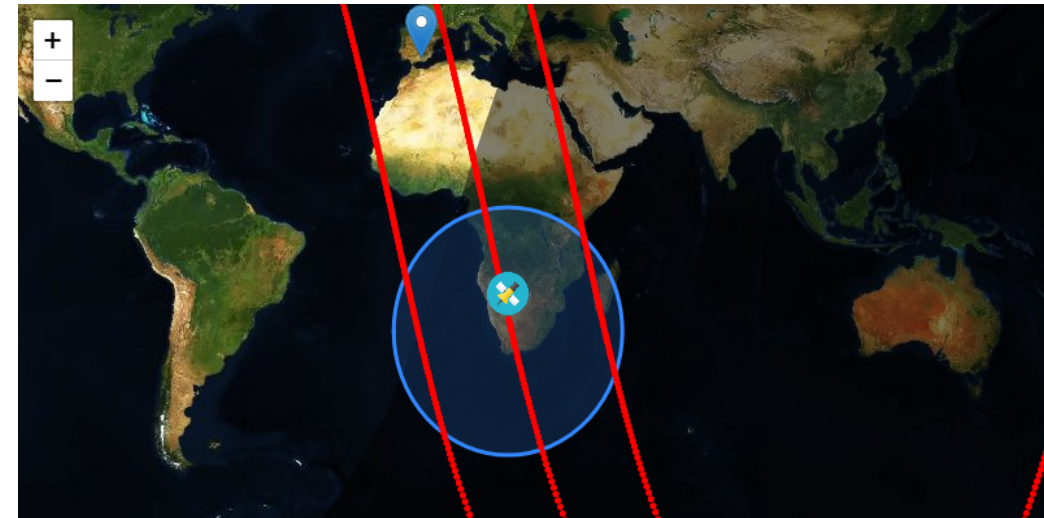




UNIVERSITY OF GRANADA

Bachelor of Computer Engineering



Bachelor Thesis

Ground station control for telemetry and telecontrol of Cubesat

Antonio Serrano de la Cruz Parra

2017/2018

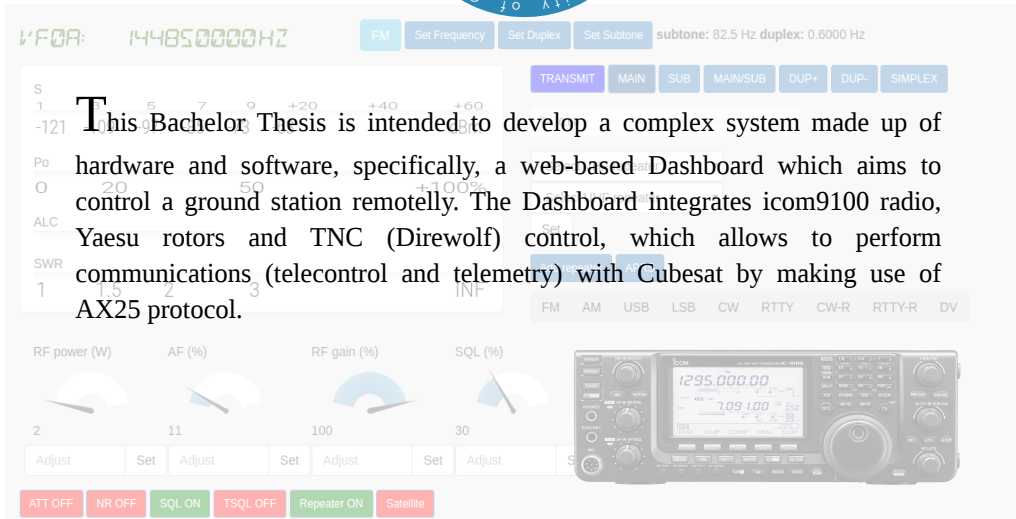
Tutor: Andrés María Roldán Aranda

Ground station control for telemetry
and telcontrol of Cubesat

Antonio Serrano de la Cruz Parra

COMPUTER
ENGINEERING

17/18



This Bachelor Thesis is intended to develop a complex system made up of hardware and software, specifically, a web-based Dashboard which aims to control a ground station remotely. The Dashboard integrates icom9100 radio, Yaesu rotors and TNC (Direwolf) control, which allows to perform communications (telecontrol and telemetry) with Cubesat by making use of AX25 protocol.



Antonio Serrano de la Cruz Parra was born in La Solana (Ciudad Real) in 1996. With this Bachelor Thesis complements his education in others specialities and finalizes his Bachelor's Degree in Computer Engineering at the University of Granada (Spain).



Andrés María Roldán Aranda is the academic head of the present project, and the student's tutor. He is professor in the Department of Electronics and Computers Technologies



Pablo Garrido Sánchez is engineer and Phd student in the Department of Electronics and Computers Technologies at the University of Granada. Currently he is collaborating with GranaSAT Project regarding Groundstation aspects.

Copy for the student / Copia para el alumno



**BACHELOR OF
COMPUTER ENGINEERING**

Bachelor Thesis

*“Ground station control for telemetry and
telecommand of Cubesat”*

ACADEMIC COURSE: 2017/2018

Antonio Serrano de la Cruz Parra



BACHELOR OF COMPUTER ENGINEERING

“Ground station control for telemetry and telecommand of Cubesat”

AUTHOR:

Antonio Serrano de la Cruz Parra

SUPERVISED BY:

Andrés María Roldán Aranda

Pablo Garrido Sánchez

DEPARTMENT:

Electronics and Computer Technologies

D. Andrés María Roldán Aranda, Profesor del departamento de Electrónica y Tecnología de los Computadores de la Universidad de Granada, como director del Trabajo Fin de Grado de D. Antonio Serrano de la Cruz Parra, y D. Pablo Garrido Sánchez, alumno de Doctorado del departamento de Electrónica y Tecnología de los Computadores de la Universidad de Granada, como cotutor del mismo

Informan:

Que el presente trabajo, titulado:

“Control de estación terrena para telecontrol y telemetría de Cubesat”

ha sido realizado y redactado por el mencionado alumno bajo nuestra dirección, y con esta fecha autorizan a su presentación.

Granada, a 10 de Junio de 2018

P. Garrido

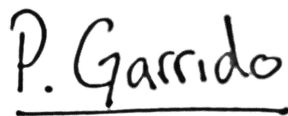


Fdo. Pablo Garrido Sánchez


Fdo. Andrés María Roldán Aranda

Los abajo firmantes autorizan a que la presente copia de Trabajo Fin de Grado se ubique en la Biblioteca del Centro y/o departamento para ser libremente consultada por las personas que lo deseen.

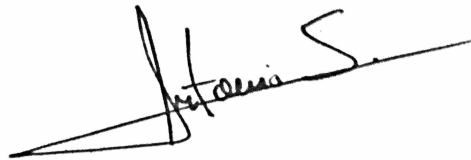
Granada, a 10 de junio de 2018

Handwritten signature of P. Garrido, underlined.

Fdo. Pablo Garrido Sánchez

Handwritten signature of Andrés María Roldán Aranda.

Fdo. Andrés María Roldán Aranda

Handwritten signature of Antonio Serrano de la Cruz Parra.

Fdo. Antonio Serrano de la Cruz Parra

Control de estación terrena para telecontrol y telemetría de Cubesat

Antonio Serrano de la Cruz Parra

PALABRAS CLAVE:

[Dashboard](#), [Cubesat](#), [Ground Station](#), [AX25](#), [TNC](#), [Direwolf](#), [satellite tracking](#), [transceiver](#), [NodeJS](#), [Javascript](#), [Bootstrap](#), [HTML](#), [CSS](#)

RESUMEN:

[GranaSAT](#) es un grupo espacial de la Universidad de Granada en el que se desarrollan diferentes actividades relacionadas con los ámbitos de la electrónica y la ingeniería espacial. Durante el desarrollo de actividades del grupo [GranaSAT](#) se comenzó el desarrollo de un [Dashboard](#) web, con el propósito de controlar una estación terrena ([Ground Station](#)) y todos sus componentes de forma remota, sin encontrarse físicamente en contacto con la estación.

El objetivo del siguiente proyecto es integrar en la web un sistema que permita el telecontrol y telemetría de un [Cubesat](#) haciendo uso de una [Ground Station](#). De este modo, el objetivo será implementar las funciones que permitan controlar remotamente la [Ground Station](#), de modo que se pueda realizar el envío de comandos (telecontrol) y obtención de información (telemetría) hacia y desde un [Cubesat](#), todo ello mediante el uso del protocolo [AX25](#). Por otro lado, se realizarán además algunas mejoras en el [Dashboard](#) (nuevas funcionalidades, gestión de usuarios, mejoras de la interfaz, etc), todo de acuerdo a los requisitos de [GranaSAT](#).

Por tanto, el presente Trabajo Fin de Grado pretende emular un encargo profesional real siguiendo una metodología orientada a producto. Toda decisión en referencia al desarrollo será tomada en función de los requisitos del cliente (en este caso [GranaSAT](#)), teniendo en cuenta en todo momento los costes.

A partir de los requisitos del cliente se realizará un proceso de análisis y comparativa de las diferentes soluciones propuestas, de manera similar a como se haría en un proyecto profesional final.

A pesar de que el alumno posee la especialidad en Computación y Sistemas Inteligentes, se ha tratado de dar al proyecto un enfoque multidisciplinar, tratando tecnologías y aptitudes que el alumno ha desarrollado durante la titulación, así como nuevas que se han adquirido durante el desarrollo del proyecto.

Ground station control for telemetry and telecommand of Cubesat

Antonio Serrano de la Cruz Parra

KEYWORDS:

Dashboard, Cubesat, Ground Station, AX25, TNC, Direwolf, satellite tracking, transceiver, NodeJS, Javascript, Bootstrap, HTML, CSS

ABSTRACT:

[GranaSAT](#) is a group consisted of students of the University of Granada, specifically, students who are willing to acquire new knowledge related to electronics and aerospace fields. Throughout all the activities carried out within the [GranaSAT](#) project, a web-based [Dashboard](#) was started, which aims to control a [Ground Station](#) and all its components remotely.

The main purpose of this Project is integrating a web system that allows the remote control of the [Ground Station](#), thereby, by making use of the [Ground Station](#) it will be possible to send commands (telecontrol) and receive telemetry from a [Cubesat](#), all of this by using the [AX25](#) protocol. Furthermore, some other functionalities and improvements will be developed in the [Dashboard](#), everything according to [GranaSAT](#) requirements.

The present Final Project tries to simulate a professional assignment following a product-oriented philosophy. Every decision regarding the development will be made taking into account client requirements ([GranaSAT](#) in this case), taking into account economic costs as well. Thus, from the client requirements, an analysis will be performed in order to determine what are the suitable technologies and solutions, in a similar way as it would be done in a professional work.

This Final Project is presented as “Trabajo Fin de Grado” within the Degree “Grado en Ingeniería Informática” at “Universidad de Granada”. Although the student has acquired his specialization in Computing and Intelligent Systems, the project is intended to have a multidisciplinary approach, involving not only technologies and aptitudes that have been already acquired by the student within the Degree, but also new ones that have been acquired during the Final Project development.

*Never stop walking,
no matter how hard it gets*

Agradecimientos:

El desarrollo del presente proyecto nunca habría sido posible sin la colaboración de numerosas personas. Muchas de ellas han contribuido de manera secundaria a través de sus aportaciones en Internet y diversos manuales, a las cuáles les estoy muy agradecido. Por otro lado, cabe agradecer expresamente en las siguientes líneas a todas las personas con las que mantuve contacto directo y que hicieron posible el desarrollo de este proyecto.

A mi cotutor Pablo Garrido Sánchez, por sus mil y una explicaciones y sugerencias durante el desarrollo de todo el proyecto, sin duda unas de las personas más competentes que he conocido.

A Luis Sánchez, por su ayuda y por prestarme su Arduino.

A mis compañeros y amigos de facultad, sin los cuáles nunca habría llegado hasta aquí.

A mi familia, por su apoyo incondicional en todas las decisiones que he tomado a lo largo de mi vida, siendo una de mis grandes motivaciones es el hecho de hacerles sentir orgullosos.

Y por último y no menos importante, a mi tutor de proyecto, Andrés María Roldán, gracias al cual he podido adquirir conocimientos en el ámbito aerospacial, algo que no había podido hacer antes durante mis cuatro años de carrera. Su afán de mejora y perfeccionismo han sacado lo mejor de mí, sin duda su aparición en este trayecto ha sido determinante.

Muchísimas gracias a todos.

Acknowledgments:

The development of the presented project could have never been possible without the collaboration of many people. Most of them were people from the Internet which I did not have the pleasure of meeting but to which I am really thankful. On the other hand, I wanted to thank sincerely all of the people I could contact and interact with during the development of the project.

My co-supervisor Pablo Garrido Sánchez, who helped me whenever I needed it. He is one of the most competent people I have ever known.

Luis Sánchez, for his help and for lending me his Arduino.

My friends, thanks to them I have improved as a person as well as a professional.

My family, always supporting me in every decision I have made in my life.

Finally, not in relevance order, my tutor, Andrés María Roldán, the one who introduced me this project and thanks to whom I have learned tons of things related to aerospace and electronics. He has been one of the most determinant professor I have ever had.

All in all, thank you all who made it possible.

INDEX

Autorización Lectura	v
Autorización Depósito Biblioteca	vii
Resumen	ix
Dedicatoria	xiii
Agradecimientos	xv
Index	xix
List of Figures	xxv
List of Videos	xxix
List of Tables	xxxi
Glossary	xxxiii

Acronyms	xxxv
1 Introduction	1
1.1 Context	1
1.2 Motivation	2
1.3 Project Goals and Objectives	3
1.4 Project Structure	4
2 System Requirements Definition	1
2.1 Functional Requirements Definition	1
2.1.1 Primary	1
2.1.2 Secondary	2
2.2 Economics Requirements Definition	2
3 System Analysis	1
3.1 Ground Station Definition	1
3.1.1 GranaSAT Ground Station	3
3.2 Review of Solutions for Ground Station Control	7
3.2.0.1 Type of application	7
3.2.0.2 Front-end	9
3.2.0.3 Server side	11
3.3 Transceiver Telecontrol	16
3.3.1 Back-end	16
3.3.2 Front-end	20
3.4 Transceiver's Audio	22
3.5 TNC	24
3.6 Yaesu Rotors Telecontrol	26
3.6.1 Back-end	26
3.6.2 Front-end	26

3.7	Satellite Tracking	27
3.7.1	Extraction of satellite data	27
3.7.2	Display over interactive map	29
3.7.3	Integration within the system	29
3.8	Other Improvements	31
3.8.1	Tooltips definition	31
3.8.2	Camera streaming on the server	32
3.8.3	User account management	33
4	System Design	1
4.1	Transceiver Telecontrol	3
4.1.1	Back-end	3
4.1.2	icom9100.js class	4
4.1.3	Front-end	6
4.1.4	Functions for graphical elements	8
4.2	Transceiver's Audio	11
4.2.1	Back-end	11
4.2.2	Front-end	12
4.3	TNC	14
4.3.1	Back-end	15
4.3.2	Front-end	17
4.4	Yaesu Rotors Telecontrol	20
4.4.1	Front-end	20
4.5	Satellite tracking	21
4.5.1	Back-end	21
4.5.2	Front-end	23
4.6	Other improvements	30
4.6.1	Tooltips design	30

4.6.1.1	Model	31
4.6.1.2	Controller	31
4.6.1.3	View	31
4.6.2	Webcam streaming design	32
4.6.2.1	Back-end	32
4.6.2.2	Front-end	34
4.6.3	User account management	35
4.6.3.1	Recovery password system	37
5	Test and Evaluation	1
5.1	Transceiver's control verification	1
5.2	Transceiver's audio verification	3
5.3	TNC (Direwolf) integration test	4
5.3.1	Receiving and decoding AX25 packets	4
5.3.2	Sending AX25 packets	6
5.4	Tracking satellite verification	8
5.5	Other improvements	9
5.6	Browser compatibility testing and performance	10
6	Conclusions and Future Lines	1
A	AX25 protocol	1
B	DarkIce and Icecast2: Installation	3
C	Direwolf: Installation	5
D	PM2: Installation and Use	7
E	SSL certificate installation	9
F	MotionEye: Installation and Configuration	11

G Project Budget	13
G.1 Hardware Cost	13
G.2 Software Cost	14
G.3 Human Resources Cost	14
References	15

LIST OF FIGURES

1.1	AAUCubeSat, University of Aalborg (Denmark) [29]	2
1.2	GranaSAT Logo	2
1.3	Waterfall model	4
1.4	Gantt diagram for planning project	6
3.1	Ground Station basic scheme	2
3.2	antenna	3
3.3	Yaesu rotors	4
3.4	Icom9100 transceiver	5
3.5	TNC (Terminal Node Controller) device [33]	5
3.6	GranaSAT Ground Station components scheme	6
3.7	Communication between Ground Station and Cubesat [35]	7
3.8	Most popular technologies during 2018 according to StackOverflow [26]	9
3.9	Most popular frameworks, libraries and tools in 2018 according to StackOverflow [26]	11
3.10	Most utilized data bases during 2018 according to StackOverflow [26]	13

3.11 GranaSAT Dashboard system architecture and technologies	14
3.12 Icom9100 data format [14]	17
3.13 ICOM9100 command table [14]	17
3.14 GPredict satellite tracking software	27
3.15 TLE example [34]	28
4.1 System architecture	2
4.2 icom9100 transceiver final front-end	6
4.3 icom9100 transceiver telecontrol process - example	8
4.4 Transceiver's graphical elements updating process	10
4.5 Audio devices available with arecord command	11
4.6 Transceivers audio streaming design	13
4.7 GranaSAT Ground Station final scheme	14
4.8 Direwolf terminal output	15
4.9 Decoding AX25 with Direwolf flowchart	17
4.10 Sending AX25 packets with Direwolf flowchart	18
4.11 Decoding packets (concretely AX25/APRS) and displaying in Dashboard	19
4.12 Sending AX25 packets on the Dashboard	19
4.13 Yaesu Rotors front-end	20
4.14 Satellite tracking flowchart	26
4.15 Satellite tracking in laptop screen	27
4.16 Satellite tracking in mobile phone device	27
4.17 Polar graph with satellites over Ground Station	29
4.18 Tooltips design diagram	30
4.19 MotionEye FrontEnd	32
4.20 MotionEye configuration	33
4.21 Streaming video final design	34
4.22 User account profile front-end	35

4.23	User account management system flowchart	36
4.24	Recovery password e-mail	37
4.25	Recovery password system flowchart	38
5.1	Transceiver control response time test	2
5.2	Transceiver's audio time delay test	3
5.3	Arduino AX25 transmitter [30]	4
5.4	AX25 decoded frame transmitted from Arduino [30]	5
5.5	AX25 telemetry received from Arduino [30]	6
5.6	NOAA 15 tracking in GranaSAT Ground Station	8
5.7	NOAA 15 tracking in www.n2yo.com	8
5.8	Dashboard CPU performance test	11
5.9	Dashboard memory performance test	11
5.10	Dashboard disk performance test	12
A.1	AX25 frame structure [15]	1
D.1	PM2 Execution	8

LIST OF VIDEOS

5.1	Direwolf decoding packets test (double click)	5
5.2	Direwolf sending packets test (double click)	6

LIST OF TABLES

3.1	Icom9100 transceiver specification	4
3.2	GranaSAT server characteristics	5
3.3	Application alternatives for Ground Station control	8
3.4	Front-end design frameworks comparison	10
3.5	Javascript web frameworks comparison	10
3.6	Javascript back-end technologies comparison	12
3.7	Databases comparison	13
3.8	Javascript gauges analyzed possibilities	20
3.9	Audio streaming from server to client analyzed possibilities	22
3.10	Virtual TNCs analyzed possibilities	24
3.11	Javascript libraries for interactive maps analyzed possibilities	29
3.12	Tooltips analyzed possibilities	31
3.13	Video streaming software analyzed possibilities	32
G.1	Hardware costs	13
G.2	Software costs	14

G.3 Human resources costs 14

GLOSSARY

AngularJS Javascript client-side framework used for creating dynamic web applications. It follows MVC pattern (<https://angularjs.org/>).

Arduino Open-source electronic prototyping platform enabling users to create interactive electronic objects featuring single-board micro-controllers and microcontroller kits for building digital devices and interactive objects that can sense and control objects in the physical world.

AX25 AX.25 A data link layer protocol derived from the X.25 protocol suite and designed for use by amateur radio operators..

azimuth Angle between a celestial body or satellite and the North, increasing this angle clockwise around the observer's horizon. If the body is in the North, azimuth is 0° , on the other hand, if the body is in the East, azimuth is 90° .

