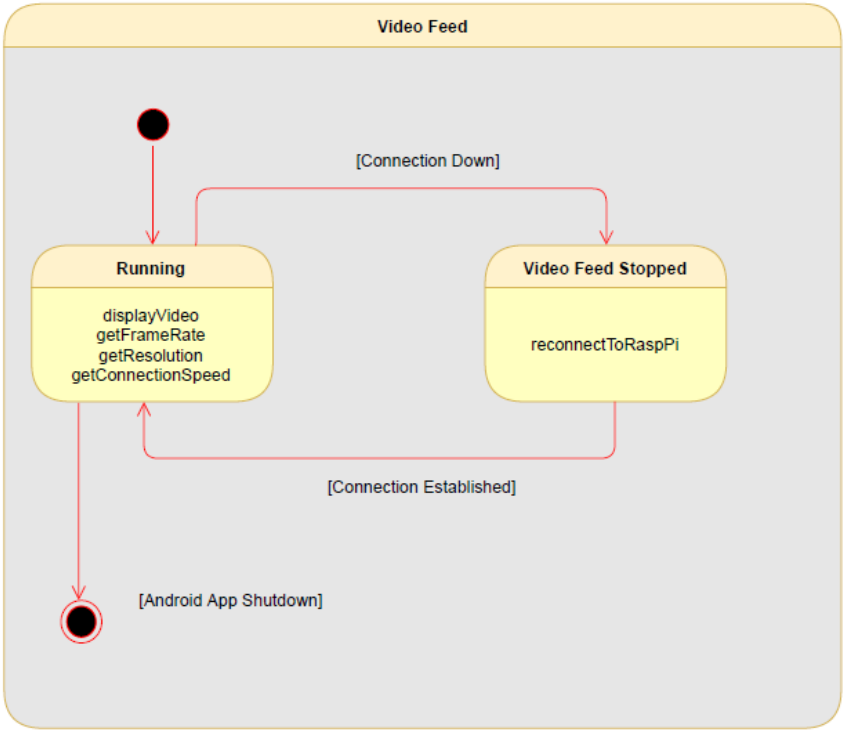

Figure 5.26.: Video feed feature diagram.

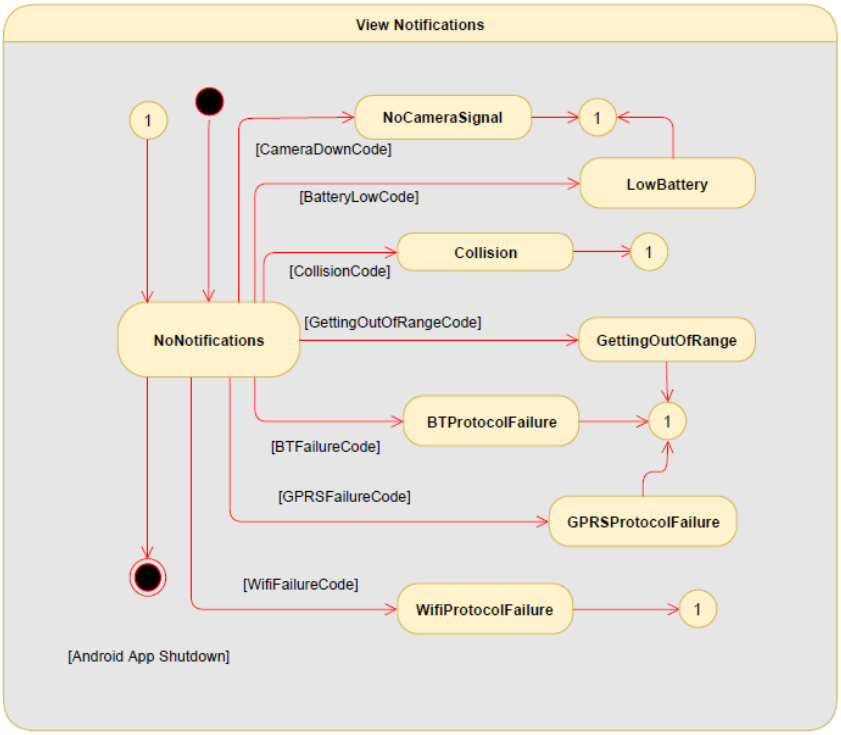

Figure 5.27.: Notification feature diagram.