Bootstrap Free and open source front-end library for responsive web design, including HTML and CSS templates that provides different components such as buttons, navigation bars, forms, etc.

Cubesat Miniaturized satellite normally for space research, with dimensions of 1 dm^3 and mass lower than 1.33 kg per unit.

Dashboard Graphical user interface that displays information to the user in a easy way so that the user can interpret it and interact with the system.

Direwolf Direwolf is a "soundcard" AX.25 packet modem/TNC and APRS encoder/decoder. It is connected to a terminal computer and it acts as virtual TNC.

downlink Term referred to the link from the satellite down to the [Ground Station](#).

elevation Up-down angle between the celestial body or satellite and the observer's horizon. When the body is exactly above the observer, elevation is 90° . On the other hand, when the body is about to not to be visible by the observer, elevation is 0° .

GranaSAT GranaSAT is an academic project from the University of Granada consisting in designing and developing a picosatellite ([Cubesat](#)). Coordinated by the Professor Andrés María Roldán Aranda, GranaSAT is a multidisciplinary project with students from different degrees, where they can acquire and enlarge the knowledge necessary to face an actual aerospace project.

Ground Station Facilities in which instruments and devices necessary to establish a radio-link communication are normally located. Also used to control and monitor antenna system.

Inkscape Free and open-source professional vector-graphic editor (<https://inkscape.org/es/>).

Javascript Interpreted programming language commonly used in the client side of web applications, however, it is extensively used in the server side as well (<https://www.javascript.com/>).

Leaflet JavaScript library for mobile-friendly interactive maps (<http://leafletjs.com/>).

Mapbox Open source mapping platform for designed maps (<https://www.mapbox.com/>).

Model-View-Controller Architectural pattern used for developing graphical interfaces in which the application logic is divided into three components: model, view, and controller.

Motion Free and open-source software motion detector for Linux. It can monitor video signal from one or more cameras (<https://motion-project.github.io/>).

nginx Reverse high performance proxy and web server <https://nginx.org/en/>.

NodeJS Open source JavaScript run-time environment for application's server-side (<https://nodejs.org/es/>).

Python Interpreted and multi-platform programming language commonly used for scientific purposes(<https://www.python.org/>).

SGP4 Simplified perturbation model used to calculate orbital state of satellites..

transceiver Digital device that consists of both transceiver and receiver.

uplink Term referred to the link from a [Ground Station](#) up to the satellite.

ACRONYMS

AFSK Frequency-Shift Keying.

ALSA Advanced Linux Sound Architecture.

API Application Programming Interface.

APRS Automatic Packet Reporting System.

CSS Cascade Style Sheet.

GUI Graphical User Interface.

HTML HyperText Markup Language.

HTTP HyperText Transfer Protocol.

HTTPS HyperText Transfer Protocol Secure.

ISS International Space Station.

JSON JavaScript Object Notation.

KISS Keep It Simple, Stupid.

NASA National Aeronautics and Space Administration.

NORAD North American Aerospace Defense Command.

PM2 Production Process Manager.

SSL Secure Socket Layer.

SVG Scalable Vector Graphics.

TLE Two-Line Element.

TNC Terminal Node Controller.

CHAPTER

1

INTRODUCTION

1.1 Context

The presented Project fits within [GranaSAT](#), a multidisciplinary group made of students from different fields, students who are willing to acquire knowledge related to Electronics and Aerospace Engineering.

The main objective of [GranaSAT](#) is to develop a [Cubesat](#), which is a miniaturized satellite, measuring 10x10x10cm and weighing no more than 1.33Kg (an example of [Cubesat](#) can be seen in figure [1.1](#)). The purpose of a [Cubesat](#) is known as “payload” and this could be: taking pictures of earth, collecting telemetry such as temperature, pressure, magnetism, radiation measuring, etc.

Generally, they are usually intended for educational and space research purposes. Currently there are about 800 cubesats in orbit, owned by different universities, organizations or even private companies.

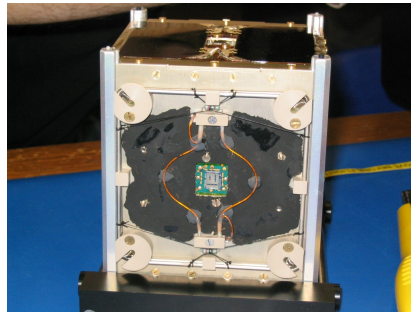


Figure 1.1 – AAUCubeSat, University of Aalborg (Denmark) [29]

The development of a [Cubesat](#) is a hard process that requires different modules and systems working together, therefore, many different projects have been already done by some students of [GranaSAT](#). However, there is still a lot of work to do in order to achieve the mentioned main objective. The presented Final Project aims to contribute to this general purpose.

Andrés María Roldán Aranda is the academic head of [GranaSAT](#), as well as the coordinator of the presented Project (see [GranaSAT](#) logo in 1.2).



Figure 1.2 – GranaSAT Logo

The laboratory and equipment needed for the presented Project are located in the iMUDS Building, next to “Parque Tecnológico de la Salud”, in Granada (Spain). The Project has been mainly developed in this laboratory, besides some work that has been done at home thanks to the possibility of working remotely with the server provided by [GranaSAT](#).

1.2 Motivation

When thinking about my Final Project, I thought I would like to go further and do something unusual, something that had nothing to do with the normal knowledge I had got from the University courses. I definitely wanted a new challenge where I had to learn new technologies, concepts and techniques that I had not seen before during the Degree.

Therefore, I found in this Final Project a great opportunity to achieve my goal, since this included things totally different from my specialization courses: hardware and software, web programming, aerospace and electronic topics, etc.

During the development of this Project I got myself in many hard situations where I have had to adapt myself and acquire a big amount of new concepts and techniques. I did not know before web programming or any framework related to this, and much less all the hardware, space communications and protocols that have been treated during this Project.

I realized that a good engineer is not the one who knows the most, but the one who knows how to adapt themselves to any kind of situation and any kind of real problem. I consider that during my Bachelor's Degree in Computer Engineering, one of the most useful things that I have learned is the ability to deal with problems and solve them somehow. This Final Project is a proof of that.

The final web-based [Dashboard](#) that comes out of this Final Project is the result of a careful process of study, analysis and comparison between different technologies regarding when to use each one to solve certain problems. I really hope this final product can be useful for the future development of [GranaSAT](#) projects.

1.3 Project Goals and Objectives

The main objectives of this Final Project are the following:

- Studying and analyzing how to approach the [Ground Station](#) control
- Studying and understanding about how communications are performed with [Cubesat](#)
- Deducing the main and secondary requirements of the system
- Analyzing current technologies which may be used to solve the presented problem, reasonably choosing the best one for each case
- Acquiring new hardware and software knowledge, getting the student closer to a real problem related to the aerospace and electronics field
- Demonstrating the knowledge acquired during the Degree, not only at the specialization coursed by the student, but also from a general point of view within Computer Engineering
- Successfully overcome the subject of the Bachelor Thesis

1.4 Project Structure

From now on, the waterfall model will be followed in order to develop the presented Project. The waterfall model sets different stages on the development process and once a phase of development is completed, the development proceeds with the next phase, without coming back.

This can be seen in 1.3.

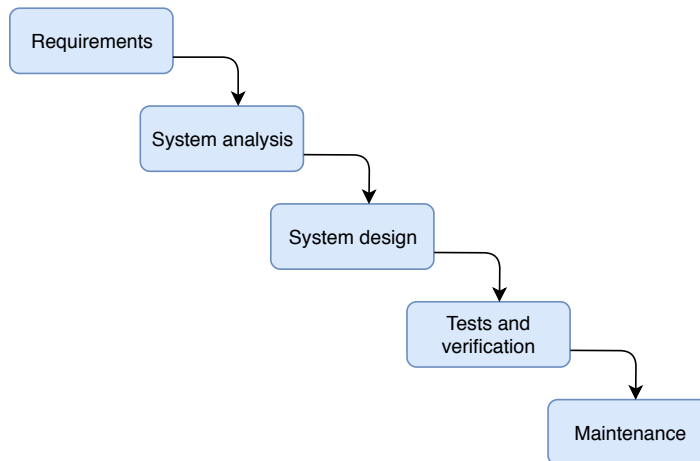


Figure 1.3 – *Waterfall model*

Based on this waterfall model, the Project is divided in six chapters and an appendix which tries to describe each part of the Project development in a chronological and logical way.

These chapters are:

- **Chapter one:** This chapter is intended to be an **introduction**, exposing objectives, motivation and planing of the presented Project.
- **Chapter two:** **Client requirements** are exposed in this chapter, both primary and secondary. These requirements mark the beginning of the later system analysis.
- **Chapter three:** The third chapter deals with **analysis** and development of different ideas in order to meet the requirements given from the client. In this stage, a high level of abstraction will be enough. Furthermore, different concepts will be analyzed and explained so that all the process is completely understood. At the end of this chapter, different alternatives will have been taken into account and the most suitable solution will be chosen and eventually implemented during the design stage.
- **Chapter four:** This chapter deals with **system design**. It includes the implementation of the different chosen solutions and aspects previously treated during the analysis. In this stage the level of abstraction will be low and every aspect regarding programming and source code will be treated.
- **Chapter five:** In this chapter, a number of **tests and evaluations** is performed in order to check whether the system works as expected and consequently the given

requirements are met.

- **Chapter six** Finally, chapter six concludes with a brief **reflection** about the results obtained and **future lines** for the Project, besides a personal consideration.
- In addition, an **appendix** has been included, which deals with secondary aspects related to the Project, besides the Project costs.

A Final Project planning can be seen in figure 1.4.

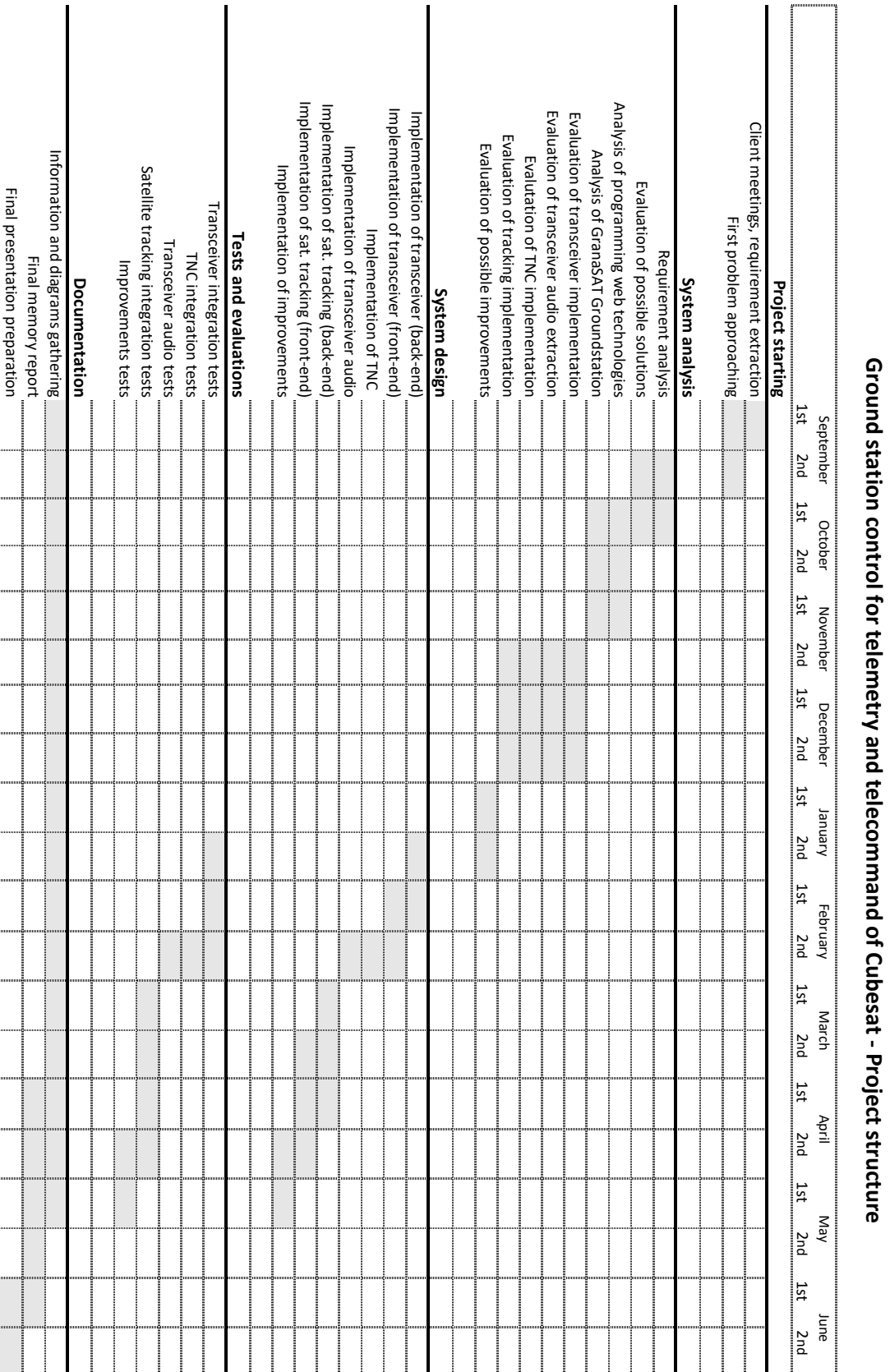


Figure 1.4 – Gantt diagram for planning project

CHAPTER

2

SYSTEM REQUIREMENTS DEFINITION

Since the presented Project follows a product-oriented philosophy, a set of system requirements are defined from the client at the first stage of the development.

2.1 Functional Requirements Definition

As Requirements Engineering process defines, functional requirements reflect what the system is expected to do. These will be given from the client (in this case [GranaSAT](#)), therefore, they do not need to be technical but a brief description in natural language. During the development, all of these requirements will be met and the system is expected to comply with all of them at the end of the design stage.

According to its importance in the system, the requirements can be divided in **primary** and **secondary**.

2.1.1 Primary

- a) The system will be able to control the [transceiver](#) Icom9100 remotely, in order to control and adjust its parameters
- b) The system will be able to send commands (also known as telecommand) by making use of a [TNC](#) that allows to use [AX25](#) packets

- c) The system will be able to decode [AX25](#) packets by using a [TNC](#) and shows the proper data that these contain (also known as telemetry)
- d) The system will receive the [transceiver](#)'s audio in live, so the users can listen remotely what is actually sounding in the [transceiver](#)
- e) The system will have a satellite tracking in live so the users can see the satellite's position at every moment, in order to perform communications when this is visible from the [Ground Station](#)

2

2.1.2 Secondary

- f) The system will allow users to modify and manage their profiles (username, password, image...) as well as recovering their passwords
- g) The front end design will be modified with more functionalities (tooltips, gauges, etc)
- h) The system will include a camera system in order to watch the antenna and the [Ground Station](#) remotely

2.2 Economics Requirements Definition

Regarding economics requirements, the Project follows a cost-oriented philosophy, therefore, during the analysis stage all the available options and technologies will be taken into account, preferably choosing open-source alternatives.

CHAPTER

3

SYSTEM ANALYSIS

In this chapter, an analysis is done in order to meet the requirements described in chapter 2. At this point, the analysis will start at a high level of abstraction, reducing it progressively until implementation in the system is defined in chapter 4.

To meet the requirements defined from the client, different technologies and solutions are suggested, choosing the most suitable for the system.

3.1 Ground Station Definition

First of all, let us begin with explaining what a [Ground Station](#) is and what is intended for.

Within the telecommunications field, a [Ground Station](#) is a terrestrial radio station designed for telecommunications with spacecraft (satellites flying in outer space), transmitting and receiving radio waves. The [Ground Station](#) provides a radio interface between space and ground for telemetry and telecommand purposes (see image 3.1). [31]

- telecommand processing: commands are sent to satellites from the [Ground Station](#), via [uplink](#). These commands are used to control the satellites remotely and they could be used for example, to turn on/off certain parts of the satellite, or to activate certain modules.

These commands are merely information that is encoded and modulated onto an assigned radio frequency band. This radio frequency is eventually amplified an

carried to an antenna for final transmission.

- telemetry processing: this is the opposite process. The satellite transmits signals via **downlink**, signals that are received in the **Ground Station**, demodulated and eventually decoded to extract the proper information. Telemetry is used to determine the status of a satellite (temperature, batteries status, etc) or to get information collected by the satellite (images, measurements, predictions, etc).

Telecommand and telemetry are usually encrypted in order to prevent unauthorized access to the satellite and its data.

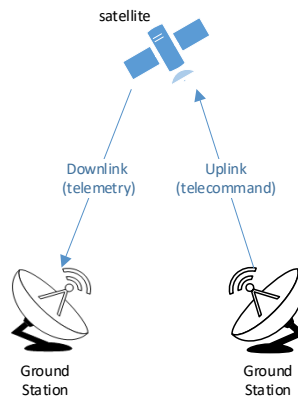


Figure 3.1 – *Ground Station basic scheme*

3.1.1 GranaSAT Ground Station

The final purpose of [GranaSAT Ground Station](#) is to perform the communications described above (telecommand and telemetry) with a possible [Cubesat](#). Therefore, the [Ground Station](#) must be able to send/receive digital data.

In [Cubesat](#) projects, [AX25](#) is the adopted protocol for this digital communication (see appendix [A](#) for more information about [AX25](#)).

Having said this, let us describe in detail [GranaSAT Ground Station](#):

Normally, a common [Ground Station](#) consists of an antenna, receiver, transmitter, a [TNC](#), a computer and operation software (e.g. satellite tracking) [[35](#)]. Specifically, the [GranaSAT Ground Station](#) components will be described below:

- **Antenna:** its aim is to perform communication with satellites on the outer space, as well as with other stations on earth. The antenna is capable of transmitting and receiving radio waves in certain band frequencies (this bands depend on the antenna's characteristics).

These transmitted and received radio frequencies can contain different data (telemetry, images, just voice, etc.) that needs to be demodulated and converted into human-understood format, that is, digital format (audio, text, etc.).

At this level of abstraction, it is not necessary to know deeper characteristics about the antenna. An image of the antenna can be seen in figure [3.2](#)



Figure 3.2 – *antenna*

- **Yaesu G5500 rotors:** these rotors allow the antenna to aim towards a desired direction. This direction is defined by two elements: [azimuth](#) and [elevation](#). The rotors have a controller device that allows to control the rotors remotely provided the desired values of [azimuth](#) and [elevation](#) angles. When performing satellite communications, the antenna needs to be pointed to the current satellite's position (which can be known with a satellite tracking software) in order to receive/send radio waves from/to the satellite. Therefore, the antenna's

position needs to be modified in order to follow the current satellite's position. An image of Yaesu rotors can be seen in figure 3.3.



Figure 3.3 – Yaesu rotors

- Icom9100 transceiver:** this is the most important hardware device of the system, which consists of a transmitter and receiver, both in a single device (see characteristics in 3.1). Connected to an antenna, its aim is to demodulate the radio frequency signals that come from the antenna into audio and modulate the input signals before transmitting them through radio frequencies. The [transceiver](#) together with a [TNC](#) can be used for sending/receiving digital data that is encapsulated over the mentioned above [AX25](#) protocol.

Frequency bands Tx:	0.03-60.000MHz, 136-174MHz 420-480MHz, 1240-1320MHz
Frequency bands Rx	1.800-1.999 MHz, 3.500-3.999 MHz 5.3305, 5.3465, 5.3665, 5.3715 5.4035 MHz, 7.000-7.300 MHz 10.100-10.150 MHz, 14.000-14.350 MHz 18.068-18.168 MHz, 21.000-21.450 MHz 24.890-24.990 MHz, 28.000-29.700 MHz 50.000-54.000 MHz, 144.000-148.000 MHz 430.000-450.000 MHz, 1240.000-1300.000 MHz
Modes	USB, LSB, CW, RTTY, AM, FM, (DV optional)
Antenna connector	SO-239 x 3 and N (50 ohms)
Usable temperature range	0°C to +50°C; +32°F to +122°F
Frequency stability	less than ± 0.5 ppm
Power supply requirement	13.8 V DC ($\pm 15\%$ negative ground)

Table 3.1 – Icom9100 *transceiver* specification



Figure 3.4 – *Icom9100 transceiver*

- **TNC:** The **TNC** is used in **Cubesat** radio communications, where digital signals are enabled to propagate using radio waves by using the **AX25** packet protocol. Concretely, the **TNC** is connected to a terminal (computer) and a **transceiver**. Data from the computer (concretely, from an software application that provides a command line interface) is formatted into **AX25** packets/frames and modulated into audio tones by the **TNC** in order to be transmitted by the radio (**transceiver**). On the other hand, the received signals are demodulated by the radio and unformatted by the **TNC**, which sends the text output to the computer for displaying in some application. [32]



Figure 3.5 – *TNC (Terminal Node Controller) device [33]*

- **Computer:** it consists of the **GranaSAT** server, located in the “iMUDS Building”. This, connected to the **transceiver** and **TNC**, allows to send/receive **AX25** packets by making use of a command line interface. Its characteristics can be seen in table 3.2.

Hostname:	granasat2.ugr.es
Operative System:	Debian Linux 9
Processor system:	Intel(R) Xeon(R) CPU 5110 @ 1.60GHz, 4 cores
RAM:	17 GB
Local disk space:	207 GB
Kernel and CPU:	Linux 4.9.06amd64 on x86_64

Table 3.2 – *GranaSAT server characteristics*

Hence, while using a **Ground Station**, the basic procedure for communicating and send commands to a **Cubesat** would be the following:

1. From the computer, in a command line interface that communicates with the **TNC**, the user writes the desired message/command to be sent, which is interpreted by the **TNC** as ASCII text.
2. The introduced text is formatted by the **TNC** using the **AX25** protocol, transforming the given data into audio tones.
3. These audio tones are modulated by the **transceiver** and sent through radio waves in the desired frequency band.
4. The **Cubesat** receives these signals, decodes it and perform the operation that corresponds with the command.

The opposite process (receiving information from the **Cubesat**) follows the same philosophy, but on the opposite way:

1. The cubesat transmits the **AX25** packets in a frequency band
2. The antenna of the **Ground Station** receives these signals
3. The **transceiver** demodulates these signals and convert them into audio
4. The **TNC** demodulates the audio that comes from the **transceiver** and decode the **AX25** frames
5. These decoded frames are displayed in the computer in a human-readable format, containing the **Cubesat** telemetry

See figures 3.6 and 3.7 for a better understanding of **Ground Station** communication components.

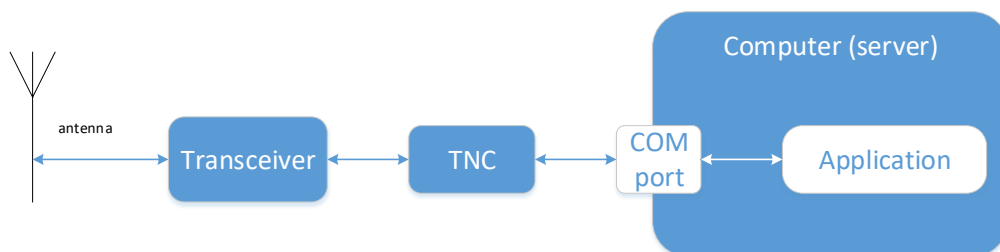


Figure 3.6 – *GranaSAT Ground Station components scheme*

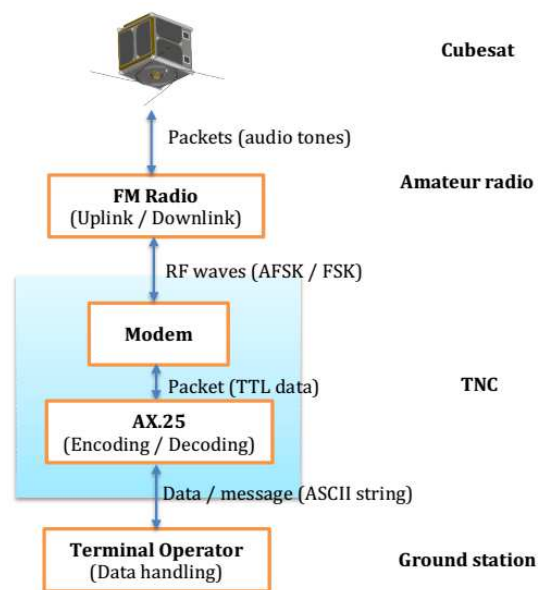


Figure 3.7 – *Communication between Ground Station and Cubesat [35]*

As it was said before, the presented Project aims to control all of this hardware remotely in order to perform communications with a **Cubesat** without being physically using the **Ground Station**. Therefore, this needs to be achieved by making use of the different current technologies, which will be reviewed below.

3.2 Review of Solutions for Ground Station Control

As described above, the objective is to develop an application for remote control of the **Ground Station**, thus, in this section different solution will be analyzed, taking into account type of application, programming languages, possible frameworks, etc.

3.2.0.1 Type of application

When talking about application development, it is important to define what type of application is going to be developed. This election is not usually something trivial and depends on many factors, such as available budget, possible deadline, target users, etc.

Let us take a look to the possible solutions:

- **Native application:** these are applications specifically made for one platform (Android, iOS, Windows, etc). They are developed in languages such as Android, Java, .NET, etc. and it is necessary to know the characteristics of the devices where the applications are going to be working.

- **Web application:** this includes applications that are accessible from the web browser and require Internet connection to work properly. In this case, it is the web browser the responsible of making the application work (and not the device itself).
- **Hybrid application:** these are a combination of the last two, taking advantage of web development and device functionalities, basically like a native application itself, but developed with web technologies.

A detailed comparison of these technologies can be seen in table 3.3

	Pros	Cons
Web	<ul style="list-style-type: none"> Multi-platform No downloads needed Cheaper and faster development 	<ul style="list-style-type: none"> Web server needed Slower performance No access to device functionalities
Native	<ul style="list-style-type: none"> Better user experience Faster and more efficient Access to device functionalities 	<ul style="list-style-type: none"> Higher development costs More development time It relies on App store
Hybrid	<ul style="list-style-type: none"> Web development facilities 	<ul style="list-style-type: none"> Not as efficient as native apps

Table 3.3 – *Application alternatives for Ground Station control*

The chosen solution is expected to meet the following requirements:

- Multi-platform: the application must be accessible and functional from any device, regardless device model, size or operative system
- The development must suit the Project schedule (no more than one academic year) and the student budget
- The user should not have to download anything in order to use the application
- In principle, since the objective is to control [Ground Station](#) remotely, there is no need of using the user's device functionalities

Hence, analyzing the given options, it is clear that the most suitable and appropriate option for the described problem is a **web-based application**, since this will work in every device provided a browser that runs the application. Furthermore, no device will require updates since once the source code of the application is modified and updated, every user will see automatically the last version of the system, which makes it much more efficient than a native/hybrid application.

During the development of [GranaSAT](#) projects, a web application prototype which aimed to control the [Ground Station](#) was started in 2016 [10] [1] [20], referred as [GranaSAT Dashboard](#) from now on.

Some modules were developed, such as the server database with different users and satellites, the log-in system, or the Yaesu rotors remote control.

Hence, the presented Project will resume the above mentioned project, adding new modules and improving the current system.

Web applications are built making use of standard technologies, which are divided in those that are used on the client side (also known as front-end) and those that are used on the server side (also known as back-end).

Since the web application was started in 2016, it has been two years. Therefore, it is necessary to analyze the technologies that have emerged since then and eventually decide whether it is worthy to continue the development with this technologies or otherwise, change these and migrate the web application.

3.2.0.2 Front-end

When talking about the front-end, it refers to the part of the application that the users interact with, that is, the [GUI](#) or the web itself.

Nowadays, [HTML](#), [CSS](#) and [Javascript](#) are the most extended languages for the front-end of web applications, furthermore, they are the most used programming languages according to StackOverflow (see figure 3.8).

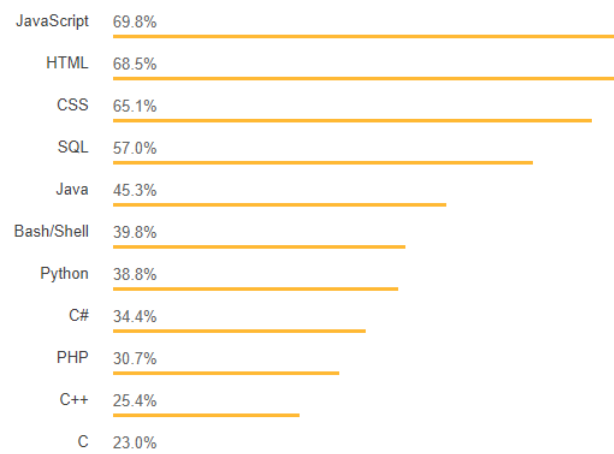


Figure 3.8 – *Most popular technologies during 2018 according to StackOverflow [26]*

Regarding the front-end development, tons of front-end frameworks are currently available. These frameworks make the front-end development faster, easier and more robust. They provide different useful widgets, tools and predefined [HTML](#) elements that do not require great skills to be used.

Before selecting an specific front-end framework, it is important to take into account different aspects, such as the developer's skills level, the desired design, whether the framework allows responsive design, available documentation, etc.

Hence, some of the most used front-end frameworks during 2016 [18] are analyzed in table 3.4.

	Pros	Cons
Bootstrap	Responsive design available Extensive documentation Slight learning curve	Complex customization
Semantic-UI	Minimal load times	Very simple designs Very large packages
Foundation	Great flexibility	Very complex for beginners

Table 3.4 – *Front-end design frameworks comparison*

It is clear that in this case, [Bootstrap](#) was the best possible solution due to the ease that this requires in order to be used, besides the great responsive design that provides. As an important fact, [Bootstrap](#) is the most used open-source front-end framework in the world, what it provides an extensive and wide documentation, besides a great community.

On the other hand, different web frameworks are used in web development in order to provide a standard way to build these. These frameworks provide libraries for database access, session and login management, security aspects, code modularization, etc. Most of these frameworks make use of the [Model-View-Controller](#) pattern, which makes the applications more modular and its code more reusable.

As it has been said before, nowadays [Javascript](#) is the most extended and used programming language, therefore, some [Javascript](#) web frameworks are reviewed in table 3.5.

	Pros	Cons
AngularJS	High scalability A lot of libraries Biggest community	Steep learning curve High size (143k)
ReactJS	Very flexible Small size (43k)	Few libraries
Vue.js	Lightly learning curve Very flexible Small size (23k)	Few libraries Closed community

Table 3.5 – *Javascript web frameworks comparison*

When the presented web application was started in 2016, [AngularJS](#) was an innovative framework designed by Google in 2009 and it was very popular and extended among the developers community. Throughout the last years, more frameworks that follow a similar philosophy that [AngularJS](#) are emerging, such as the above seen ReactJS or VueJS.

These are becoming more popular among the new developers, since these new frameworks have a slightly learning curve. However, nowadays [AngularJS](#) is still one of the most extended and utilized frameworks (with new versions launched, such as Angular2), as it is shown in figure 3.9.

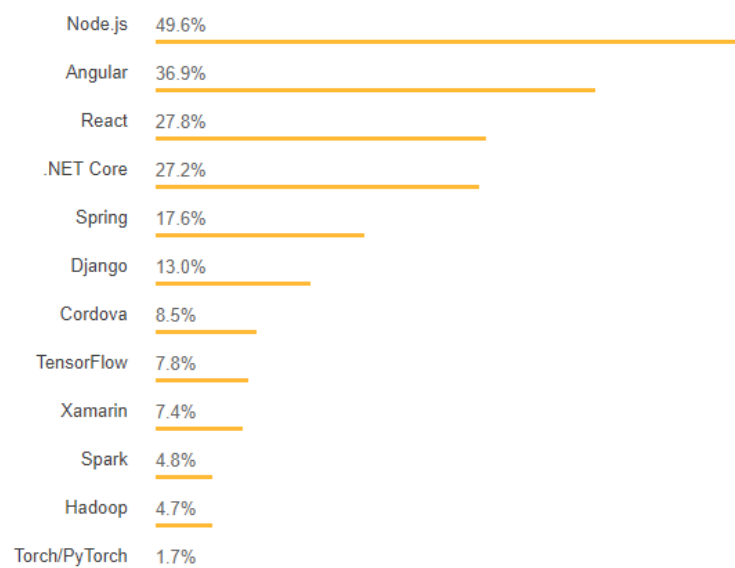


Figure 3.9 – Most popular frameworks, libraries and tools in 2018 according to StackOverflow [26]

Hence, nowadays [AngularJS](#) is considered an appropriate and optimal solution to continue the development of the current system.

3.2.0.3 Server side

When talking about the back-end, it refers to the server part of the application where all the components of the system work and interact with a data base. The back-end does not necessarily need to have the same programming language as the front-end, however, this would be something convenient for the application's modularity and structure.

Before choosing a technology or framework for the back-end, it is necessary to take into account different aspects, such as the programming language to be used and its ease to learn it, the amount of libraries that are available in order to solve the presented problems, performance taking into account the purpose of the desired system, etc.

A table with different technologies for the back-end can be seen in 3.6.

	Pros	Cons
JavaScript (NodeJS)	Largest library registry Easy to learn Active and huge community Javascript full stack allowed	Use of callbacks (messy code) Unsuitable for large applications Low scalability
Python (Django)	Easy and fast development High scalability Security is key	Monolithic Much code for small projects
Ruby on Rails	Easy and fast development Very flexible	Slow performance Difficult to find good docs High computer resources
PHP	Simplicity Tons of available frameworks Large open source community	Low performance It might be insecure

Table 3.6 – *Javascript back-end technologies comparison*

As it is seen in the table above, [NodeJS](#) was the best possible solution for the back-end and nowadays it would have been as well, even with more reason, specially for its big amount of available libraries, ease of use and potential. Actually, as it was seen in figure 3.9, nowadays it is one of the most used technologies.

Before [NodeJS](#) was launched, it was strictly necessary to use a different language from [Javascript](#) for the back-end. Nowadays, it is possible to program the front-end and back-end in the same language thanks to the launch of [NodeJS](#), which makes it definitely the future in web programming.

Regarding the database management system that will be used in the back-end, many options are currently available.

When choosing a specific data base, it is necessary to take into account factors such as size of the data to be stored, speed and scalability, amount of people that will access the data, security aspects, etc.

The most utilized data bases during 2018 is shown in 3.10.

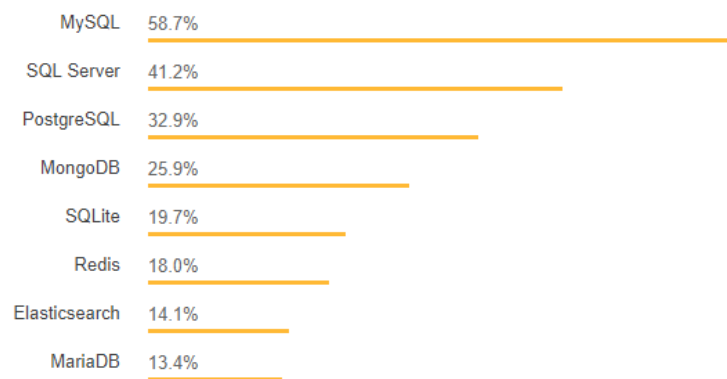


Figure 3.10 – Most utilized data bases during 2018 according to StackOverflow [26]

Hence, a brief comparison of some of the most used databases is shown in table 3.7.

	Pros	Cons
MySQL	High security level Standard in production system	It is not community driven anymore
PostgreSQL	High scalability Many available interfaces	Slower performance Tricky documentation
MongoDB	Flexible schemas High scalability Easy to use	It does not use SQL language Less query flexibility Default setting not secure
SQLite	Server independent Single file to store data Cross-platform database file	Not recommended for large apps

Table 3.7 – Databases comparison

It is clear that **MySQL** has been always an standard in production systems, therefore, this will be the used data base when the application is running on the server.

On the other hand, as it can be inferred from the table above, **SQLite** is a database that works independently from the server, with a single file that stores the data. This means that this data base can be used when developing locally, being able to transfer this file easily from one computer to another, which makes the development really agile and efficient.

In summary, as a result of all the chosen technologies that will be used, a brief scheme with the web application architecture and technologies is shown in figure 3.11.

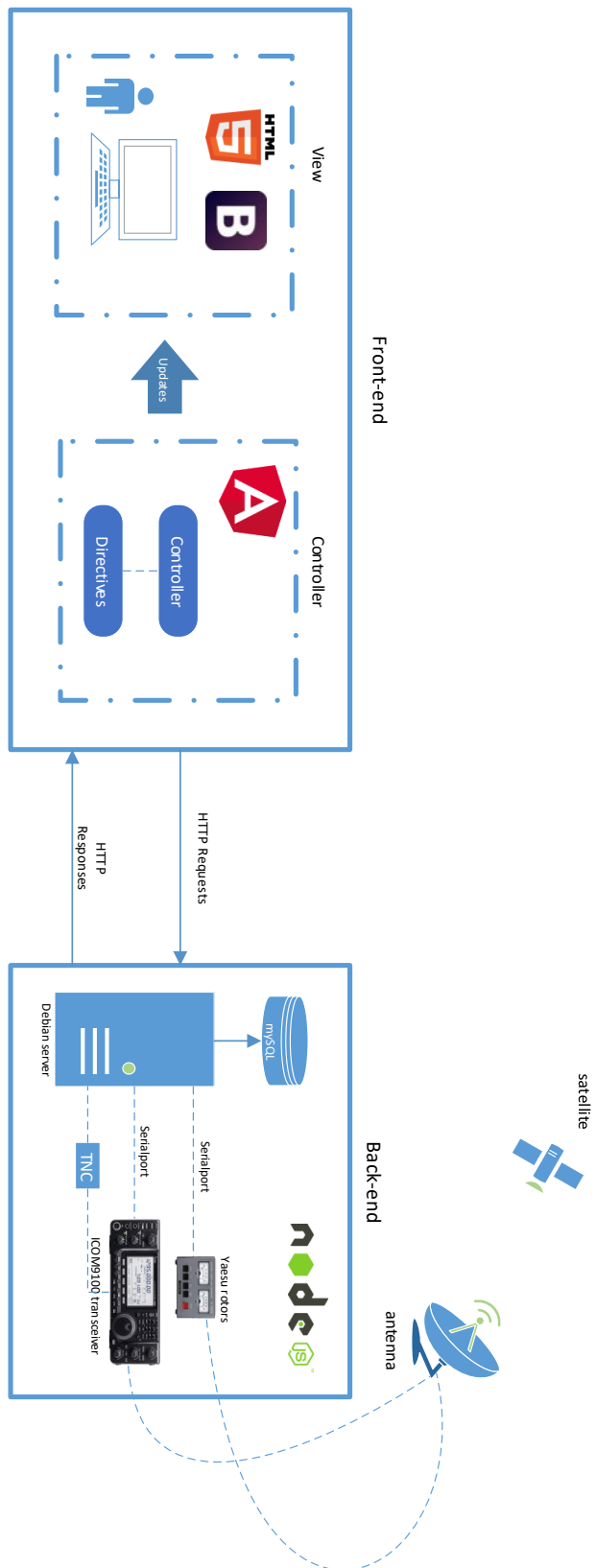


Figure 3.11 – GramaSAT Dashboard system architecture and technologies

Having chosen the technologies that will be used in the system, the following sections will try to analyze in detail how the different system requirements given in 2.1 will be met. Specifically, different approaches will be treated and analyzed, eventually choosing the most suitable for its later design.

3.3 Transceiver Telecontrol

As described in requirement a), users should be able to control the [transceiver](#) remotely, without being physically controlling it. In this case, the [transceiver](#) will be controlled from the web-based application.

This telecontrol is possible thanks to the USB Icom CI-V Interface, which allows the [transceiver](#) to be connected to a computer via serial port (in this case the [GranaSAT](#) server) and so be programmable.

On the one hand, it will be necessary to design a [transceiver GUI](#), that is, a front-end that tries to simulate the real aspect and behavior of the [transceiver](#). From this front-end the user will interact remotely with the [transceiver](#).

On the other hand, the back-end needs to be designed as well. The back-end is intended to communicate with the hardware (in this case the [transceiver](#)) and be able to send the proper commands to this via serial port, using the proper protocol described in [14]. Thus, the following analysis is performed in order to describe what technologies are appropriate and how this problem can be approached.

3.3.1 Back-end

To begin with, it is necessary to know the communication protocol between the [transceiver](#) and a computer (in this case the server) so that the [transceiver](#) is controllable from this. This will allow later to control the transceiver from the [Dashboard](#), since clients will perform requests to the server, which will communicate with the [transceiver](#).

As described in [14], a computer connected to the [transceiver](#) can send and receive byte to byte to and from it. Specifically, the way these bytes are sent/received is specified in the Icom9100 instruction manual and it is strictly necessary to follow its protocol.

A table with the data format that must be used is shown in 3.12.

◇ **Data format**

The CI-V system can be operated using the following data formats. Data formats differ depending on command numbers. A data area or sub command is added to some commands.