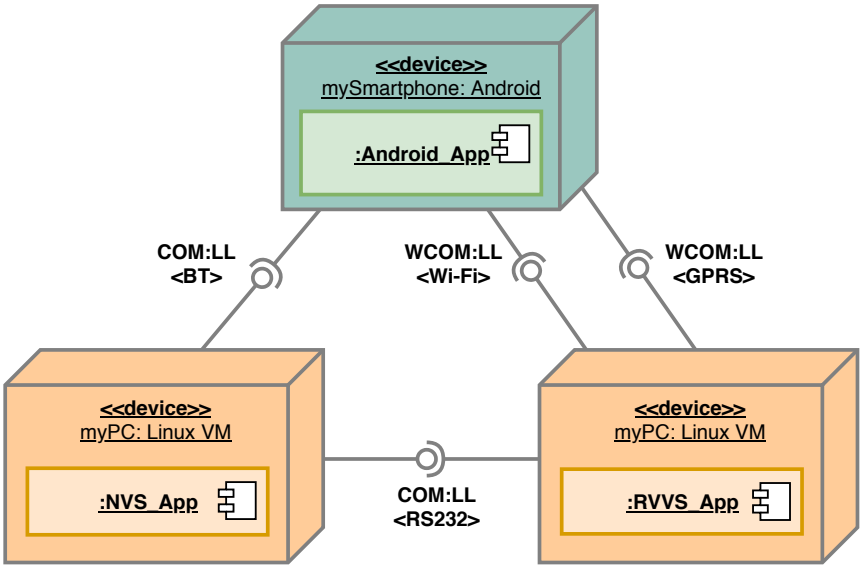

Figure 5.28.: RFCAR Deployment diagram

6. Implementation

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

6.1. Navigation Virtual Subsystem

6.1.1. Control

6.2. Physical Environment Virtual Subsystem

6.3. Remote Vision Virtual Subsystem

6.4. Smartphone

The mobile OS chosen for this project was Android. Usually, android apps can be implemented using Kotlin, Java and C++. For the RFCAR project, the language chosen was Java due to the knowledge gained in prior course years where the need to implement android-targeted applications rose. Additionally, the code was conceived using the Android Studio Integrated Development Environment (IDE). One must notice that despite the user-friendliness of the development environment through context-sensitive guidelines and code suggestions, this language and IDE are not the best in terms of full system control due to the multiple abstraction layers preemptively defined. Therefore, one might wonder why the C++ route with a cross-platform framework like QT wasn't selected. Multiple options were taken into account at this stage and its a fact that the one that ended up being selected might not deliver as much implementation freedom as the second option forementioned. However, it allowed deadline fulfilment and code reuse notwithstanding the additional effort put into delving deeper within some contexts.

6.4.1. Accelerometer Interaction

The smartphone has a built-in MEMS accelerometer, this means that at micro-level it can measure acceleration values through capacitance changes in multiple capacitors as a result of its internal assembly (calibration mass and spring contacts) displacement. With at least a MEMS system in each plane (x,y,z), one can measure the acceleration per axis.

6.4.1.1. Accelerometer Data Retrieval

The code in listing 6.1 (based on [?]) represents the retrieval of the linear acceleration for each plane. In the first place, the definition of the sensor type is crucial to access its values (line 9). SensorManager grants access to the sensors of the android device (line 7). Following, one must create a listener that checks the sensors values at a determined sampling frequency using the SensorManager's method registerListener (line 10). Upon doing so, one overwrites the onSensorChanged method (line 15) so the pretended variables that hold the values of the sensor can only be updated on smartphone movement. An acceleration sensor measures the acceleration applied to the device but regarding the force of gravity. For this reason, the values retrieved do not represent the linear accelerations for each plane. To resolve this problem, a low pass filter can be used to isolate the force of gravity in each axis (line 28) and then remove its contibution from the acceleration values (line 33).

```
1  TextView x_val, y_val, z_val;
2  private float sensorX, sensorY, sensorZ;

4  ...

6  Log.d(TAG, "onCreate: Initializing Sensor Services");
7  sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

9  accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
10 sensorManager.registerListener(MainActivity.this, accelerometer,
       SensorManager.SENSOR_DELAY_NORMAL);
11 Log.d(TAG, "onCreate: Registered accelerometer listener");

13 ...
```

```
15 @Override
16     public void onSensorChanged(SensorEvent event) {
17         Sensor mysensor = event.sensor;
18         if(mysensor.getType() == Sensor.TYPE_ACCELEROMETER) {

20             final float alpha = (float) 0.8;
21             float gravityX = 0;
22             float gravityY = 0;
23             float gravityZ = 0;
24             float linear_accelerationX = 0;
25             float linear_accelerationY = 0;
26             float linear_accelerationZ = 0;

28             // Isolate the force of gravity with the low-pass filter.
29             gravityX = alpha * gravityX + (1 - alpha) * event.values[0];
30             gravityY = alpha * gravityY + (1 - alpha) * event.values[1];
31             gravityZ = alpha * gravityZ + (1 - alpha) * event.values[2];

33             // Remove the gravity contribution with the high-pass filter.
34             linear_accelerationX = event.values[0] - gravityX;
35             linear_accelerationY = event.values[1] - gravityY;
36             linear_accelerationZ = event.values[2] - gravityZ;

38             Log.d(TAG, "onSensorChanged: X: " + linear_accelerationX + " Y
                   : " + linear_accelerationY + " Z: " + linear_accelerationZ)
                   ;

40             x_val.setText(String.format("X: %s", linear_accelerationX));
41             y_val.setText(String.format("Y: %s", linear_accelerationY));
42             z_val.setText(String.format("Z: %s", linear_accelerationZ));

44             long currTime = System.currentTimeMillis();

46             if((currTime - lastSensorUpdateTime) > 50){
```

```
47                  lastSensorUpdateTime = currTime;

49                  sensorX = linear_accelerationX;
50                  sensorY = linear_accelerationY;
51                  sensorZ = linear_accelerationZ;
52              }
53          }
54      }
```

Listing 6.1: Accelerometer data retrieval code

6.4.1.2. Applying Accelerometer Data

The project requires that the acceleration values obtained from the accelerometer sensor are applied to make the vehicle move in the intended direction with the expected speed. To implement the code referent to this sub-subsection, one must pay close attention to the axis orientation in a common device, represented in figure 6.1. In order to test this concept, the code presented in 6.4.1.1 was added to another project that

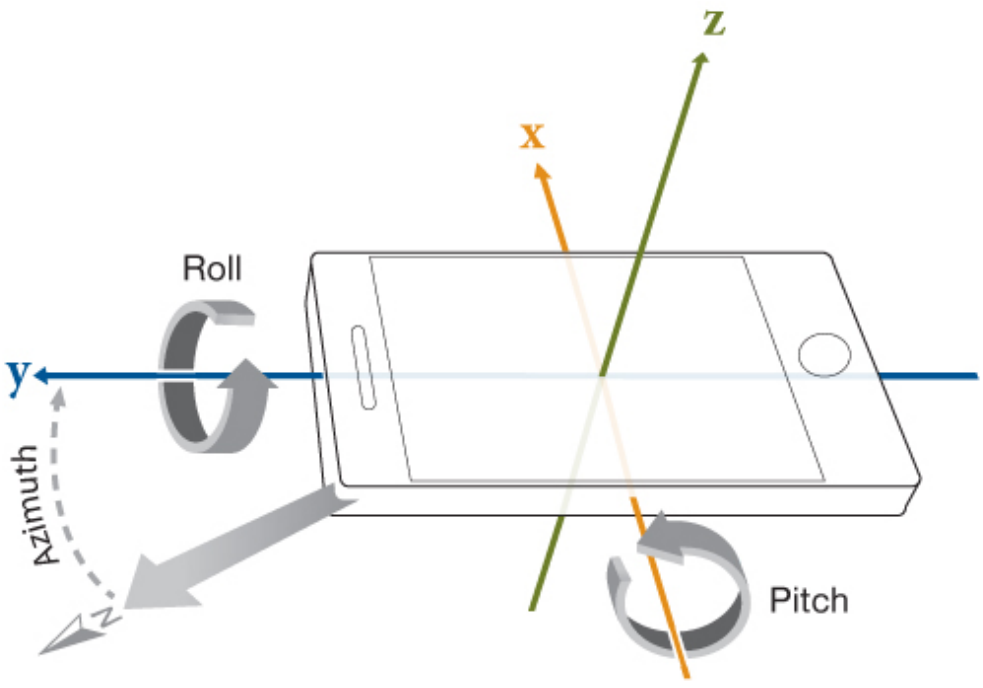

Figure 6.1.: Axis orientation in a smartphone

allowed ball movement based on the accelerometer values. It must be noticed that the z acceleration value it's not relevant to the ball movement (from line 59 to 67) since only roll and pitch affect a 2D (bidimensional) object and the smartphone height relative to the ground doesn't, as one should expect by analysing figure 6.1. This code in listing 6.2 representing subsection 6.4.1 will be tested in 7.1.4.3.

```
1  public class MainActivity extends AppCompatActivity implements
       SensorEventListener {

3      private static final String TAG = "MainActivity";

5      private SensorManager sensorManager;
6      Sensor accelerometer;

8      TextView x_val, y_val, z_val;
9      private float sensorX, sensorY, sensorZ;

11     private CanvasView canvas;
12     private int circleRadius = 30;
13     private float circleX, circleY;

15     private Timer timer;
16     private Handler handler;
17     private long lastSensorUpdateTime = 0;

19     @Override
20     protected void onCreate(Bundle savedInstanceState) {
21         super.onCreate(savedInstanceState);
22         setContentView(R.layout.activity_main);

24         x_val = (TextView) findViewById(R.id.xValue);
25         y_val = (TextView) findViewById(R.id.yValue);
26         z_val = (TextView) findViewById(R.id.zValue);

28         Log.d(TAG, "onCreate: Initializing Sensor Services");
```

```
29          sensorManager = (SensorManager) getSystemService(Context.
                SENSOR_SERVICE);

31          accelerometer = sensorManager.getDefaultSensor(Sensor.
                TYPE_ACCELEROMETER);
32          sensorManager.registerListener(MainActivity.this, accelerometer,
                SensorManager.SENSOR_DELAY_NORMAL);
33          Log.d(TAG, "onCreate: Registered accelerometer listener");

35          Display display = getWindowManager().getDefaultDisplay();
36          Point size = new Point();
37          display.getSize(size);

39          int screenWidth = size.x;
40          int screenHeight = size.y;

42          circleX = (screenWidth >> 1) - circleRadius;
43          circleY = (screenHeight >> 1) - circleRadius;

45          canvas = new CanvasView(MainActivity.this);
46          setContentView(canvas);

48          handler = new Handler(){
49              public void handleMessage(Message message){
50                  canvas.invalidate();
51              }
52          };

54          timer = new Timer();
55          timer.schedule(new TimerTask() {
56              @Override
57              public void run() {

59                  if(sensorX > 0)
```

```
60                      circleX -= 10;
61                  else
62                      circleX += 10;
63
64                  if(sensorY > 0)
65                      circleY += 10;
66                  else
67                      circleY -= 10;
68
69                  handler.sendEmptyMessage(0);
70              }
71          }, 0, 50);
72      }
73
74      @Override
75      public void onAccuracyChanged(Sensor sensor, int i){}
76
77      @Override
78      public void onSensorChanged(SensorEvent event) {