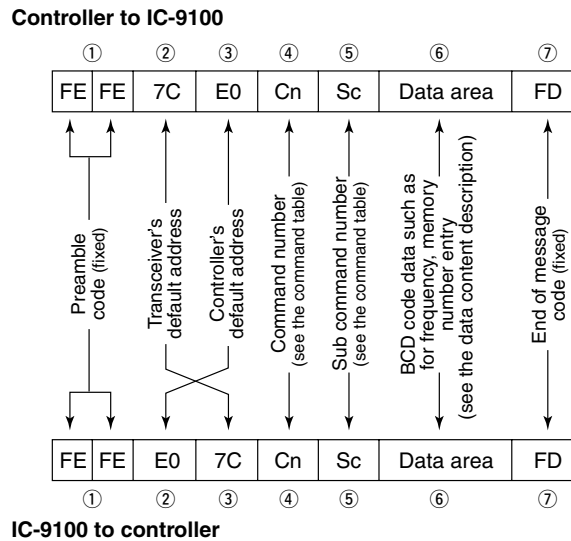


Figure 3.12 – Icom9100 data format [14]

As it is shown above, the commands are sent with certain format, being the sections 4,5,6 the ones that need to be changed according to the commands that are used. To specify these commands, it is necessary to look at the command table in [14] (pp 184-190).

A partial example of this table is shown in table 3.13

◇ **Command table (continued)**

Cmd.	Sub cmd.	Data	Description
15	01	00	Read squelch status (squelch close)
		01	Read squelch status (squelch open)
	02	0000 to 0255	Read S-meter level (0000=S0, 0120=S9, 0240=S9+60 dB)
	11	0000 to 0255	Read RF power meter (0000=0%, 0141=50%, 0215=100%)
	12	0000 to 0255	Read SWR meter (0000=SWR1.0, 0041=SWR1.5, 0081=SWR2.0, 0120=SWR3.0)
	13	0000 to 0255	Read ALC meter (0000=Min. to 0120=Max.)
	14	0000 to 0255	Read COMP meter (0000=0 dB, 0120=15 dB, 0240=30 dB)

Figure 3.13 – ICOM9100 command table [14]

Following the data format and command table described above, let us see a simple example:

In the [transceiver](#), the squelch is used in order to set the noise threshold, that is, squelch removes noise output when no signal is received. Thus, in order to read the squelch status (open/closed), it is necessary to send the command:

$$FEFE7CE0 \quad \underbrace{15} \quad \underbrace{01} \quad FD$$

command number subcommand number

On the other hand, if the squelch is open, it is expected to receive:

$$FEFEE07C1501 \quad \underbrace{01} \quad FD$$

squelch open

otherwise, if this is closed:

$$FEFEE07C1501 \quad \underbrace{00} \quad FD$$

squelch closed

This is how the Icom9100 [transceiver](#) can be controlled from a computer by using its protocol. The [transceiver](#) has tons of functions, modes and parameters but logically, only those that will be used will be implemented.

Specifically, the following functions will be implemented:

- **Frequency:** The Icom9100 can operate on the HF/50 MHz, 144 MHz, 430 MHz and 1200 MHz frequency bands. These frequency bands can be assigned to the MAIN (for transmission/reception) and SUB Band (only reception) for operating convenience, however, the frequency band, selected in either the MAIN or SUB band, cannot be selected on the other band.

Users should be able to choose the operative frequency (Main or Sub), and change the frequency bands by their convenience.

- **S-meter:** it indicates the signal strength while receiving signals. The user should be able to see the s-meter indicator.
- **RF power meter:** it indicates input power while receiving. The user should be able to see the RF power indicator.
- **SWR:** the SWR of the antenna of the antenna. The user should be able to see the SWR indicator.
- **ALC:** it indicates the micro gain during transmission. The user should be able to see the ALC indicator.

- **Attenuator:** in order to prevent desired signal from being distorted when very strong signals are near the signal's frequency. The user should be able to activate/deactivate the attenuator.
- **AF:** in order to increase/decrease the audio output level. The user should be able to adjust the AF level.
- **RF power control:** it indicates the transmit output power while transmitting, in Watts. The user should be able to adjust the RF power.
- **NR (Noise Reduction):** function that reduces random noise components and enhances audio signals which are buried in noise. The user should be able to activate/deactivate noise reduction.
- **Squelch:** The squelch removes noise output to the speaker when no signal is received. The user should be able to adjust the squelch threshold.
- **Operating mode:** the Icom9100 has different operative modes (CW,USB,LSB, AM, FM, RTTY and DV). The user should be able to modify the operating mode.
- **Satellite mode:** the satellite mode allows to both downlink and uplink frequencies simultaneously increase or decrease in the same steps when you change the frequency band. The user should be able to activate/deactivate the satellite mode.
- **Transmission:** the Icom9100 turns into transmission mode when the user presses the PTT (Press To Talk, that is, the microphone) or when transmission mode is activated. The user should be able to activate/deactivate the transmission mode.
- **Duplex mode and duplex offset frequency:** some repeaters can be accessed using the duplex operation to set the frequency shift to the same value as the repeater's frequency offset. The user should be able to set the duplex mode and the duplex offset frequency in order to access repeaters.
- **Repeaters sub-tones:** Some repeaters require a sub-audible tone to be accessed. Sub-audible tones are superimposed on the normal signal and must be set first. Icom9100 has 50 tones from 67.0 Hz to 254.1 Hz. The user should be able to change the repeater sub-tone in order to access repeaters.

As it was said before, the communication between server and [transceiver](#) is done via serial port, therefore, some library will be needed in order to make use of the serial port from the server.

As described in [3.2.0.3](#), the back-end of the system will be programmed in [NodeJS](#), therefore, the most suitable solution will be to use the standard [NodeJS](#) library that is called **Node-SerialPort** [21], which provides a set of functions to open, write and read a serial port.

3.3.2 Front-end

Once it is known how the [transceiver](#) protocol works and how to communicate server and [transceiver](#), the next step is to analyze how to implement the front-end, that is, what technologies will be used in order to define the aspect of the application, what the user will see and what the user will interact with.

In summary, the client side will be basically a [GUI](#) that will simulate the [transceiver](#) behavior in a simple and handy way for the user. Taking a look to the [transceiver](#) appearance (see [3.4](#)), it is noticeable that there is a main LCD screen, besides some buttons and rotors. In order to make the front-end as real as possible, some graphical elements that will simulate the real ones need to be defined.

- Rotors: while reviewing, some open-source [Javascript](#) graphic libraries that could be used to design these were found. Some of them are:
 - **gauge.js**: simple [Javascript](#) library that allow us to use gauges with needles. [[11](#)]
 - **justGauge**: it is a handy [Javascript](#) plugin for generating and animating nice and clean gauges. It is completely resolution independent and self-adjusting. [[17](#)]
 - **D3.js**: this is a really complex [Javascript](#) library for manipulating documents based on data. It has many graphical examples, among those, gauges. [[8](#)]

A brief comparison of these libraries is shown in table [3.8](#)

	Pros	Cons
gauge.js	All in one js file Flexible customization Easy to use	Very simple design
justGauge	High customization SVG elements	Poor documentation It might be deprecated
D3.js	High customization Huge community and documentation SVG elements	Complex library

Table 3.8 – *Javascript gauges analyzed possibilities*

Analyzing these options, **gauge.js** will be the chosen option, since justGauges seems to be a little bit deprecated and therefore future problems could arise during the design stage. On the other hand, D3.js gauges designs do not really fit with the expected design idea and it would be really complex to adjust its designs, due to the complexity of this library.

- LCD screen: since the LCD screen contains bars and some legends, [Bootstrap](#) default bars provide an easy and straightforward solution for this purpose. Regarding the legends, [SVG](#) images could be inserted. These could be designed with certain graphical design software such [Inkscape](#), taking advantage of its free license.
- Buttons: once again, native [Bootstrap](#) buttons could be used since there is no need of anything else.

The implementation of the [transceiver](#) protocol on the back-end and proper [GUI](#) on the front-end will be designed in section [4.1](#)

3.4 Transceiver's Audio

As described in requirement d), the [Dashboard](#) is expected to receive and play the [transceiver](#)'s audio in live, so that the user can listen the audio that is really coming out of the [transceiver](#) remotely. An special section is necessary to analyze this problem, since there is no specific protocol in the [transceiver](#)'s manual to approach this problem.

Specifically, the presented problem is how to send an audio stream from the server to the client with an acceptable audio delay.

Hence, different alternatives were reviewed and some of them are described below, choosing the most suitable for the system.

- **audio websockets + web Audio API:** this solution consist of getting the [transceiver](#)'s audio stream output (raw data) and redirect it to the client by using a websocket (for what an external library is needed). On the client, connecting to the websocket and interpret the raw data with the web Audio API would be necessary, in order to be able to play the desired audio with the proper format.
- **VLC:** the well known multimedia player allows to stream audio and video of any device in the system with several methods, being the [HTTP](#) protocol among those.
- **Icecast2 + Darkice:** Icecast2 is an open-source streaming media server that supports different audio formats. On the other hand, Darkice is a live audio streamer that can capture audio from an audio interface, sending the captured audio to a streaming server (Icecast2 in this case). It has been used since 2002 and it is very stable. [7] [13]

A brief comparison of these technologies is shown in table 3.9

	Pros	Cons
websockets	Low audio delay (1/3s) Tons of websockets libraries Good documentation	Very complex
VLC	GUI available	Tricky configuration
Icecast2+Darkice	Very easy to use Good documentation and community Many clients at the same time	Some audio delay (3/5s)

Table 3.9 – Audio streaming from server to client analyzed possibilities

An important aspect to take into account is the audio delay that the system will have. Concretely, an audio delay bigger than 8-10 seconds could not be accepted, since this would be really annoying for the user due to the lack of real and updated information that the user would have.

With the given solutions, almost a real time audio could be obtained with websockets, nevertheless, this would be much more complex than using Icecast2 together with Darkice, with what only couple of seconds more of delay are expected, something totally affordable.

Therefore, with the presented options, the most suitable solution for the presented problem is to use the streaming server **Icecast2 together with the Darkice client**.

The specific implementation and integration of these software on the [Dashboard](#) will be seen in section [4.2](#).

3.5 TNC

As it has been seen before, the system will have to integrate a **TNC** in order to send/receive **AX25** packets, which will contain the telecommands/telemetry that will be sent/received to and from a **Cubesat**.

Therefore, as specified in requirements b) and c), the user is expected to use the **Dashboard** to send and receive **AX25** frames/packets by making use of a **TNC**.

To approach this problem, two possible solutions are suggested:

- **Using a physical TNC via serial port:** as it was seen in the previous section, the **transceiver** can be controlled remotely by making use of a serial port. Hence, this approach could be used in the same way for controlling a physical **TNC**.
- **Virtual TNC:** Using a software that substitutes/simulates a real **TNC**, which would be running on the terminal (**GranaSAT** server in this case). In this case, the physical **TNC** is substituted by a software that takes advantage of the computer hardware in order to perform the same operations that would be done by a real physical **TNC**.

Given these options, it is clear that using a physical **TNC** would be more complex than using a substitute software, since the configuration would be probably more sophisticated than using an up-to-date software. Furthermore, a virtual **TNC** will provide a much more flexible configuration than a physical one.

Therefore, since everything must be done as simple as possible, the most suitable solution is to use a software that substitutes a physical **TNC**.

While reviewing, it was found that this kind of software is not really popular and it is quite difficult to find a stable and supported software. Specifically, two up-to-date alternatives were found and they are reviewed in table 3.10.

	Pros	Cons
Direwolf	Flexible configuration Frequent updates Good and extensive documentation	No GUI available
Packet Engine Pro	GUI available	Available for Windows and MacOS Requires a 30\$ fee after 30 days

Table 3.10 – Virtual **TNCs** analyzed possibilities

With the comparison described above, it is clear that **Direwolf** is the best and unique option, since this is the only available option for Linux, which is the server's operative system.

Another aspect to take into account is that when communicating with a **TNC** (in this case **Direwolf**), it is necessary to do it with a special protocol, which is known as **KISS**

protocol.

Physical TNC's provide some line-command applications that implement this protocol and make the communication with the TNC easier, users needing only to write the data (normal ASCII text input) that they want to format into AX25 packet. These applications communicate to the physical TNC via serial port.

In this case, the server will need to communicate with Direwolf. Hence, this communication could be approached by making use of an external library that allows to communicate with a TNC.

Two options were found while reviewing, one of them is a Python library that implements the KISS protocol (<https://github.com/ampledata/kiss>), while the other option is a Javascript library [9] that could be easily used in the NodeJS server and therefore, it will be the chosen solution.

Lastly, since Direwolf will be running in background when the Dashboard is launched and it is necessary to display its output on the Dashboard. The easiest approach for this is to send the Direwolf output from the server to the client making use of websockets, sending the Direwolf output to the client as a simple text/string.

This Direwolf output will be displayed on the Dashboard making use of a simulated terminal that was developed by a GranaSAT colleague [10].

Direwolf integration and implementation on the Dashboard will be seen in section 4.3.

3.6 Yaesu Rotors Telecontrol

As it was said before, the [Dashboard](#) was started in 2016 and some of the parts that were already designed was the Yaesu rotor's remote control.

Nevertheless, in order to complete the system analysis and make everything make sense, a brief description about the rotors implementation will be done.

As it has been seen in section [3.3](#), the [transceiver](#) is connected to the server and so it can be programmable via serial port.

Hence, the Yaesu rotors are programmable via serial port as well, which allows to synchronize the Yaesu rotors and some software satellite tracking in order to modify the antenna's position and point it to the desired current satellite's position.

3

3.6.1 Back-end

Contrary to the [transceiver](#) telecontrol, which have many possible commands, Yaesu Rotors protocol is much easier and it has the following commands:

- **C2**: it returns the current antenna position ([elevation](#) and [azimuth](#)).
- **Waaa + eee**: set the current antenna position to “aaa” azimuth (0-420°) and “eee” elevation (0-90°)
- **S**: stop the rotors

3.6.2 Front-end

The current front-end to manipulate the Yaesu Rotors is quite simple (plain text), therefore, this will be modified in order to show the current [elevation](#) and [azimuth](#) with customized gauges.

As it was already analyzed in [3.3.2](#), there are many available option for implementing gauges.

In this case, the most suitable option will be the **D3.js** library, due to its high customization and because it is considered to provide the appropriate design to display the antenna's [azimuth](#) and [elevation](#).

The implementation of these aspects will be seen in section [4.4](#).

3.7 Satellite Tracking

Every **Ground Station** must have a satellite tracking software in order to allow users to know what is the current position of the satellite. This is necessary since the satellite has a limit coverage area and it would not be possible to perform communications with it if the **Ground Station** is not inside the satellite's coverage area.

Furthermore, the satellite tracking allow to aim the antennas to the current satellite's position, since current **elevation** and **azimuth** are provided.

Therefore, the current satellite's position must be before attempting any communication.

Nowadays there are many available satellite tracking software, being GPredict (<http://gpredict.oz9aec.net/>) or JSatTrak (<http://www.gano.name/shawn/JSatTrak/>) some of the most popular tools.

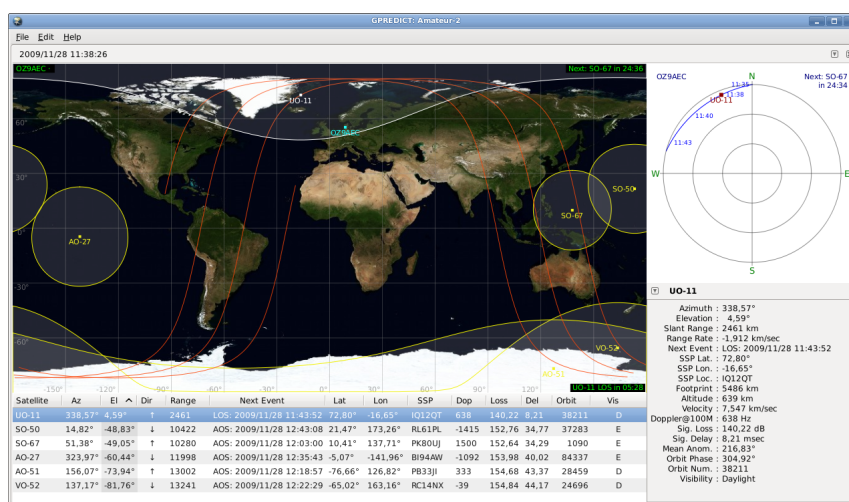


Figure 3.14 – GPredict satellite tracking software

Hence, as described in requirement d), the system is expected to have its own satellite tracking in real-time so that the user can see what is the position of the **Cubesat**. In addition, in order to not to have just a simple tracking, some parameters are expected to be able for the user, such as the satellite's longitude, latitude, altitude, **azimuth** and **elevation** with respect to the **Ground Station**, coverage area of the satellite, etc.

Regarding how to develop the satellite tracking system, some aspects need to be taken into account:

3.7.1 Extraction of satellite data

In order to extract all the satellite's data, some library needs to be used. Currently there are many open source libraries for this purpose, most of them written in **Python** due to its scientific use. Some of these libraries are libpredict, orbit-predictor, PyOrbital, PyEphem or

SGP4.py. Furthermore, there are also [Javascript](#) libraries that are directly based on these (for example [satellite.js](#) or [jspredict](#)), which is a good straightforward solution taking into account that the application is running in [NodeJS](#).

Some of these libraries have been analyzed and the chosen ones will be:

- **satellite.js** [25] : this library is really complete and it will provide most of the needed data, such as velocity of the satellite, longitude, latitude, [azimuth](#), [elevation](#) and height.
- **jspredict.js** [12] : this library will be used to get the coverage area of the satellite and whether this is on light or night position.
- **PyEphem** [24] : the satellite's orbit will be calculated with this library, since it is really easy to perform, only needed couple of "for" iterations with N interval times that define the satellite's orbit in the future and in the past (where the satellite has already been).

All of this information is calculated in these libraries by making use of the well-known "two-line elements" developed by [NASA](#) and [NORAD](#). Also called [TLE](#), this is composed of two lines with several parameters that define the orbit of a satellite [3]. These parameters are: size and shape of the orbit, orientation, epoch year, etc. (an example of [ISS TLE](#) can be seen in figure 3.15).

The described [TLE](#) is used as an input in the libraries described above for the algorithm [SGP4](#), which calculates future satellites orbits.

These [TLEs](#) need to be updated frequently, since these lose precision over time. TLEs of public satellites are available from pages such <https://celestrak.com/>, which provides many available [TLE](#)'s in single .txt files.

Regarding the [GranaSAT Cubesat](#), its [TLE](#) would be provided by the launch provider. More accurately, a pre-launch [TLE](#) (calculation based on the expected orbit) would be provided by the launch provider and it would be used until the post-launch [TLE](#) would be released from the Department of Defense Joint Space Operations Center, based on observations made with cameras and radars [15].

```
ISS (ZARYA)
1 25544U 98067A 08264.51782528 -.00002182 00000-0 -11606-4 0 2927
2 25544 51.6416 247.4627 0006703 130.5360 325.0288 15.72125391563537
```

Figure 3.15 – *TLE example* [34]

Since there is no [Cubesat](#) yet in orbit, the satellite tracking will be tested with current real satellites, such as [ISS](#) or other [Cubesat](#) such as e-ST@R-2 ([Cubesat](#) owned by the University of Turin).

3.7.2 Display over interactive map

Once the satellites data is retrieved, this needs to be displayed over a map on the [Dashboard](#). To do this, there are tons of different open source solutions that deal with interactive maps making use of [Javascript](#) [4].

A comparison of some of these map libraries is shown in table 3.11.

	Pros	Cons
OpenLayers	High flexibility Maturity Easy to use	Complex configuration
GMaps	High customization Good documentation Google Maps API	None
Leaflet	Basic use Good and extensive documentation Mobile friendly Small size (38K)	None

Table 3.11 – *Javascript libraries for interactive maps analyzed possibilities*

In summary, all of these libraries are quite similar, therefore, among these alternatives, **Leaflet** [19] will be the chosen solution, due to its small size (38Kb) and ease of use, besides being suitable for mobile devices.

Leaflet allows to use different map layers from different web pages, in this case, the map layer is taken from [Mapbox](#), for what a key-token will be needed in order to use its [API](#). For this, previously a user account has been created in [Mapbox](#). To define the satellite's orbit, coverage area and satellite situation, it is possible to define lines, circles, polygons, etc. by making use of the [Leaflet API](#).

Lastly, a light-night layer will be set over the map, so that the user can see which part of the Earth is on light and night in every moment. For this, there is an open source and straightforward solution [16] that can be used with [Leaflet](#).

3.7.3 Integration within the system

Once the technologies are properly chosen, it is necessary to think about how everything is going to be integrated within the system.