79          Sensor mysensor = event.sensor;
80          if(mysensor.getType() == Sensor.TYPE_ACCELEROMETER) {
81
82              final float alpha = (float) 0.8;
83              float gravityX = 0;
84              float gravityY = 0;
85              float gravityZ = 0;
86              float linear_accelerationX = 0;
87              float linear_accelerationY = 0;
88              float linear_accelerationZ = 0;
89
90              // Isolate the force of gravity with the low-pass filter.
91              gravityX = alpha * gravityX + (1 - alpha) * event.values[0];
92              gravityY = alpha * gravityY + (1 - alpha) * event.values[1];
93              gravityZ = alpha * gravityZ + (1 - alpha) * event.values[2];
```

```
95              // Remove the gravity contribution with the high-pass filter.
96              linear_accelerationX = event.values[0] - gravityX;
97              linear_accelerationY = event.values[1] - gravityY;
98              linear_accelerationZ = event.values[2] - gravityZ;
99
100             Log.d(TAG, "onSensorChanged: X: " + linear_accelerationX + " Y
                    : " + linear_accelerationY + " Z: " + linear_accelerationZ)
                    ;
101
102             x_val.setText(String.format("X: %s", linear_accelerationX));
103             y_val.setText(String.format("Y: %s", linear_accelerationY));
104             z_val.setText(String.format("Z: %s", linear_accelerationZ));
105
106             long currTime = System.currentTimeMillis();
107
108             if((currTime - lastSensorUpdateTime) > 50){
109                 lastSensorUpdateTime = currTime;
110
111                 sensorX = linear_accelerationX;
112                 sensorY = linear_accelerationY;
113                 sensorZ = linear_accelerationZ;
114             }
115         }
116     }
117
118     //Canvas
119     private class CanvasView extends View {
120         private Paint pen;
121         public CanvasView(Context context){
122             super(context);
123             setFocusable(true);
124
125             pen = new Paint();
```

```
126             }
128             public void onDraw(Canvas screen){
129                 pen.setStyle(Paint.Style.FILL_AND_STROKE);
130                 pen.setAntiAlias(true);
131                 pen.setTextSize(30f);
133                 int color = ContextCompat.getColor(MainActivity.this, R.color.
                        Orange);
134                 pen.setColor(color);
136                 screen.drawCircle(circleX, circleY, circleRadius, pen);
137             }
139         }
140 }
```

Listing 6.2: Code for ball movement based on accelerometer data

6.4.2. Bluetooth

As aforementioned earlier in this dissertation, an initial approach consisted of an android that interacted with a RaspberryPi and the STM board. However, due to the extraordinary conditions, one can only rely on the virtualization of the latter two for the perpetuation of the following project. Specifically, the STM board was simulated on a virtual machine with an 18.04 Lubuntu image, a lightweight Linux distribution based on Ubuntu that provides more freedom, simplicity and compatibility as opposed to other operating systems like Windows 10. This change implied some modifications in the smartphone Bluetooth module when interfacing with a virtual device rather than a physical one.

6.4.2.1. Bluetooth Connection Setup

Starting from the design and reusing previous Java code made on other course units (based on [?]), the Bluetooth setup was implemented on Android Studio and then deployed in a smartphone, the code is depicted in listing 6.3. Note that the forementioned code has threads to simulate parallelism computing since

the app should be able to send data to the paired device and receive data from it. In lines 10, 11 and 12, are represented the three thread classes used for the Bluetooth connection setup. These classes inherit from the Thread class (extends Thread - line 114), considering it must exist concurrency and resource sharing for the application to send and receive data simultaneously. The AcceptThread (line 10) it's related to the discovery of new connections since it plays an important role in the creation of a BluetoothServerSocket that allows two intended devices to find and accept each other as a part of the initial device inquiry prior to the connection stage. Line 11 represents the ConnectThread responsible for managing the connection between two devices. This thread starts as soon as the two devices accept each other and then proceeds to create a RfcommSocket and connect the devices (pair) when the user clicks on an unpaired device from the list presented. Finally, the IOThread assures the message exchange between devices by accessing the input and output streams of the Bluetooth Socket. As stated earlier, it was necessary to make some includes in the Bluetooth app as well as virtualize some ports in order to interface the Personal Computer (PC) virtual machine with the smartphone, this will be more thoroughly explained in the testing section (7.1.4.4). Finally, from the smartphone point of view, the protocol purpose was to transmit commands to control the vehicle by sending the phone accelerometers' values and receive its message warnings on screen.

```
1  public class ChatHandler {
2      private static final String TAG ="log";
3      private static final String APP_NAME = "RFCAR App";

5      // Unique UUID for this application
6      private static final UUID MY_UUID = UUID.fromString("8ce255c0-200a-11
           e0-ac64-0800200c9a66");

8      private final BluetoothAdapter bluetoothAdapter;
9      private final Handler handler;
10     private AcceptThread  acceptThread;
11     private ConnectThread  connectThread;
12     private IOThread ioThread;
13     private int state;
14     private Context mContext;
15     private UUID deviceUUID;
16     private BluetoothDevice bluetoothDevice;
17     byte[] buffer;
```

```
19      private static final int LISTENING_STATE = 1;
20      private static final int CONNECTING_STATE = 2;
21      private static final int NO_STATE = 4;
22      static final int CONNECTED_STATE = 3;

24      //Constructor
25      ChatHandler(Context context, Handler handler) {
26          mContext = context;
27          bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
28          state = NO_STATE;
29          this.handler = handler;
30          start();
31      }

33      // initiate connection to remote device
34      synchronized void initializeClient(BluetoothDevice device, UUID uuid)
            {
35          // Cancel any thread
36          if (state == CONNECTING_STATE) {
37              if (connectThread != null) {
38                  connectThread.cancel();
39                  connectThread = null;
40              }
41          }

43          // Cancel running thread
44          if (ioThread != null) {
45              ioThread.cancel();
46              ioThread = null;
47          }

49          // Start the thread to connect with the given device
50          connectThread = new ConnectThread (device, uuid);
```

```
51          connectThread.start();
52          setState(CONNECTING_STATE);
53      }

55      synchronized int getState() {
56          return state;
57      }

59      private synchronized void connected(BluetoothSocket socket,
            BluetoothDevice device) {

61          Log.d(TAG, "connected: Starting connected thread");

63          //Cancelling all kinds of threads that may be running to start a
                new one

65          // Stopping the thread if it is in connecting state
66          if (connectThread != null) {
67              connectThread.cancel();
68              Log.d(TAG, "connected: Cancelled connectThread");
69              connectThread = null;
70          }

72          // Cancel running thread that refers to the connected state
73          if (ioThread != null) {
74              ioThread.cancel();
75              Log.d(TAG, "connected: Cancelled ioThread");
76              ioThread = null;
77          }

79          //Stopping the scanning thread
80          if (acceptThread != null) {
81              acceptThread.cancel();
82              Log.d(TAG, "connected: Cancelled acceptThread");
```

```
83              acceptThread = null;
84          }
86          // Start the thread to manage the connection and perform
                transmissions
87          ioThread = new IOThread(socket);
88          ioThread.start();
90          Log.d(TAG,"IOthread: ioThread.start() called");
92          Message msg = handler.obtainMessage(Drawer_Activity.
                MESSAGE_DEVICE_OBJECT);
93          Bundle bundle = new Bundle();
94          bundle.putParcelable(Drawer_Activity.DEVICE_OBJECT, device);
95          msg.setData(bundle);
96          handler.sendMessage(msg);
98          setState(CONNECTED_STATE);
99      }
101     void write(byte[] out) {
102         IOThread r;
103         synchronized (this) {
104             if (state != CONNECTED_STATE) {
105                 Log.d(TAG, "IOthread: Write: state != CONNECTED_STATE");
106                 return;
107             }
108             r = ioThread;
109         }
110         r.write(out);
111     }
113     // runs while scanning for connections
114     private class AcceptThread  extends Thread {
```

```
115         private final BluetoothServerSocket serverSocket;

117         //Constructor
118         AcceptThread() {
119             BluetoothServerSocket bluetoothServerSocket = null;
120             try {
121                 bluetoothServerSocket = bluetoothAdapter.
                        listenUsingInsecureRfcommWithServiceRecord(APP_NAME,
                        MY_UUID);
122                 Log.d(TAG, "AcceptThread: Setting up the Server using: " +
                         MY_UUID);
123             } catch (IOException ex) {
124                 ex.printStackTrace();
125                 Log.e(TAG, "AcceptThread: IOException " + ex.getMessage())
                        ;
126             }
127             serverSocket = bluetoothServerSocket;
128         }

130         public void run() {



133             Log.d(TAG, "run: AcceptThread Running.");
134             BluetoothSocket bluetoothSocket = null;
135             while (state != CONNECTED_STATE) {
136                 try {
137                     Log.d(TAG, "Running: RFCOM server socket started...");
138                     bluetoothSocket = serverSocket.accept();
139                     int d = Log.d(TAG, "AcceptThread: accepted the request
                            ");
140                 } catch (IOException e) {
141                     Log.e(TAG, "AcceptThread: IOException" + e.getMessage
                            () );
142                     break;
```

```
143                 }
145                 // Only one thread can execute at a time.
146                 // sync_object is a reference to an object
147                 // whose lock associates with the monitor.
148                 // The code is said to be synchronized on
149                 // the monitor object
150                 //synchronized(sync_object)
151                 //{
152                 //    // Access shared variables and other
153                 //    // shared resources
154                 //}
156                 // If a connection was accepted
157                 if (bluetoothSocket != null) {
158                     synchronized (ChatHandler.this) {
159                         switch (state) {
160                             case LISTENING_STATE:
161                             case CONNECTING_STATE:
162                                 // start the connected thread.
163                                 Log.i(TAG, "Call to connected method");
164                                 connected(bluetoothSocket, bluetoothSocket
                                        .getRemoteDevice());
165                                 Log.i(TAG, "END
                                        initializeClientAcceptThread ");
166                                 break;
167                             case NO_STATE:
168                             case CONNECTED_STATE:
169                                 // Either not ready or already connected.
                                        Terminate
170                                 // new socket.
171                                 try {
172                                     bluetoothSocket.close();
173                                 } catch (IOException e) {}
```

```
174                                 break;
175                             }
176                         }
177                     }
178                 }
179             }

181             void cancel() {
182                 try {
183                     serverSocket.close();
184                     Log.i(TAG, "END mAcceptThread ");
185                 } catch (IOException e) {
186                     Log.e(TAG, "cancel: Close of AcceptThread ServerSocket
                        failed. " + e.getMessage() );
187                 }
188             }
189         }

191         // runs while attempting to pair with a device
192         private class ConnectThread  extends Thread {
193             private BluetoothSocket bluetoothSocket;

195             //Constructor
196             ConnectThread(BluetoothDevice device, UUID uuid) {
197                 Log.d(TAG, "ConnectThread: started.");
198                 bluetoothDevice = device;
199                 deviceUUID = uuid;
200             }

202             public void run() {
203                 setName("PairingThread");
204                 Log.i(TAG, "Running ConnectThread ");

206                 BluetoothSocket temp = null;
```

```
207             try {
208                 Log.d(TAG, "ConnectThread: Trying to create
                        InsecureRfcommSocket using UUID: "+ MY_UUID );
209                 temp = bluetoothDevice.createRfcommSocketToServiceRecord(
                        MY_UUID);

211             } catch (IOException e) {
212                 e.printStackTrace();
213                 Log.e(TAG, "ConnectThread: Could not create
                        InsecureRfcommSocket " + e.getMessage());
214             }

216             bluetoothSocket = temp;

218             //Cancel discovery
219             bluetoothAdapter.cancelDiscovery();

221             // Make a connection to the BluetoothSocket
222             try {
223                 bluetoothSocket.connect();
224             } catch (IOException e) {
225                 try {
226                     bluetoothSocket.close();
227                     Log.d(TAG, "run: Closed Socket");
228                 } catch (IOException exception) {
229                     Log.e(TAG, "ConnectThread: run: Unable to close
                            connection in socket " + exception.getMessage());
230                 }

232                 //connectionFailed
233                 Log.d(TAG, "run: ConnectThread: Could not connect to UUID:
                        " + MY_UUID );
234                 Message msg = handler.obtainMessage(Drawer_Activity.
                        MESSAGE_TOAST);
```

```
235                 Bundle bundle = new Bundle();
236                 bundle.putString("Toast", "Unable to connect to the device
                        ");
237                 msg.setData(bundle);
238                 handler.sendMessage(msg);
239             }

241             // Reset the ConnectThread
242             synchronized (ChatHandler.this) {
243                 connectThread = null;
244             }

246             // Start the connected thread
247             connected(bluetoothSocket, bluetoothDevice);
248         }

250         void cancel() {
251             try {
252                 Log.d(TAG, "cancel: Closing Client Socket");
253                 bluetoothSocket.close();
254             } catch (IOException e) {
255                 Log.e(TAG, "cancel: close() of mmSocket in Connectthread
                        failed. " + e.getMessage());
256             }
257         }
258     }

260     // runs during a connection with a remote device
261     private class IOThread extends Thread {
262         private final BluetoothSocket bluetoothSocket;
263         private final InputStream inputStream;
264         private final OutputStream outputStream;

266         IOThread(BluetoothSocket socket) {
```

```
267             Log.d(TAG, "IOThread: Starting.");

269             bluetoothSocket = socket;
270             InputStream auxIn = null;
271             OutputStream auxOut = null;

273             try {
274                 auxIn = socket.getInputStream();
275                 auxOut = socket.getOutputStream();
276             } catch (IOException ex) {
277                 ex.printStackTrace();
278             }
279             inputStream = auxIn;
280             outputStream = auxOut;
281         }

283         public void run() {
284             buffer = new byte[1024]; // buffer for the stream
285             int bytes;

287             //Listening to the InputStream
288             while (true) {
289                 try {
290                     // Reading from the InputStream

292                     bytes = inputStream.read(buffer);

294                     String incomingMessage = new String(buffer, 0, bytes);
295                     Log.d(TAG, "InputStream: " + incomingMessage);
296                     // Send the obtained bytes to the UI Activity
297                     handler.obtainMessage(Drawer_Activity.MESSAGE_READ,
                            bytes, -1,
298                             buffer).sendToTarget();
```

```
299                     Log.d(TAG, "InputStream: " + "Msg Received :" +
                            incomingMessage);



302                 } catch (IOException e) {
303                     Log.e(TAG, "write: Error reading Input Stream. " + e.
                            getMessage() );
304                     Message msg = handler.obtainMessage(Drawer_Activity.
                            MESSAGE_TOAST);
305                     Bundle bundle = new Bundle();
306                     bundle.putString("Toast", "Device connection was lost"
                            );
307                     msg.setData(bundle);
308                     handler.sendMessage(msg);

310                     // Start the service over to restart listening mode
311                     ChatHandler.this.start();
312                 }
313             }
314         }

316         // writing to OutputStream
317         void write(byte[] buffer) {
318             try {
319                 Log.d(TAG, "write: Writing to outputstream: " + Arrays.
                        toString(buffer));
320                 outputStream.write(buffer);
321                 Log.d(TAG, "InputStream: " + "Msg Sent :" + Arrays.
                        toString(buffer));
322                 handler.obtainMessage(Drawer_Activity.MESSAGE_WRITE, -1,
                        -1, buffer).sendToTarget();
323             } catch (IOException e) { Log.e(TAG, "write: Error writing to
                    output stream. " + e.getMessage() );}
324         }
```

```
326         void cancel() {
327             try {
328                 bluetoothSocket.close();
329             } catch (IOException e) {
330                 e.printStackTrace();
331                 Log.e(TAG, "cancel: Error closing the socket " + e.
                        getMessage() );
332             }
333         }
334     }

336     private synchronized void start() {
337         // Cancel any thread
338         if (connectThread != null) {
339             connectThread.cancel();
340             connectThread = null;
341         }

343         // Cancel any running thread
344         if (ioThread != null) {
345             ioThread.cancel();
346             ioThread = null;
347         }

349         setState(LISTENING_STATE);
350         if (acceptThread == null) {
351             acceptThread = new AcceptThread ();
352             acceptThread.start();
353         }
354     }

356     private synchronized void setState(int state) {
357         this.state = state;
```

```
358     }
359
360     public synchronized void stop() {
361         if (connectThread != null) {
362             connectThread.cancel();
363             connectThread = null;
364         }
365
366         if (ioThread != null) {
367             ioThread.cancel();
368             ioThread = null;
369         }
370
371         if (acceptThread != null) {
372             acceptThread.cancel();
373             acceptThread = null;
374         }
375         setState(NO_STATE);
376     }
377 }
```