Regarding the client part of the system there is no need of a deep analysis, since with [AngularJS](#) can deal with the map display, making use of [Leaflet API](#). The client part of

the system will ask for the satellites data to the server (making [HTTP](#) requests), which will return a [JSON](#) object to the client as a response.

This will contain all the information, information that will be visually displayed on the client.

On the other hand, the back-end is developed in [NodeJS](#) and [Python](#) code needs to be executed (in order to use PyPredict), therefore, some library needs to be used in order to do this from [NodeJS](#).

While reviewing, two different options were found:

- **spawn childprocess:** standard module in [NodeJS](#) that allows to run a [Python](#) script.
- **PythonShell.py:** external library that simulates a [Python](#) terminal in order to execute its code.

Both are useful to execute the [Python](#) script that will collect all the data satellites, therefore, the **spawn module** will be used since it does not require an external library.

Further details about implementation will be described in section [4.5](#)

3.8 Other Improvements

As described in requirements f), g) and h), different aspects of the [Dashboard](#) will be improved adding several functionalities that will make a better system. To do this, different available technologies are studied and finally, the most suitable will be chosen and implemented in section [4.6](#).

3.8.1 Tooltips definition

A tooltip is a graphical element (a little box) that pops up when the user hovers the pointer over an item (image, hyperlink, etc). Its function is to provide extra information to the user.

There are many different available options and libraries to implement tooltips, specifically, three possible solutions will be taken into account and eventually one of them will be implemented.

- **Default tooltips:** it is the default tooltip in [HTML](#), it is easy to use by adding “title” attribute to a [HTML](#) tag.
- **tooltips.js:** this is an open-source library for tooltips implementation. It provides an [API](#) to create, show, hide and toggle customized tooltips [[27](#)].
- **Twitter [Bootstrap](#) tooltips:** the well known framework [Bootstrap](#) allow us to implement tooltips by using its already defined classes, besides giving the possibility of customization. [[28](#)].

A brief comparison of these technologies is shown in table [3.12](#)

	Default tooltip	tooltips.js	Bootstrap tooltip
Pros	Native in HTML Easy to implement	High customization	Framework already in use Uniformity
Cons	No customization	Library needed	

Table 3.12 – *Tooltips analyzed possibilities*

Taking into account the characteristics described above, [Bootstrap](#) tooltips will be the chosen solution, specially for the consistency and uniformity that this provides to the system, since [Bootstrap](#) is one of the frameworks that has been used in the system from the beginning of the development.

The implementation of tooltips is seen in section [4.6.1](#).

3.8.2 Camera streaming on the server

In this section different available tools are analyzed in order to set up a cameras system on the server. Due to the complexity of the problem and the purpose that the cameras will have, it is not extremely necessary to have complex cameras, therefore, simple webcams will be used.

The purpose of the first camera will be to stream a video of the antenna, thereby, the user will be able to see how this is changing its elevation and azimuth when the user is making the proper changes on the [Dashboard](#).

Regarding the second camera, this will be set up in the laboratory and its function will be to watch the [Ground Station](#) and detect movement, recording video that will be saved on the server.

Three possible applications that meet the given requirements are the following:

- **VLC:** the widely known multimedia player. This program is open source and it includes the possibility of streaming video and movement detection, besides many other options related to multimedia playing (DVD, audio, etc).
- **Motion:** this software is open-source and it allows streaming, recognition of movement and video recording. It is highly configurable and it can be easily used and installed in Linux from the terminal. Furthermore, it includes a [GUI](#) that can be displayed in a server port.
- **ZoneMinder:** this is a really complex open source surveillance software. It includes many options such as monitoring, recording, movement detection, scheduling, etc. This software includes a [GUI](#) as well.

A brief comparison between their characteristics can be seen in table [3.13](#).

Software	VLC	Motion	ZoneMinder
Size	200Mb	6Mb	8Mb
OS	Windows, MacOS, Linux	MacOS, FreeBSD, Linux	FreeBSD, Linux
Multiple cameras	Yes (multiple instances)	Yes	Yes
GUI on server port	No	Yes	Yes

Table 3.13 – *Video streaming software analyzed possibilities*

Given these options, the best considered option is **Motion** for several reasons:

- It is very flexible and provides easy configuration by `.config` files
- Easy to install from the terminal
- It allows multiple cameras without running multiple instances
- Small size suitable for the server
- Attractive **GUI** to be displayed in a server port

The implementation of these aspects are seen in section [4.6.2](#).

3.8.3 User account management

In the last version of the [Dashboard](#) it was only possible to create an user, not being able to change their password, username, organization, etc. Therefore, the system is expected to meet this requirements.

For this, it is only necessary to create a front-end with a form where users can modify their profiles. After this, the controller ([AngularJS](#)) will verify that the data is correct and will send the data to the server in order to modify the user's data in the database.

In addition, the user is expected to have the possibility of changing their passwords, receiving an e-mail with a new temporary password. This will be done with **node-mailer**, a [NodeJS](#) module that allow us to send e-mails to several destinations.

Further design aspects will be seen in section [4.6.3](#).

3

CHAPTER

4

SYSTEM DESIGN

In this chapter, all the parts of the system that were analyzed in chapter 3 will be implemented, therefore, the level of abstraction at this stage of the development will be as lowest as possible.

As it was explained during the analysis, on the client side [HTML](#), [CSS](#), [Bootstrap](#) and [AngularJS](#) are used. [NodeJS](#) is used on the server side.

The application is divided in tabs, being every tab a directive of [AngularJS](#), which is made up of its [HTML](#) view and the script that defines the behaviour of the tab and acts as a controller.

The main `index.html` file exchanges every tab according where user clicks whilst there is a general controller (`mainController.js`) that controls the general behavior of the application (logging status, tab selected, user information, etc.).

On the other hand, the `app.js` file will be the server logic, that is, here are defined the [NodeJS](#) routes that handle the [HTTP](#) requests from the clients.

These requests are intended to access the data base, to control the [transceiver](#), to control the [TNC](#), to request satellite's data, etc. There is a "module_name".js on the server for every logic part of this, so for the [transceiver](#) control, there is a module, as well as for the rotors control, satellite tracking, [TNC](#) communication, etc.

See image [4.1](#) to see the system architecture.

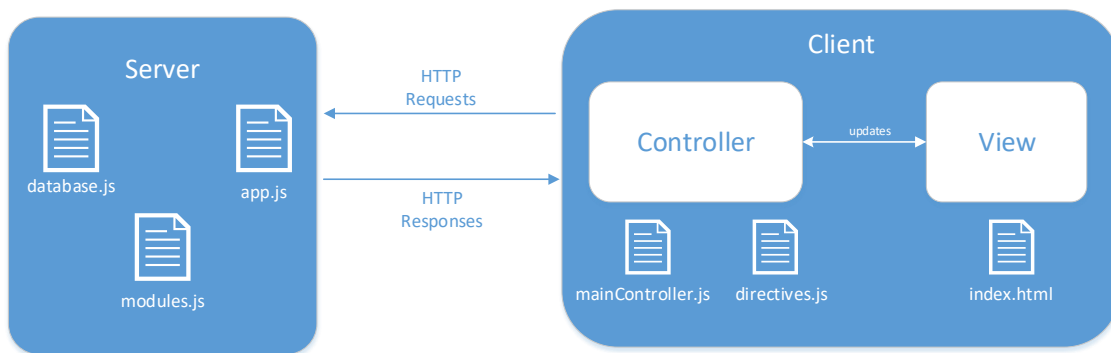


Figure 4.1 – *System architecture*

Hence, in the following sections, when a new front-end is created, this means that a directive of [AngularJS](#) is created, with a [HTML](#) view and its [Javascript](#) script (controller) associated.

On the other hand, when the server needs to perform some operation, new [NodeJS](#) routes are created, besides the proper modules `.js` to interact with within the server.

4.1 Transceiver Telecontrol

As described in section 3.3, the system will include functionalities and commands that will allow to control the [transceiver](#) remotely. This is possible due to the possibility of sending commands via serial port to the [transceiver](#), using format tables described in sections 3.13 and 3.12.

Here, two parts are pointed out: the server part, where [NodeJS](#) makes the communication with the [transceiver](#) by using the serial port and on the other hand, the client part, where the user can make use of all the functions that are displayed by interacting with the front-end.

4.1.1 Back-end

First of all, a class is defined in order to represent the [transceiver](#) and its functions. Here the library [Node-SerialPort](#) [21] is used in order to send and read commands to and from the [transceiver](#) by using the physical serial port that is connected from the [transceiver](#) to the server.

It is important to notice that the user can control the [transceiver](#)'s functions remotely and this will be physically reflected in the [transceiver](#), but it needs to be taken into account the same process in the opposite direction, that is, if someone modifies the [transceiver](#), the [Dashboard](#) must reflect these changes.

Therefore, the [Dashboard](#) client will ask every second for the [transceiver](#)'s attributes. Since [Javascript](#) is asynchronous, a problem arises while reading in the serial port, since this can be only accessed by one thread at the same time. Therefore, the most suitable solution is to define the [transceiver](#) parameters and update these every second by applying the commands one by one.

To apply these one by one, [Javascript](#) promises can be used, which allow to execute these functions in a way that the next function will not be executed until the previous one has finished, therefore, the serial port will be accessed only once every time.

When these functions read the serial port and retrieved data from the [transceiver](#), they update the attributes of this. When the client asks for the parameters, they are just returned with the common get functions. Therefore, from the server it is only necessary to consult these attributes that will be updated every second.

4.1.2 icom9100.js class

This class represents the `transceiver` behavior. The `transceiver` has some attributes that define its behavior. These are the following:

```

1|   var parameters = {
2|     freq: null,
3|     s_meters: null,
4|     rf_meter: null,
5|     swr: null,
6|     alc: null,
7|     comp: null,
8|     att: null,
9|     tone_squelch : null,
10|    tone_squelch_freq : null,
11|    repeater_tone : null,
12|    repeater_tone_freq : null,
13|    rf_power_position: null,
14|    af_position : null,
15|    rf_gain_level : null,
16|    sat_mode : null,
17|    sql_status : null,
18|    sql_position : null,
19|    nr : null,
20|    transceiver_status : null,
21|    operating_mode : null,
22|    offset_freq : null,
23|  };

```

The `transceiver` class has functions that access the serial port in order to write and read commands. When writing or reading commands, the above parameters are updated so that the server knows every moment the current state of the `transceiver`.

Every function receives a callback, which is a function that will be call back when the operation finishes. This callback will contain the operation result: “Done” or “Error” in case it is a SET function. On the other hand, if it is a GET function, the callback will contain the proper result (`Javascript` is interpreted so there are not types, however these will always return an `String`) or “Error” in case the GET functions fail.

Some of the most important functions are the following:

- `getFrequency() : String`: it returns in a callback the current frequency or an error, if applicable
- `setFrequency(freq : String) : String` : it sets a new frequency and returns the operation result in a callback
- `getOperatingMode() : String` : it returns in a callback the current operating mode or an error, if applicable
- `setOperatingMode(mode : String) : String` : it sets new operating mode (“USB,LSB,CW,AM,FM,RTTY,DV”) and returns the operation result in a callback
- `getAttenuator() : String` : it returns in a callback the current attenuator status (“on/off”) or an error, if applicable
- `setAttenuator(status : String) : String` : it sets new attenuator status (“on/off”) and returns the operation result in a callback
- `getNoiseReduction() : String` : it returns in a callback the current noise reduction

status ("on/off") or an error, if applicable

- **setRepeaterToneFreq(freq : String) : String** : it sets new repeater tone frequency and returns the operation result in a callback

- **getRepeaterToneFreq() : String** : it returns in a callback the repeater tone frequency or an error, if applicable

- **setDuplexOffset(freq : String) : String** : it sets new duplex offset frequency and returns the operation result in a callback

- **getDuplexOffset() : String** : it returns in a callback the duplex offset frequency or an error, if applicable

- **getRFPower() : String** : it returns in a callback the rf power value, or an error, if applicable

- **setRFPower(value : String) : String**: it sets new rf power value and returns the operation result in a callback

- **getAF() : String** : it returns in a callback the af position value, or an error, if applicable

- **setAF(value : String) : String**: it sets new af position value and returns the operation result in a callback

- **getRFGainLevel() : String** : it returns in a callback the rf gain level value, or an error, if applicable

- **setRFGainLevel(value : String) : String**: it sets new gain level value and returns the operation result in a callback

- **getSQLPosition() : String** : it returns in a callback the squelch position value, or an error, if applicable

- **setSQLPosition(value : String) : String**: it sets new squelch position value and returns the operation result in a callback

- **getTransceiverStatus() : String** : it returns in a callback the current transceiver status ("transmission(tx)/reception(rx))

- **setTransceiverStatus(status : String) : String**: it sets new transceiver status ("tx/rx") value and returns the operation result in a callback

- **setMainBand() : String**: it sets the main band as the operative one and returns the operation result in a callback

- **setSubBand() : String**: it sets the sub band as the operative one and returns the operation result in a callback

- **exchangeBands() : String**: it exchanges main and sub bands and returns the

operation result in a callback

4.1.3 Front-end

On the client part the front-end is defined, which looks like a control panel with all the bars, gauges, bottoms, etc that define the [transceiver](#) behavior.

Depending on what the user manipulates in the view, the controller part will make the proper [HTML](#) requests to the server in order to execute the above described [transceiver](#) functions. Once the client gets the [HTTP](#) response, the controller will update the view again.

The final front-end design can be seen in figure [4.2](#)

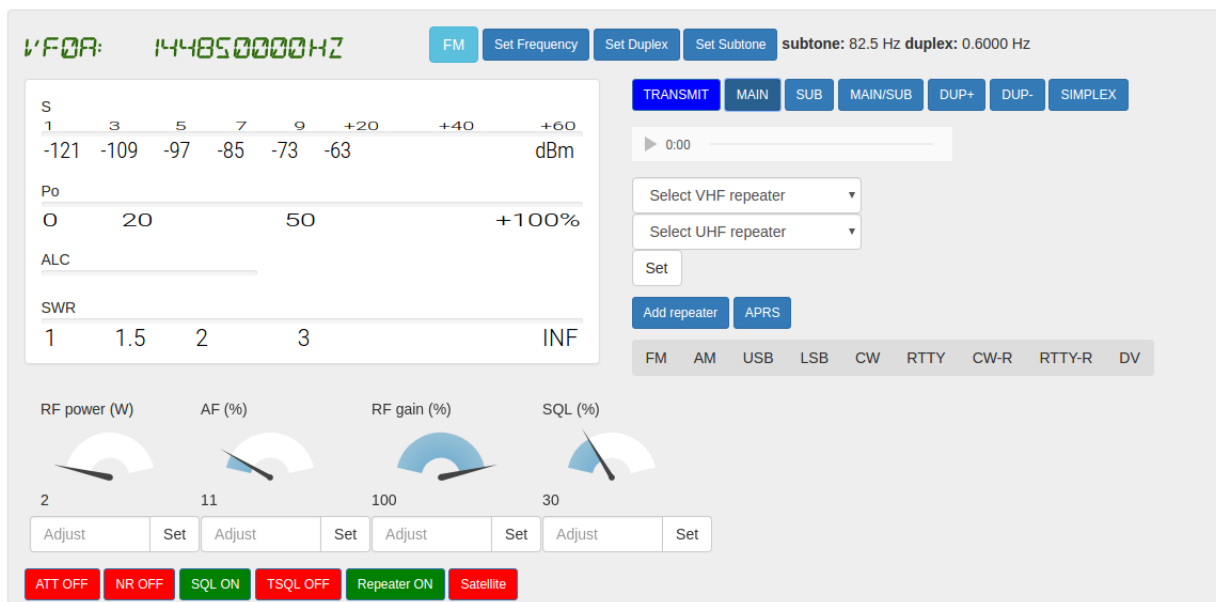


Figure 4.2 – *icom91100 transceiver final front-end*

Let us see in detail how the implementation works, taking the attenuator function as an example, that is, since the moment the user presses the attenuator button to activate, to the moment the view is updated showing that the operation has been performed correctly:

1. The user sets ON the attenuator button in the front end, by clicking the proper button (see button above in figure [4.2](#)).

2. The [AngularJS](#) controller handles the click button and perform the [HTTP](#) POST request to the server in order to set the attenuator ON in the [transceiver](#).

```

1
2      /**
3       * It switches on/off the transceiver's attenuator
4       */
5       scope.attenuatorButton = function() {
6
7           // Checking attenuator button status
8           var elem = document.getElementById("attenuator");
9
10          if (elem.innerHTML == "ATT OFF") {
11              elem.innerHTML = "ATT ON";
12              elem.style.background = "green"
13              scope.setAttenuator("on"); // Sending command
14          } else {
15              elem.innerHTML = "ATT OFF";
16              elem.style.background = "red"
17              scope.setAttenuator("off"); // Sending command
18          }
19      };
20
21
22
23      scope.setAttenuator = function(status) {
24          return $http({
25              method: 'POST',
26              url: `radiostation/attenuator`,
27              data: {option: status}
28          });
29      };

```

3. The server gets the [HTTP](#) POST request under the route `/radiostation/attenuator` and calls the function that is in the class “`transceiver.js`”, which is called `setAttenuator`.

```

1 app.post('/radiostation/attenuator', function(req, res) {
2
3     radioStation.setAttenuator(req.body.option, function(data) {
4         res.json(data);
5     });
6 });

```

4. In the function `setAttenuator`, the command that sets the attenuator ON (“FEFE7CE01120FD”) is sent to the [transceiver](#) using the serial port.

```

1
2     function setAttenuator(status, cb) {
3
4         var option;
5         if (status == "on") {
6             option = "20";
7         } else if (status == "off") {
8             option = "00";
9         }
10        serial.write(Buffer("FEFE7CE011" + option + "FD", "hex"), function (err) {
11            if (err) {
12                console.log("Error writing to ICOM 9100", "error")
13                cb({
14                    error: "Serial Write"
15                });
16            } else {
17                parameters.att = status
18                cb({
19                    status: "Done"
20                });
21            }
22        })
23    }

```

5. The transceiver sets the attenuator ON and the server sends back an [HTTP](#) response with an OK (“Done”) status, or an “Error” in case some error happens.

A flowchart of the process can be seen in figure [4.3](#).

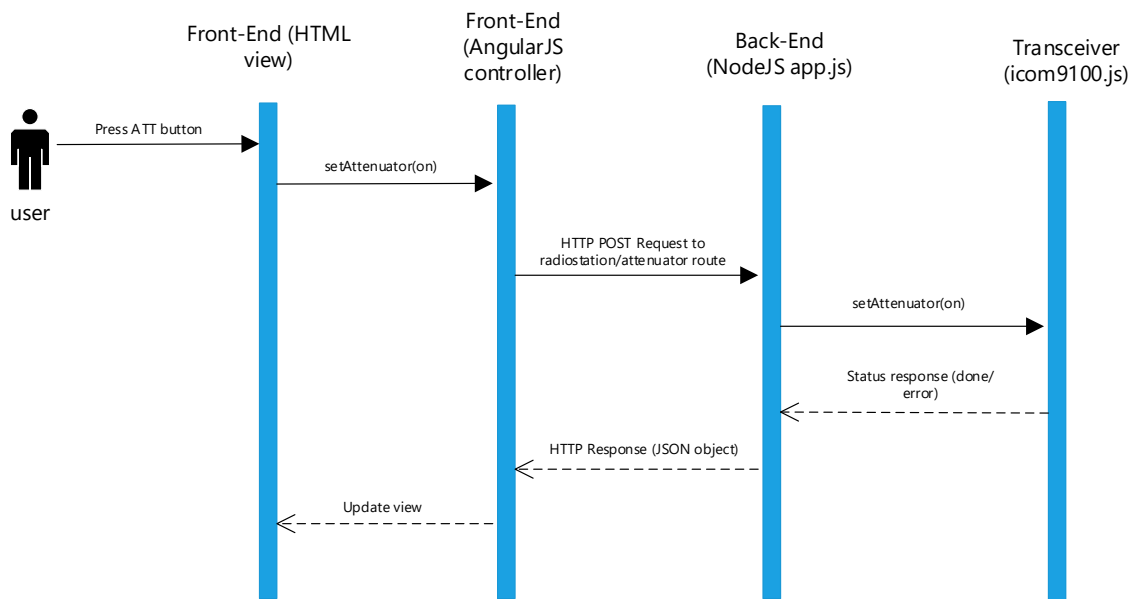


Figure 4.3 – *icom9100 transceiver telecontrol process - example*

4

4.1.4 Functions for graphical elements

As it was said during the analysis in section 3.3.2, the graphical elements of the [transceiver](#) front-end are designed with [Bootstrap](#) elements and the [gauge.js](#) library.