Listing 6.3: Code for bluetooth connection setup

7. Testing

After implementing the solution developed in the various domains into the target platforms, the system's behaviour is tested in several levels of granularity: at the subsystem level — unit testing, and system level — integrated testing. The idea is progressively test the behaviour of each subsystem and its integration into the system. In this chapter are presented the unit testing and integrated testing for the RFCAR.

7.1. Unit testing

7.1.1. Navigation Virtual Subsystem

7.1.1.1. Control

7.1.2. Physical Environment Virtual Subsystem

7.1.3. Remote Vision Virtual Subsystem

7.1.4. Smartphone

7.1.4.1. Bluetooth Connection

After effectively changing the direction of a ball according to the accelerometer's state in the test app, one could only need to merge its code developed with the Bluetooth setup, so that, instead of sending plain text messages, one can transfer contents of the phone. However, to prove this new compound functionality, the receiving device should also have Bluetooth drivers (setup) and also specific hardware to use that protocol. An initial test was made with the HC-05 Bluetooth module inserted on the STM board. That approach only allows system engineers who have access to that module to fully test both Bluetooth client and server. Fortunately, a new solution was found where instead of having to own more hardware, one could run server Bluetooth app in a virtual machine using, for example, COM6 port to communicate, and then redirecting

to that port another one (COM9) for the establishment of the phone connection, since the former port was already used by the STM virtualization. As depicted in Fig. zz, there is a bidirectional exchange of information proving the android-STM communication functionality.

7.1.4.2. UI

In the beginning, an app UI was initially thought out and made a first test design. When the user opens the program, the image of the RFCAR appears along with a built-in set of features. It is then explained the vehicle purpose as a navigation tool through inhospitable environments. Following that, on the top left corner, there's a button that when pressed it redirects to the main app activity (more on that later) allowing first-person vision on the controlled car just like a simulated car game, only this time in a real-life scenario. Some statistics, such as the transport position and its velocity, appear in front of full-screen video capture. From that point on, one can also choose to see the map and possibly the route traced along with some more detailed statistics of the remote control car.