The functions that update the graphical elements are located in a module which is called `graphicalElement.js`.

Here the functions that set up the `gauge.js` gauges are located, besides the functions that update the gauges and bar values (see example below).

```

1 | function setAFGauge() {
2 |     var opts = {
3 |         lines: 12,           // The number of lines to draw
4 |         angle: 0.07,        // The length of each line
5 |         lineWidth: 0.44,    // The line thickness
6 |         fontSize: 32,
7 |         pointer: {
8 |             length: 0.9,    // The radius of the inner circle
9 |             strokeWidth: 0.046, // The rotation offset
10 |             color: '#424242' // Fill color
11 |         },
12 |         limitMax: 'false',  // If true, the pointer will not go past the end of the
13 |         gauge
14 |         gradients: ['#6FADCF', '#B6D0DE'],
15 |         strokeColor: 'FFFFFF', // to see which ones work best for you
16 |         generateGradient: true
17 |     };
18 |
19 |     var target = document.getElementById('af_gauge'); // your canvas element
20 |     gaugeAF = new Gauge(target).setOptions(opts); // create sexy gauge!
21 |     gaugeAF.setMinValue(0); // Prefer setter over gauge.minValue = 0
22 |     gaugeAF.animationSpeed = 32; // set animation speed (32 is default value)
23 |     gaugeAF.maxValue = 100;
24 |     gaugeAF.setTextField(document.getElementById('tfDisplayAF'));
25 |     gaugeAF.set(1);
26 |
27 | }
28 |
29 |
  
```

```
30 | function updateAFGauge(value) {  
31 |     gaugeAF.set(parseInt(value)); // set current value  
32 |     gaugeAF.setTextField(document.getElementById('tfDisplayAF'));  
33 | }
```

A flowchart that describes the process of updating the graphical elements is shown in figure 4.4

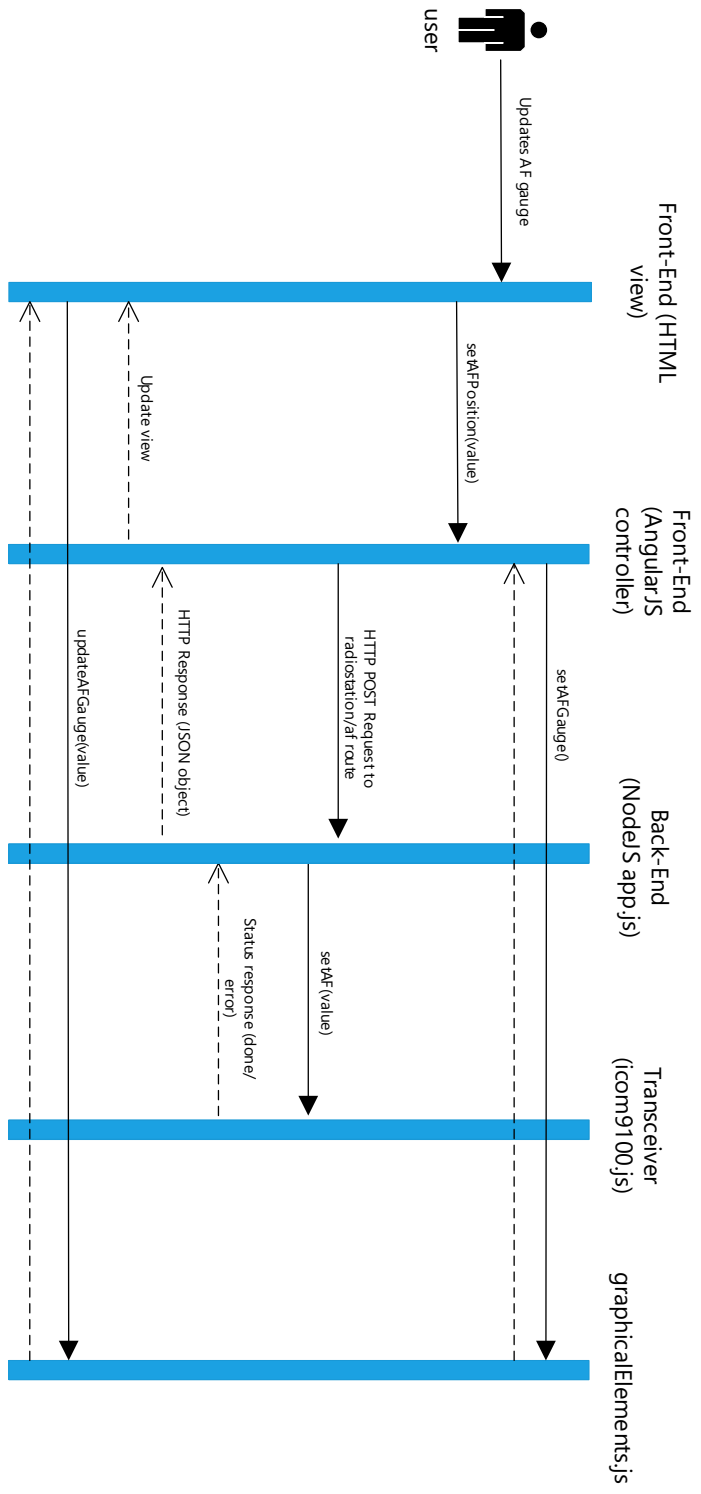


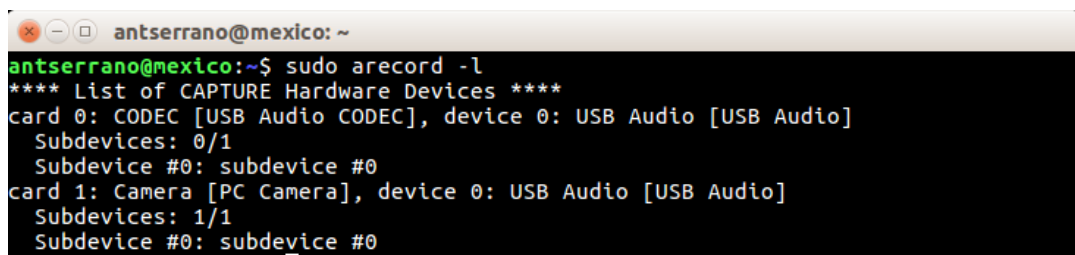
Figure 4.4 – Transceiver’s graphical elements updating process

4.2 Transceiver's Audio

As it was explained in section 3.4, the [transceiver](#)'s audio will be streamed in-live making use of Darkice and Icecast2 applications.

First of all, as it has been said before, the [transceiver](#) is connected to the server via serial port, therefore, the server can access the [transceiver](#)'s audio card thanks to the [ALSA](#) sound card [API](#), a software framework that is part of the Linux kernel [2].

Executing the [ALSA](#) command “arecord -l” on the server will display the audio devices available for capturing, including the [transceiver](#) audio card among those.



```
ant serrano@mexico: ~
ant serrano@mexico:~$ sudo arecord -l
**** List of CAPTURE Hardware Devices ****
card 0: CODEC [USB Audio CODEC], device 0: USB Audio [USB Audio]
  Subdevices: 0/1
  Subdevice #0: subdevice #0
card 1: Camera [PC Camera], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

Figure 4.5 – Audio devices available with arecord command

Thus, the audio that is coming out of the [transceiver](#) could be easily accessed by selecting the proper audio device from an external application (in this case Darkice). Specifically, as it is shown in 4.5, the device that corresponds with the [transceiver](#) would be “hw:0,0”, which corresponds with card 0, device 0.

4.2.1 Back-end

For this, first of all, the Icecast2 server needs to be running as a daemon in some port of the server.

To start it up, it is necessary to execute:

```
$/etc/init.d/icecast2 start
```

This will start the streaming server on port 8000 (it is possible to modify it).

Icecast2 has a configuration file where some parameters are adjustable, such as the admin user with its password, port where the application runs or the max. number of accepted clients at the same time.

This file is in `/etc/icecast2/icecast2.xml`.

Once Icecast2 is running, DarkIce will capture the audio output of the [transceiver](#) and afterward will send this to the server in order to be streamed in live.

DarkIce is run with a specified configuration file, which is the following in this case:

```

1 | # this section describes general aspects of the live streaming session
2 | [general]
3 | duration = 0
4 | bufferSecs = 1
5 | reconnect = yes
6 |
7 | [input]
8 | device = dsnoop:0,0 # VERY IMPORTANT to use dsnoop plugin in order to allow
9 |                   # two programs to use the same audio device
10 |
11 | sampleRate = 48000
12 | bitsPerSample = 16
13 | channel = 2
14 |
15 | [icecast2 -0]
16 | bitrateMode = abr
17 | format = vorbis
18 | bitrate = 256
19 | server = 0.0.0.0
20 | port = 8000 #icecast2 server port
21 | password = admin #icecast2 server password (you can change it in /etc/icecast2/icecast2.xml)
22 | mountPoint = streaming #audio location, in this case: server:8000/streaming
23 | name = Icom9100 streaming
24 | description = Icom91000 transceiver audio output
25 | public = yes
26 | localDumpFile = /utils/dump.ogg

```

It is important to notice that the audio output of the [transceiver](#) will be used from two sources, [Direwolf](#) and [Darkice](#), which is not possible since the following error is raised: “the selected audio device is busy”. To solve this, the [dsnoop ALSA](#) plugin is used, which allows several applications to record from the same device [22].

Therefore, in the configuration file (see above), `dsnoop:0,0` is chosen as input device, which is the [transceiver](#) audio card.

Hence, when the [Dashboard](#) starts up, [DarkIce](#) will be executed with this configuration file (using the [spawn NodeJS](#) module), which will start automatically the audio streaming under `granasat2.ugr:8000/streaming`

```

1 | const audioStreaming = spawn('darkice', ['-c', config.audio_file_configuration]);

```

4.2.2 Front-end

Lastly, from the client ([Dashboard](#)), it is trivial to connect to this mount-point with the [HTML](#) audio tag, so the user can listen to this live audio streaming.

Notice that the final URL is `granasat2.ugr.es/audio` instead of `granasat2.ugr:8000/streaming`. This is because the latter is an insecure URL (not [HTTP](#)), therefore, making use of [nginx](#) all the insecure URLs are redirected to secure ones (see [appendix E](#)).

```

1 | <audio autoplay controls>
2 |   <source src="https://granasat2.ugr.es/audio" />
3 | </audio>

```

A final scheme with the design can be seen in [4.6](#)

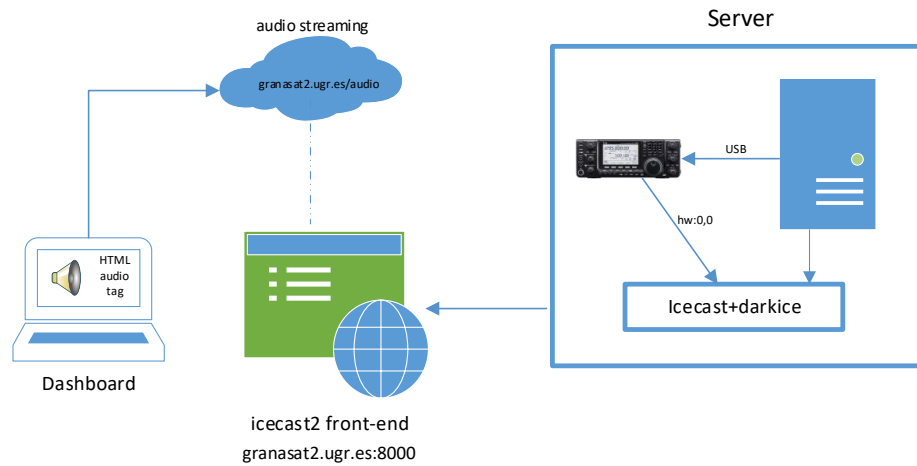


Figure 4.6 – *Transceivers audio streaming design*

4.3 TNC

As it was explained in 3.5, the TNC will be implemented in the system making use of [Direwolf](#), a virtual TNC.

[Direwolf](#) is a software "soundcard" AX.25 packet modem/TNC, that is, it is basically a virtual TNC. [Direwolf](#) configuration file allows to define the audio input/output that the program will take. In this case, once again thanks to the [ALSA API](#) the [transceiver](#) input and output audio cards will be available, which will be defined as input/output of [Direwolf](#). This will simulate a physical connection between the [transceiver](#) and the TNC ([Direwolf](#)), since [Direwolf](#) is connected to the [transceiver](#) through the server audio interface.

Hence, with the inclusion of [Direwolf](#), the [Ground Station](#) structure would be as follows:

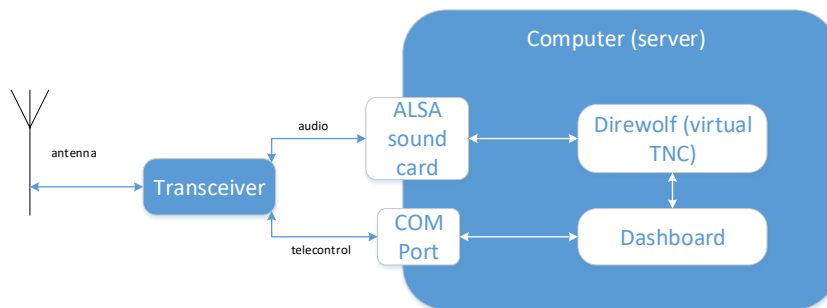


Figure 4.7 – *GranaSAT Ground Station final scheme*

As described in section 3.5, in order to communicate with a TNC it is needed a serial port. Since [Direwolf](#) is a virtual TNC, there is no physical serial port that allow a client [KISS](#) application to communicate with the TNC ([Direwolf](#) in this case).

This is solved by [Direwolf](#) with the option “-p”, which creates a virtual serial port under the path “/tmp/kisstnc”, allowing the chosen [KISS](#) [9] library to communicate with [Direwolf](#) from the server.

[Direwolf](#) is executed in a Linux terminal, therefore, this will be executed it on the server when the [Dashboard](#) application is started up. When executed in the terminal, [Direwolf](#) is not an interactive application, that is, it only shows the received and decoded [AX25](#) packets, as well as the packets sent by the user.

An example of [Direwolf](#) normal output can be seen in figure 4.8, where the blue messages are the decoded [AX25](#) packets.

```

antonio@antonio-Lenovo-G580:~$ sudo direwolf
Dire Wolf version 1.2
Audio device for both receive and transmit: plughw:0,0 (channel 0)
Channel 0: 1200 baud, AFSK 1200 & 2200 Hz, E+, 44100 sample rate.
Note: PTT not configured for channel 0. (Ignore this if using VOX.)
Ready to accept AGW client application 0 on port 8000 ...
Ready to accept KISS client application on port 8001 ...
Use -p command line option to enable KISS pseudo terminal.

Digipeater N6EX-1 audio level = 7(1/1) [NONE] |||____
[0.1] AE6MP>SS5PPQ-2,N6EX-1*:.](n->/"4W]
MIC-E, normal car (side view), Unknown manufacturer, En Route
N 33 50.0100, W 118 05.1200, 24 MPH, course 334, alt 210 ft

Digipeater N6EX-5 audio level = 2(0/0) [NONE] ||:____
[0.1] AE6MP>SS5PPQ-1,N6EX-5*:.](n->/"4W]
MIC-E, normal car (side view), Unknown manufacturer, En Route
N 33 50.0100, W 118 05.1200, 24 MPH, course 334, alt 210 ft

Digipeater N6EX-1 audio level = 0(0/0) [NONE] ||_____
[0.0] KD6UZM-15>S3UWTS,WB6JAR-10,N6EX-1*:-<0x1d>l <0x1c>v\":r]
MIC-E, OVERLAYED Van, Unknown manufacturer, In Service
N 33 57.4300, W 117 13.0100, 0 MPH, alt 2090 ft
^C

```

Figure 4.8 – *Direwolf* terminal output

4.3.1 Back-end

Direwolf needs to be executed every time the *Dashboard* starts up, so that the user can send/receive *AX25* packets from this.

Direwolf can be run with a specific configuration file, which in this case, will indicate the source audio card where *Direwolf* will take the audio from and where it will send the audio through. (similar to the configuration file for *Darkice* described in 4.2).

The audio input/output is defined in the configuration file as follows. (Notice once again the use of *dsnoop*, as it was already explained in 4.2).

```
1|ADEVICE dsnoop:1,0 plughw:1,0
```

Hence, *Direwolf* will be run when the *Dashboard* application starts up, specifically by making use of the *spawn* *NodeJS* module (in the same way *Darkice* was executed in the previous section).

At the same time, as it was already explained, *Direwolf* will be running in background and its output needs to be catch on the server and be sent to the client via *websocket*. The following code makes this:

```

1
2 var privateKey = fs.readFileSync('/certificados/granasat2_ugr_es.key', 'utf8');
3 var certificate = fs.readFileSync('/certificados/bundle.crt', 'utf8');
4 var credentials = { key: privateKey, cert: certificate };
5
6 //pass in your credentials to create an https server
7 var httpsServer = https.createServer(credentials);
8 httpsServer.listen(8003);
9
10 var WebSocketServer = require('ws').Server;
11 var wss = new WebSocketServer({
12   server: httpsServer
13 });
14
15 // Executing direwolf
16 const direwolf = spawn('direwolf', ['-n','2','-p', '-t', '0', '-c', config.
17   direwolf_configuration]);
18 wss.on('connection', function connection(ws) {

```

```

19
20 var connected = true;
21 ws.on('message', function incoming(message) {
22   console.log('received: %s', message);
23 });
24
25 ws.on('close', function(connection) {
26   ws.close();
27   connected = false;
28 });
29
30 direwolf.stdout.on('data', function(data) {
31
32   // Saving content into log file
33   fs.appendFile(config.aprs_log_file, data.toString(), (err) => {
34     if (err) {
35       throw err;
36     }
37   });
38
39   try {
40     if (connected) {
41       ws.send(data.toString());
42     }
43   }
44   catch (err) {
45     console.log(err.message)
46   }
47 });
48 });

```

When sending commands, the function that receives the desired text to be packed into [AX25](#) and communicates with [Direwolf](#) is the following (notice callsign of the [GranaSAT Ground Station](#) and a supposed callsign for a [Cubesat](#)).

```

1 function send_string(str) {
2
3   const packet = new AX25.Packet();
4   packet.type = AX25.Masks.control.frame_types.u_frame.subtypes.ui;
5   packet.source = { callsign : 'EB7DZP', ssid : 0 };
6   packet.destination = { callsign : 'GRNSAT', ssid : 0 };
7   packet.payload = Buffer.from(str, 'ascii');
8   tnc.send_data(packet.assemble(), () => log('Sent AX25 frame:' + str));
9
10 }

```

When the user writes the desired text to be packed into [AX25](#) packets, the client makes an [HTTP POST](#) request to the server, concretely to the [NodeJS](#) route which is called “/radiostation/send_packet”.

This function sets the [transceiver](#) into transmission mode before communicating with [Direwolf](#). After this, the command is sent by using the function described above.

Once the command has been sent, the [transceiver](#) is set back to reception mode, in order to not to saturate this.

The described function is as follows:

```

1 app.post('/radiostation/send_packet', isMember, function(req, res) {
2   radioStation.setTransceiverStatus("tx", function(data) {
3     if (data.status === "Done") {
4       process.on('SIGTERM', tnc.close);
5       tnc.on('error', console.log);
6       tnc.open(() => {
7         log('TNC opened');
8         send_string(req.body.command);
9         setTimeout(function() {
10          // Setting transceiver back to RX
11          radioStation.setTransceiverStatus("rx", function(data) {
12            res.json(data)
13          });
14          }, ((1/1200)*8*req.body.command.length) + config.delay_error);
15        });
16      });
17   } // if
18   else { res.json({ error: "Error" });
19   });
20 });

```

4.3.2 Front-end

The client, in order to see the [Direwolf](#) output, will need to connect to the websocket described above and display its content. In this case, it will be displayed on the [Dashboard](#) terminal, where messages can be displayed by making use of its functions, in this case, with the function “logHTML”.