7.1.4.3. Applying Accelerometer Data

7.1.4.4. Message Exchange

7.2. Integrated testing

8. Verification and Validation

After testing the behaviour of the devised solution, its performance should now be tested and analysed in respect to the foreseen product specifications. Additionally, the product should be validated by an external agent to the development team to assess its overall suitability to its intended purpose. In this chapter, the verification and validation tests performed are presented and analysed.

8.1. Verification

In the verification phase, the product's performance is tested and verified its compliance to the foreseen specifications.

8.2. Validation

In this stage, the product is tested by an external agent to the development team to assess its overall suitability to its intended purpose.

9. Conclusion

In this chapter the conclusions and prospect for future work are presented.

Appendices

A. Project Planning — Gantt diagram

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

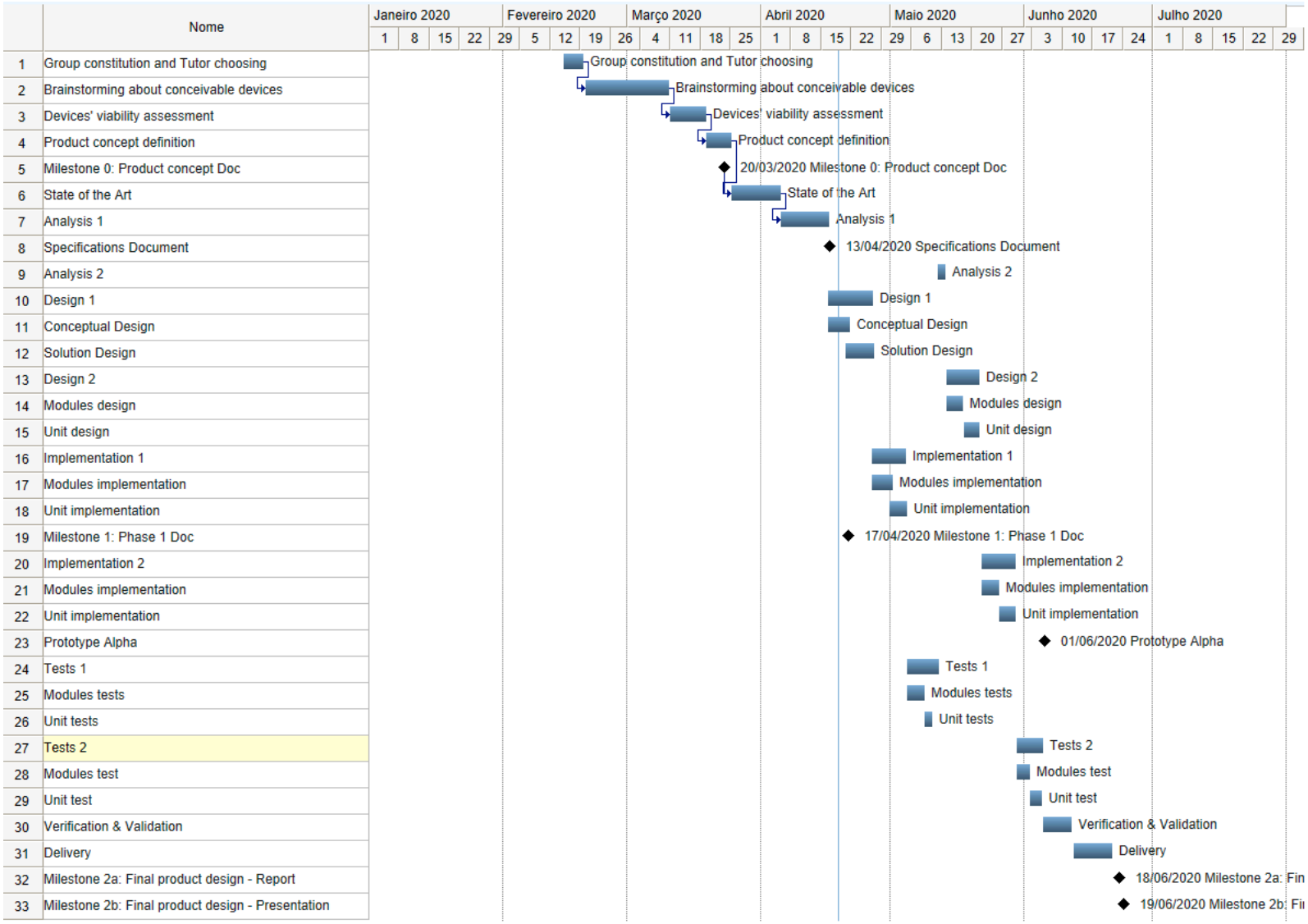

Figure A.1.: Project planning — Gantt diagram