```

1 // websocket for direwolf output
2 const ws = new WebSocket('wss://granat2.ugr.es:8003');
3
4 // Reading web socket content
5 ws.binaryType = 'arraybuffer';
6 ws.onmessage = function (e) {
7
8     // Showing AX25 message decoded
9     if (direwolfEnabled) {
10        con.logHTML(
11            "<p style='color:blue'>" + e.data + "</p>"
12        );
13    }
14 }

```

As a result of the described design, there are two processes that come out of the [Direwolf](#) implementation:

On the one hand, [Direwolf](#) automatically decodes the received signals and displays it on the [Dashboard](#), without any user interaction (see figure 4.9).

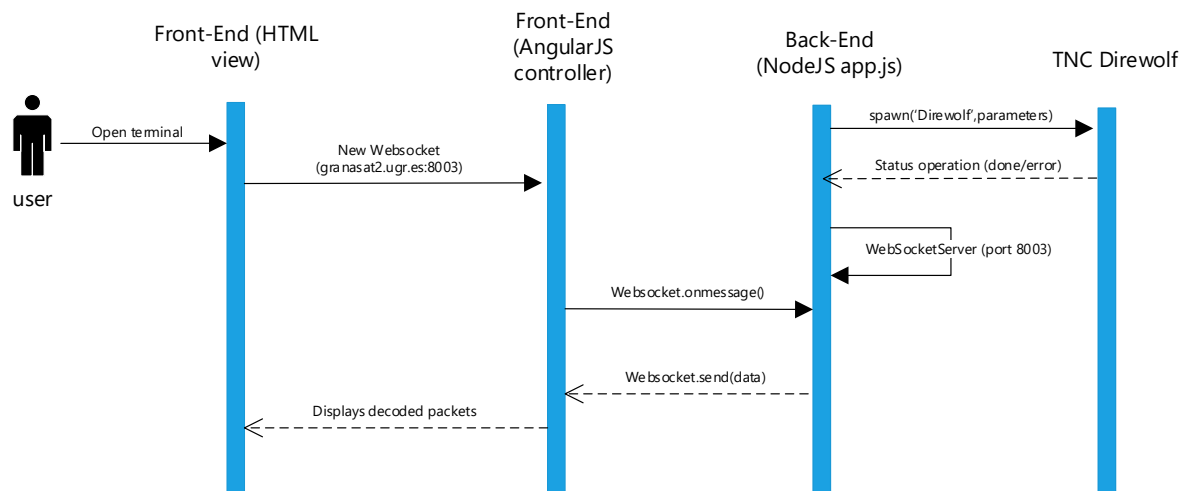


Figure 4.9 – Decoding [AX25](#) with [Direwolf](#) flowchart

On the other hand, the user interaction is needed in order to send [AX25](#) packets making use of the previously chosen library [9], which allows us to communicate with [Direwolf](#) see figure 4.10).

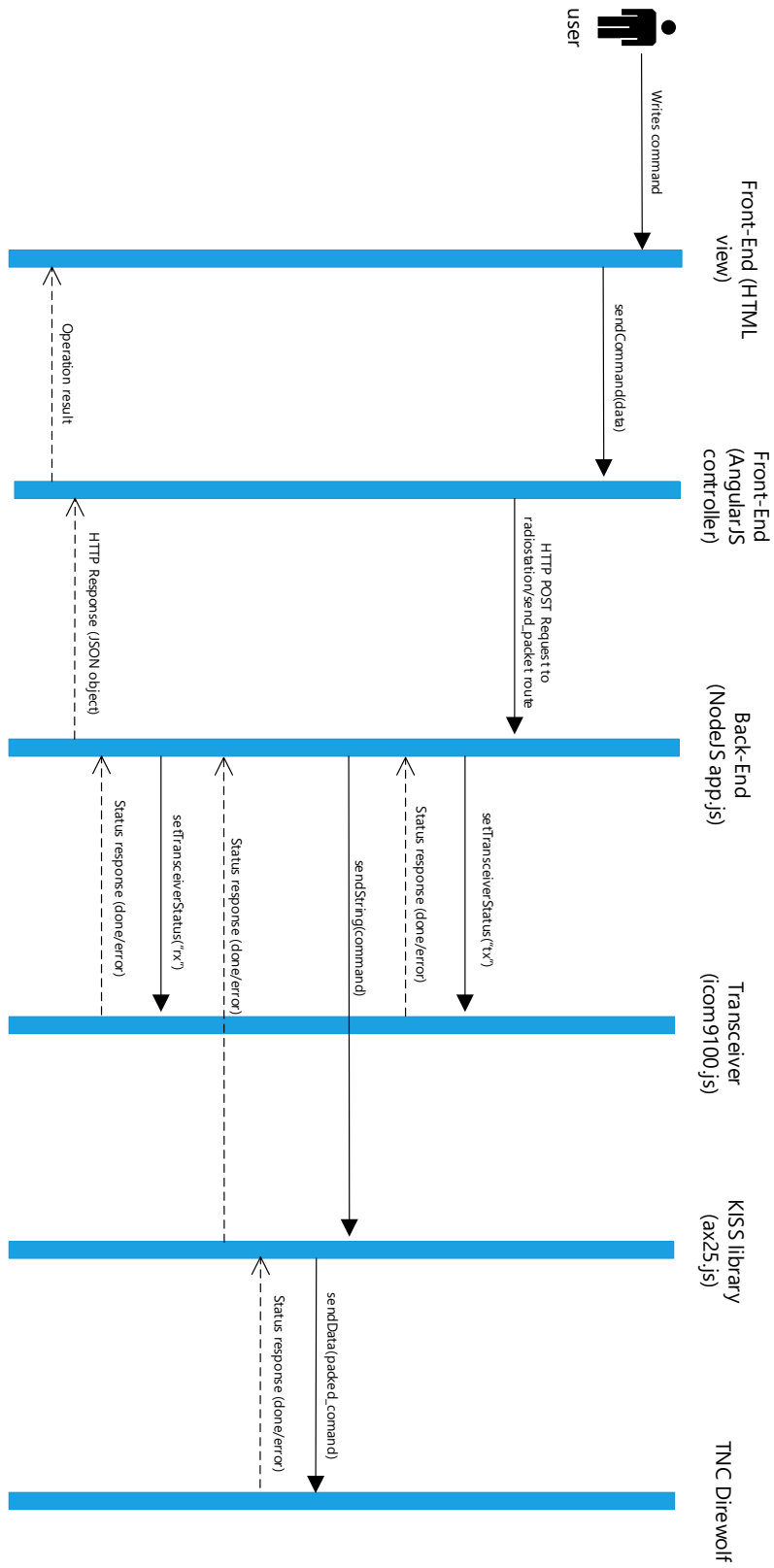
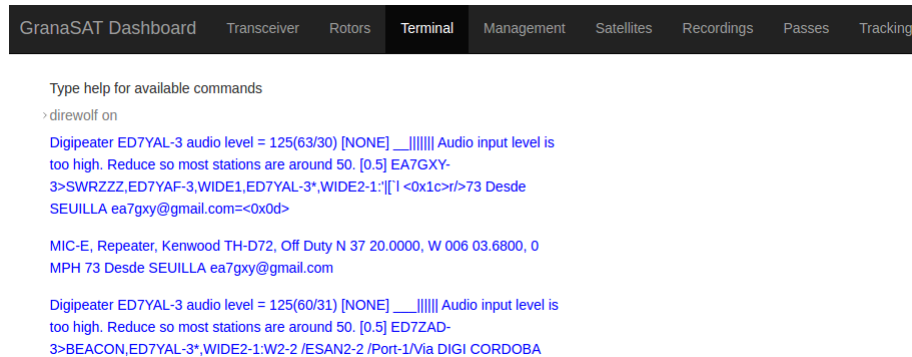


Figure 4.10 – Sending AX25 packets with Direwolf flowchart

Hence, as a result of the described processes, in the terminal are displayed the packets that are being decoded by [Direwolf](#) (see figure 4.11), as well as being able to pack commands into [AX25](#) packets and send it (see figure 4.12).



```

GranaSAT Dashboard  Transceiver  Rotors  Terminal  Management  Satellites  Recordings  Passes  Tracking

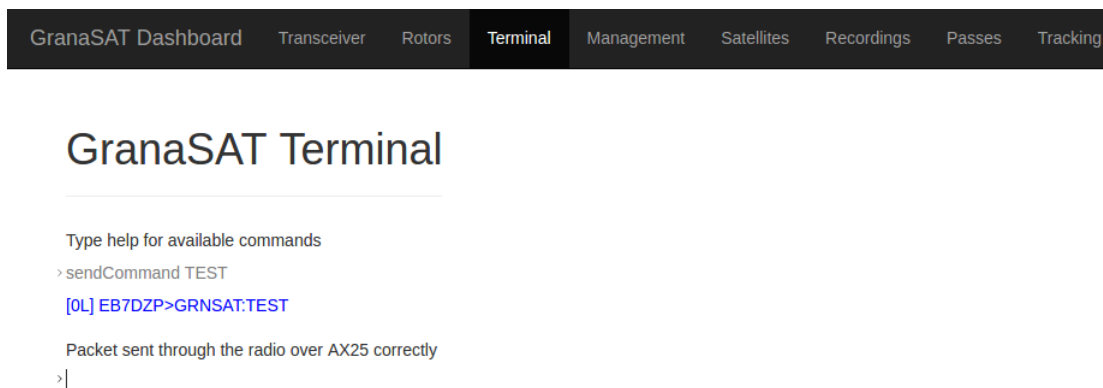
Type help for available commands
> direwolf on
Digipeater ED7YAL-3 audio level = 125(63/30) [NONE] |_|||||| Audio input level is
too high. Reduce so most stations are around 50. [0.5] EA7GXY-
3>SWRZZZ,ED7YAF-3,WIDE1,ED7YAL-3*,WIDE2-1:[*] <0x1c>rt/73 Desde
SEUILLA ea7gxy@gmail.com=<0x0d>

MIC-E, Repeater, Kenwood TH-D72, Off Duty N 37 20.0000, W 006 03.6800, 0
MPH 73 Desde SEUILLA ea7gxy@gmail.com

Digipeater ED7YAL-3 audio level = 125(60/31) [NONE] |_|||||| Audio input level is
too high. Reduce so most stations are around 50. [0.5] ED7ZAD-
3>BEACON,ED7YAL-3*,WIDE2-1:W2-2 /ESAN2-2 /Port-1/Via DIGI CORDOBA

```

Figure 4.11 – Decoding packets (concretely [AX25/APRS](#)) and displaying in [Dashboard](#)



```

GranaSAT Dashboard  Transceiver  Rotors  Terminal  Management  Satellites  Recordings  Passes  Tracking

GranaSAT Terminal

Type help for available commands
> sendCommand TEST
[0L] EB7DZP>GRNSAT:TEST

Packet sent through the radio over AX25 correctly
>|

```

Figure 4.12 – Sending [AX25](#) packets on the [Dashboard](#)

4.4 Yaesu Rotors Telecontrol

4.4.1 Front-end

As it was seen in section 3.6, in this case, since the back-end was already developed, only the front-end for the Yaesu telecontrol will be modified with customized gauges that will display **azimuth** and **elevation** of the antenna by making use of the D3.js library.

These **azimuth** and **elevation** values are retrieved by an **AngularJS** directive that performs the proper **HTTP** requests to the server.

Therefore, it is only necessary to display these provided values on the gauges.

To do this, a function is defined within the **AngularJS** directive. This function updates the gauge every second, showing the current values of **azimuth** and **elevation**.

```

1   scope.updateGauges = function () {
2
3       // Retrieving values of azimuth and elevation
4       var value_elevation = scope.yaesuPosition.ele;
5       var value_azimuth = scope.yaesuPosition.azi;
6
7       // In case values are lower than 1 or null (only happens in Local), we set it to
8       0
9       if (value_elevation < 0 || value_elevation == null) {value_elevation = 0}
10      if (value_azimuth < 0 || value_azimuth == null) {value_azimuth = 0}
11
12      // Redrawing gauges
13      gauges["elevation"].redraw(value_elevation); // Gauge for elevation
14      gauges["azimuth"].redraw(value_azimuth); // Gauge for azimuth
15  };

```

The result can be seen in figure 4.13.



Yaesu 5500 Rotors

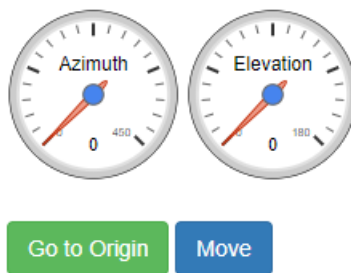


Figure 4.13 – Yaesu Rotors front-end

4.5 Satellite tracking

As described in section 3.7, the satellite tracking will be implemented making use of [Leaflet](#) on the client side, and some libraries for satellites orbit prediction on the server, in order to get all the satellite information to be displayed in the maps provided by [Leaflet](#).

4.5.1 Back-end

Here the [NodeJS](#) routes are defined so that the server handles the [HTTP](#) requests from the client. Specifically, these requests will be made in order to send the satellites data back to the client, as a [JSON](#) object, which will contain information about the satellites following the [JSON](#) format, e.g. longitude: [156.13248,-20.14638], latitude: [175.21547,24.21348], footprint: 4500, etc

As described in section 3.7, satellites data will be retrieved from different libraries (two [Javascript](#) libraries and one [Python](#) library), therefore, two routes will be defined in the `app.js` file.

These routes are `getSatellitesData` and `getSatellitesOrbit` and are briefly described below:

- `getSatellitesData` : This function will call a [Javascript](#) module called “Propagator”, which gets name, tle1, tle2 of the satellite and [Ground Station](#) coordinates in order to make use of the satellite libraries described in section 3.7.

```

1 app.get('/getSatellitesData', function (req,res) {
2
3   // Getting satellite's name from http request
4   var satellite_name = req.query.sat_name;
5   var tle1 = req.query.tle1;
6   var tle2 = req.query.tle2;
7
8   // Getting satellites data
9   new Propagator(tle1, tle2, satellite_name, config.ground_station_lng, config.
    ground_station_lat, config.ground_station_alt).then(function (p) {
10      res.json(p.getStatusNow()); // sending response
11    });
12 });

```

This “Propagator” module has a function `getStatusNow`, which will return the following [JSON](#) object with all the satellite’s information:

```

1   return {
2     azi: lookAngles.azimuth * satellite.constants.rad2deg,
3     ele: lookAngles.elevation * satellite.constants.rad2deg,
4     dopplerFactor: dopplerFactor,
5     height : positionGd.height,
6     latitude : satellite.degreesLat(positionGd.latitude),
7     longitude : satellite.degreesLong(positionGd.longitude),
8     footprint : footprint,
9     light: light,
10    velocity : Math.sqrt(Math.pow(velocityEci.x,2) +
11      Math.pow(velocityEci.y,2) + Math.pow(velocityEci.z,2))
12  }

```


- **getSatellitesOrbit**: this function will use the `child_process` defined in section 3.7 and will execute the [Python](#) script that gets the satellite's orbit prediction.

This script calculates N interval times of past and future and then, by using the library `PyEphem`, computes the satellites orbit taking into account the previous calculated times. The main part of the script looks like:

```

1  #-----
2  # CALCULATING SATELLITE ORBIT
3  #-----
4  for date in times:
5      sat = ephem.readtle(name, 11, 12)
6      sat.compute(date) # calculate
7      c = (np.rad2deg(sat.sublat), np.rad2deg(sat.sublong)) # get coordinates
8      data["coordinates"].append(c) # save

```

Lastly, this information is saved in a [JSON](#) object, which is returned as a response by the server route:

```

1 app.get('/getSatellitesOrbit', function(req, res) {
2
3     // Getting satellite's name and TLE
4     var satellite_name = req.query.sat_name;
5     var tle1 = req.query.tle1;
6     var tle2 = req.query.tle2;
7
8     // Executing script
9     var path_satellites= spawn('python3', ['sat_library/getSatOrbit.py', satellite_name,
10     tle1, tle2])
11
12     path_satellites.on('close', function (code) {
13         if(code === 0){
14
15             // Reading file created by the script above
16             var conts = fs.readFileSync("./sat_library/data.json");
17
18             // Definition to the JSON type
19             var jsonCont = JSON.parse(conts);
20
21             res.json(jsonCont); // sending response
22         }
23         else{
24             res.json({
25                 error: "Error while executing python"
26             });
27         }
28     });
29 });
30
31 });

```

4.5.2 Front-end

Regarding the client side, there are several functions that control the map display:

- **followSatellite**: this function calls the function that moves the antenna, providing [azimuth](#) and [elevation](#) that is available in the satellite's information. This function was already programmed, therefore it is not going to be explained in detail.
- **updateTLE**: this function updates the satellites TLE's, making an [HTTP](#) request to the server in order to execute a script that updates the TLE in the data base, downloading them from <https://celestrak.com>. This function was already programmed on the server, therefore, it is not going to be explained in detail.
- **setUpMap**: this function sets up the [Leaflet](#) map, setting the proper layer from [Mapbox](#), adding a marker that points to the [Ground Station](#) and setting the terminator layer.

```

1 // Map elements
2
3 var mymap = null; // map
4 var marker = null; // satellite marker
5 var path = []; // satellite orbit
6 var foot = []; // satellite footprint
7 var terminator = null; // terminator
8 var myIcon = null; // satellite icon
9
10 function setUpMap() {
11
12     // Setting up map
13     mymap = L.map('mapid', {
14
15         center: [0,0],
16         minZoom: 0,
17         maxBounds : bounds,
18         worldCopyJump : false,
19         noWrap:true,
20         continuousWorld: false,
21     }).setView([0, 0], 1);
22
23     // Adding Groundstation marker
24     L.marker ([37.179640,-3.6095], { title : "Groundstation"}).addTo(mymap);
25
26     // Adding map layer (satellite layer)
27     L.tileLayer('https://api.tiles.mapbox.com/v4/{id}/{z}/{x}/{y}.png?access_token=pk.eyJ1Ijo1YW50c2VycmFubyIsImEiOiJjamU4ZXF6bm0wYTm5Mn1wZTF0NWVhbnhbWprIn0.iscZVwbCjSmzZD1GDV6zYg', {
28         maxZoom: 8,
29         attribution: 'Map data &copy; <a href="http://openstreetmap.org">OpenStreetMap </a> contributors, ' +
30             '<a href="http://creativecommons.org/licenses/by-sa/2.0/">CC-BY-SA</a>, ' +
31             'Imagery <a href="http://mapbox.com">Mapbox</a>',
32         id: 'mapbox.satellite',
33     }).addTo(mymap);
34
35     // Adding terminator
36     terminator = L.terminator().addTo(mymap)
37 }

```

- **updateMap**: this function updates the map with the satellites data gotten from the server. To do this, [Leaflet](#) functions are used to display the satellite's orbit, current position and footprint over the map, besides updating the terminator.

```

1 function updateMap(coord, footprint) {
2
3     var coordinates = coord.data["coordinates"];
4     var current_position = coord.data["now"];
5
6     // Removing what it was before
7     if (marker != null) {
8         mymap.removeLayer(marker);
9     }

```

```

10
11     if (path != null) {
12         var total_path = path.length;
13         for (var i=0; i<total_path; i++) {
14             mymap.removeLayer(path[i]);
15         }
16     }
17
18     if (foot != null) {
19         mymap.removeLayer(foot);
20     }
21
22     // Adding satellite marker
23     marker = L.marker (current_position , {icon: myIcon}).addTo(mymap);
24
25     // Drawing footprint (radius needs to be in meters)
26     foot = L.circle(current_position , {radius: footprint*1000/2}).addTo(mymap);
27
28     // Drawing satellite orbit
29     var totalMarkers = coordinates.length;
30     for (var i = 0; i<totalMarkers; i++){
31         var datos = (coordinates[i])
32
33         var x = L.circle ([datos[0],datos[1]] , {radius: 0.1, color:"red"}).addTo(mymap);
34         path.push(x)
35     }
36
37     // Updating terminator
38     terminator.setLatLngs(terminator.getLatLngs());
39     terminator.redraw();
40 }

```

- **getSatellitesOrbit**: it performs the [HTML](#) request to the server in order to get the coordinates of the satellite's orbit. Once it gets it, it call the above defined updateMap function in order to redraw the orbit.

```

1     scope.getSatellitesOrbit = function (sat_name) {
2         return $http({
3             method: 'GET',
4             url: "/getSatellitesOrbit",
5             params: {sat_name : sat_name}
6         }).then(function (res){
7
8             if (!res.data.error) {
9                 updateMap(res, scope.footprint)
10            }
11        });
12    };
13

```

- **getSatellitesData** : similar to the previous function, but getting more satellite's data that will be displayed in a [Bootstrap](#) table.

```

1     scope.getSatellitesData = function (sat_name) {
2         return $http({
3             method: 'GET',
4             url: "/getSatellitesData",
5             params: {sat_name : sat_name}
6         }).then(function (res){
7             if (!res.data.error) {
8                 scope.elevation = res.data["ele"].toFixed(4);
9                 scope.altitude = res.data["height"].toFixed(4);
10                scope.azimuth = res.data["azi"].toFixed(4);
11                scope.longitude = res.data["longitude"].toFixed(4);
12                if (res.data["light"] == 1) {
13                    scope.light = "Yes"
14                } else {
15                    scope.light = "No"
16                }
17                scope.footprint = res.data["footprint"].toFixed(4);
18                scope.latitude = res.data["latitude"].toFixed(4);
19                if (scope.elevation > 0) {
20                    scope.over_groundstation = "Yes"
21                } else {
22                    scope.over_groundstation = "No"
23                }
24                scope.velocity = res.data["velocity"].toFixed(4);
25            }
26        });
27    };

```

The two latter functions will be called every two seconds, so the satellite information is updating all the time with the current available data.

A flowchart with the whole the process followed is shown in figure [4.14](#)

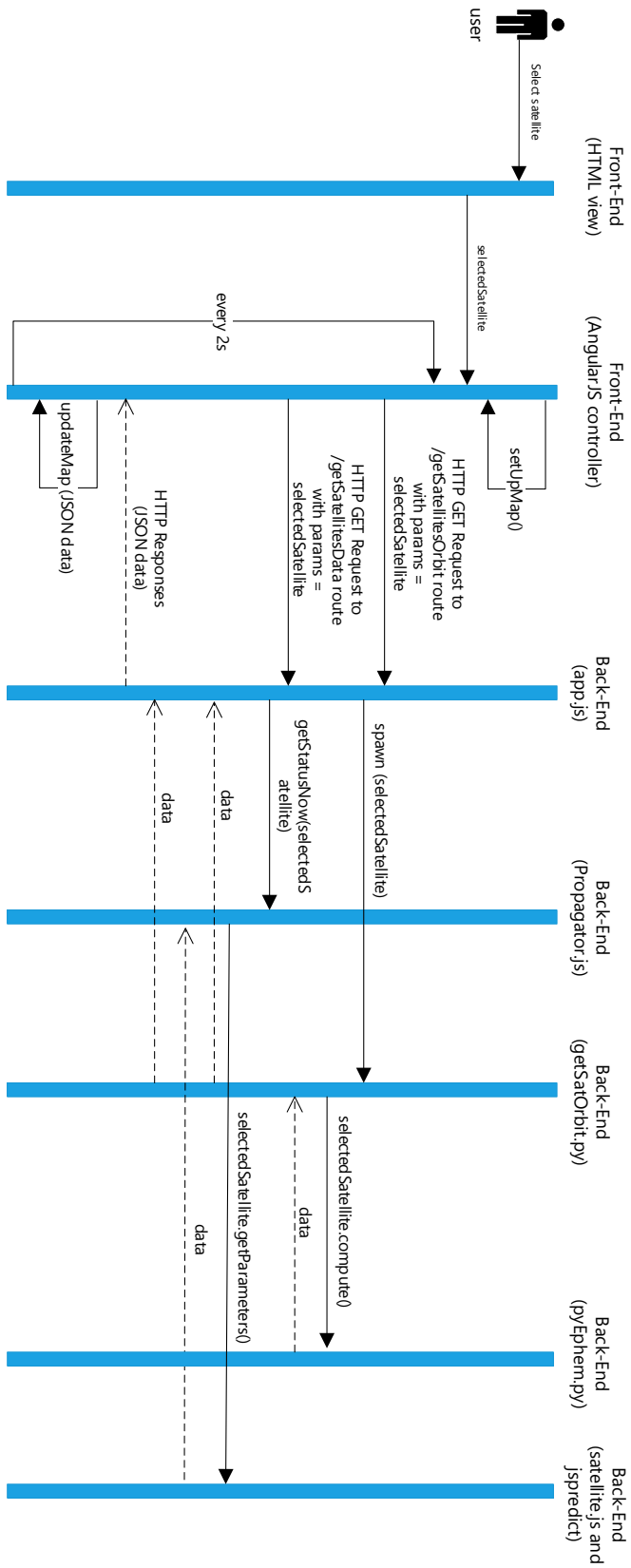


Figure 4.14 – Satellite tracking flowchart

The obtained results can be seen in figures 4.15 and 4.16 (notice responsive design in the latter)

Satellites tracking

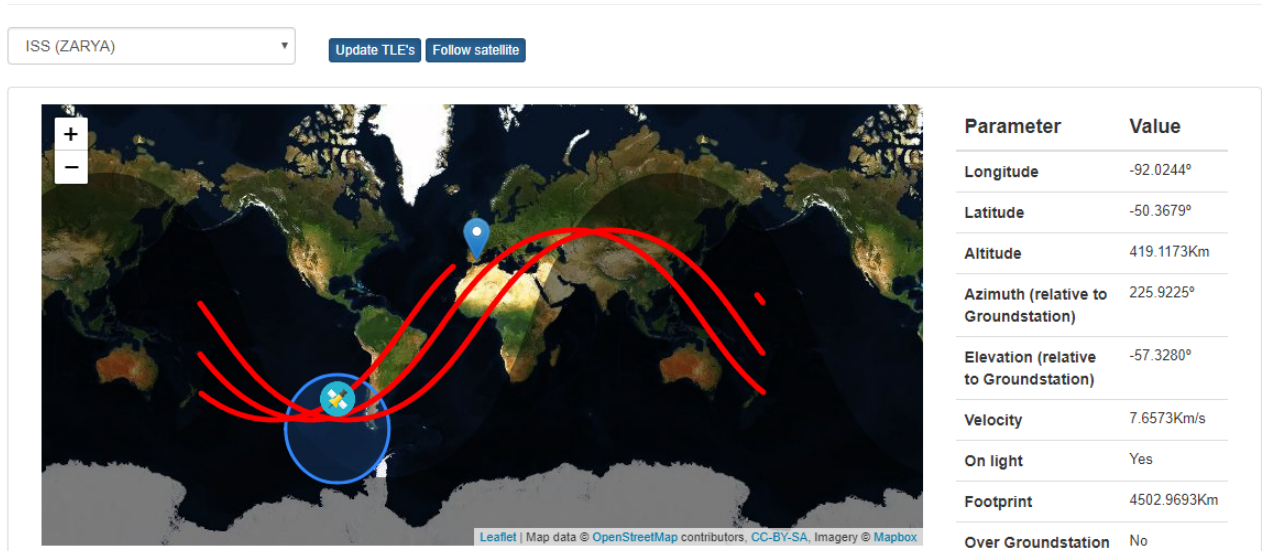


Figure 4.15 – Satellite tracking in laptop screen

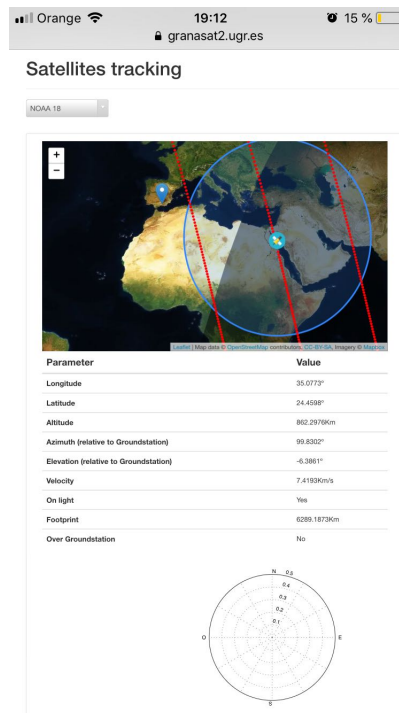


Figure 4.16 – Satellite tracking in mobile phone device

In addition, a polar graph that indicates the current satellites over [Ground Station](#) has been added (see figure 4.17), making use of the polar graph designed by [1]. This polar graph will be updating every second, therefore, in every moment, the user will know what are the satellites over the [Ground Station](#). When elevation is higher than 0, the satellite is over the [Ground Station](#), therefore, with the retrieved data, those satellites whose elevation is higher than 0 are filtered, displaying them on the polar graph.

```

1 |
2 | setInterval(scope.updatePolarGraph = function () {
3 |
4 |     if (scope.satellitesOverGround != null) {
5 |
6 |         for (var i = 0; i < scope.satellitesOverGround.length; i++) {
7 |
8 |             // Setting parameters
9 |             var name = scope.satellitesOverGround[i].name
10 |            var elevation = scope.satellitesOverGround[i].ele;
11 |            var azimuth = scope.satellitesOverGround[i].azi;
12 |            var colors = ["red", "blue", "purple", "black", "orange", "brown"]
13 |
14 |
15 |            var data = [];
16 |            data.push({ele: elevation, azi: azimuth})
17 |
18 |            // Drawing satellites as a circle if elevation > 0
19 |            svg.append("circle")
20 |                .attr("id", "satellite" + i)
21 |                .data(data)
22 |                .attr("r", 4)
23 |                .attr("fill", colors[i])
24 |                .attr("cx", function (d) {
25 |                    return Math.sin(d.azi * conv) * radius * ((-d.ele + 90) / 90)
26 |                })
27 |                .attr("cy", function (d) {
28 |                    return -(Math.cos(d.azi * conv) * radius * ((-d.ele + 90) / 90))
29 |                });
30 |
31 |            // Satellite name
32 |            svg.append("text")
33 |                .attr("id", "name" + i)
34 |                .attr("x", +radius - 340)
35 |                .attr("y", -radius - 10 + (15*i)); // 15*i allows to set more than
36 |                one legend correctly, one under another
37 |
38 |            svg.select("#name" + i)
39 |                .text(name);
40 |
41 |            // Satellite red circle in legend
42 |            svg.append("circle")
43 |                .attr("id", "sat_legend" + i)
44 |                .attr("r", 4)
45 |                .attr("fill", colors[i])
46 |                .attr("cx", +radius - 345)
47 |                .attr("cy", -radius - 15 + (15*i)); // 15*i allows to set more than
48 |                one legend correctly, one under another
49 |
50 |            // Removing for next iteration (so satellites don't stay when they
51 |            // are not anymore over ground)
52 |            d3.select("#satellite" + i).remove();
53 |            d3.select("#name" + i).remove();
54 |            d3.select("#sat_legend" + i).remove();
55 |         }
56 |     }, 1000)

```

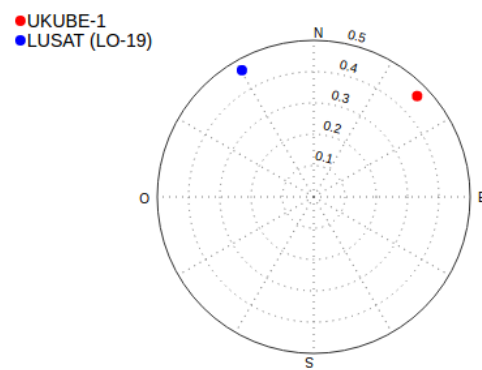


Figure 4.17 – Polar graph with satellites over *Ground Station*

4.6 Other improvements

In this section, some secondary aspects of the system have been implemented. They are not as important as the primary requirements but it is necessary to point them out as well.

4.6.1 Tooltips design

As it was described in section 3.8.1, the tooltips will be developed with Twitter Bootstrap.

For the tooltip's display there is no need of making HTTP requests to the server, rather these are stored on the client side.

All the tooltips will be defined in a single file in order to provide modularity, scalability and ease of modification to the system. This file is called `mainTooltips.js`.

The **Model-View-Controller** pattern will be followed, making use of **AngularJS**, that is, tooltips will be saved in a file (model) and the controller will be showing the different tooltips to the client (view).

A simple scheme with the logic of the implementation can be seen in figure 4.18.

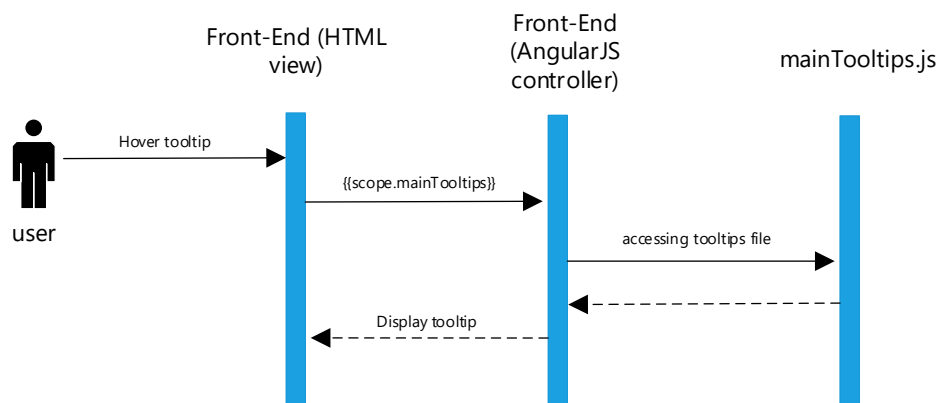


Figure 4.18 – *Tooltips design diagram*

4.6.1.1 Model

The model stores all the tooltip's texts in a file by using a simple Array. Since it is important to identify where the tooltip is going to be displayed, it is necessary to name them with recognizable names.

```

1 | var mainTooltips = {
2 |
3 |   mainMenu: {
4 |     rotorsTab: "<sub>Set</sub> elevation and azimuth ",
5 |     antennasTab: "Set the frequency of the radio",
6 |     terminalTab: "Same options via terminal",
7 |     passesTab: "Choose and schedule your passes",
8 |     managementTab: "Users management",
9 |     satellitesTab: "Available satellites",
10 |    recordingsTab: "Available recordings",
11 |    trackingSatellitesTab: "Satellite tracking",
12 |    rotorsYaesu: "Current elevation and azimuth ",
13 |    radioStation: "Current frequency",
14 |    scheduledPasses: "<sub>Cur</sub>rent scheduled passes"
15 |  },
16 |  // rest of tooltips

```

4.6.1.2 Controller

The controller (mainController.js) merely needs to retrieve the information provided by the model, in this case, this is made by adding a variable to the controller scope, which gets the Array object already described above.

```

1 | // With this we include all the tooltips from mainTooltips.js
2 | scope.mainTooltips = mainTooltips

```

4.6.1.3 View

The view makes use of the tooltip text provided from the controller by using `{{name_tooltip}}` ([AngularJS](#) syntax).

To include it within [HTML](#), the proper jQuery function of [Bootstrap](#) is used (using "data-toggle='tooltip'"), besides some CSS classes that that are defined (in case customization is wanted).

Some other parameters are adjustable (such as relative position of tooltip). For example:

```

1 | <h3 class="red" data-toggle="tooltip" data-placement="bottom"
2 |   title={{mainTooltips.mainMenu.rotorsYaesu}}>

```

4.6.2 Webcam streaming design

4.6.2.1 Back-end

As described in section 3.8.2, Motion will be the software that will be used to develop the streaming video part of the system. Specifically, MotionEye has been used [5], which is the GUI of the Motion daemon.

The installation of MotionEye can be seen in F .

Once MotionEye is installed and running on the server as a daemon, its web front-end can be accessed from the default port 8765 as it is shown in figure 4.19.

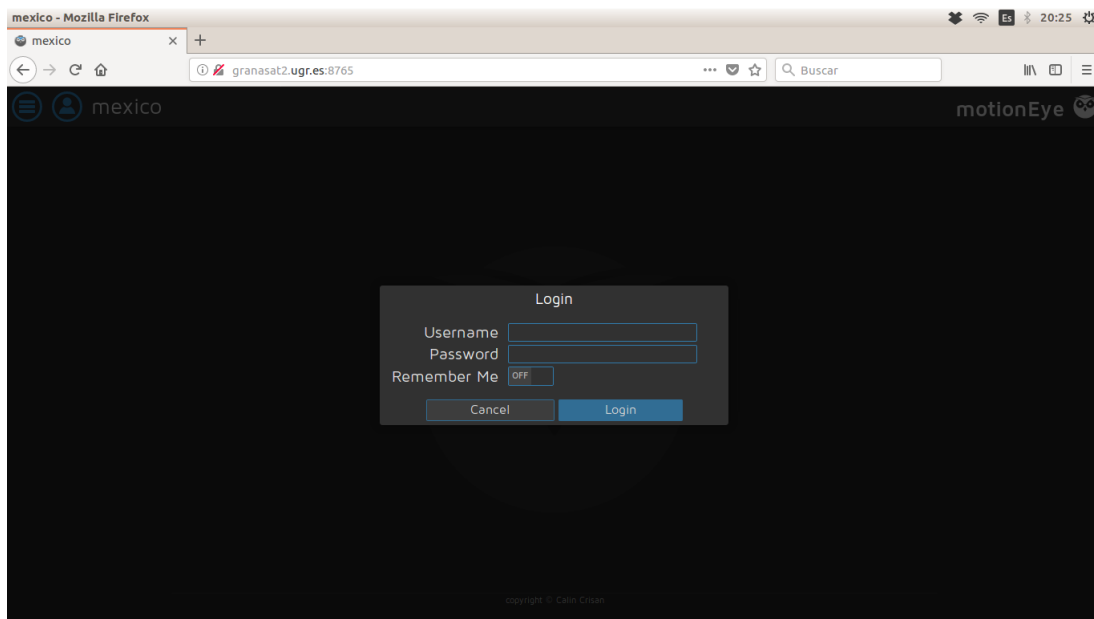


Figure 4.19 – MotionEye FrontEnd

Once the user logs in, this can easily define two cameras by using the GUI displayed. (by clicking on Add Camera tab). Some simple parameters are adjustable via GUI, such as: name of the cameras, schedule recording, video streaming enabler, etc. as it is shown in figure 4.20

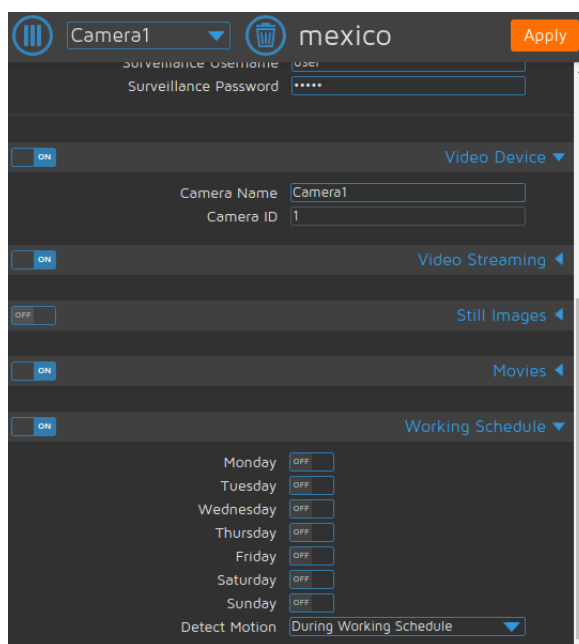


Figure 4.20 – MotionEye configuration

In addition, if necessary, it is possible to adjust more parameters for MotionEye from the file `/etc/motioneye/motioneye.conf`:

This configuration includes, among other parameters:

```

1
2 \# path to the configuration directory (must be writable by motionEye)
3 conf\_path \etc/motioneye
4
5 \# path to the directory where pid files go (must be writable by motionEye)
6 run\_path \var/run
7
8 \# path to the directory where log files go (must be writable by motionEye)
9 log\_path \var/log
10
11 \# default output path for media files (must be writable by motionEye)
12 media\_path \var/lib/motioneye
13
14 \# the log level (use quiet, error, warning, info or debug)
15 log\_level info
16
17 \# the IP address to listen on
18 \# (0.0.0.0 for all interfaces, 127.0.0.1 for localhost)
19 listen 0.0.0.0
20
21 \# the TCP port to listen on
22 port 8765
  
```

Furthermore, once a camera is defined, a `.conf` file will be created with the name `thread-<id>.conf`. In this case, two cameras have been defined, therefore, the files `/etc/motioneye/thread-1.conf` and `/etc/motioneye/thread-2.conf` have been created. In this files, the specific parameters of each camera can be defined. This includes:

```

1
2 \# Specify whether you want to record video when motion is detected or not \
3 ffmpeg\_output\_movies off/on \
4
5 \# Specify whether you want to take pictures when motion is detected or not \
6 output\_pictures off/on \
7
8 \# Camera brightness \
  
```

```

9| brightness 0-100 \\  

10| \  

11| \# Port where the video stream will be showed \<\  

12| stream\_port 8082 \<\  

13| \  

14| \# Text to be displayed in the video \<\  

15| text\_left "text" \<\  


```

In summary, two cameras have been set up: the first one will be pointing the antenna and the stream video will be seen from the [Dashboard](#) so that the users can see how the antenna changes its position. Regarding the second camera, this will be watching out the [Ground Station](#) and when motion is detected, a video will be saved in the folder `/var/lib/motioneye`.

4.6.2.2 Front-end

Once the video of the antenna is streaming in some port of the server (specifically in port 8081), this needs to be accessed from the client side so that the users can see the [Ground Station](#) in live.

This is easily made by pointing the server port within an `` tag in the html file.

```

```

To sum up, in figure 4.21 is shown how the implementation has been designed.

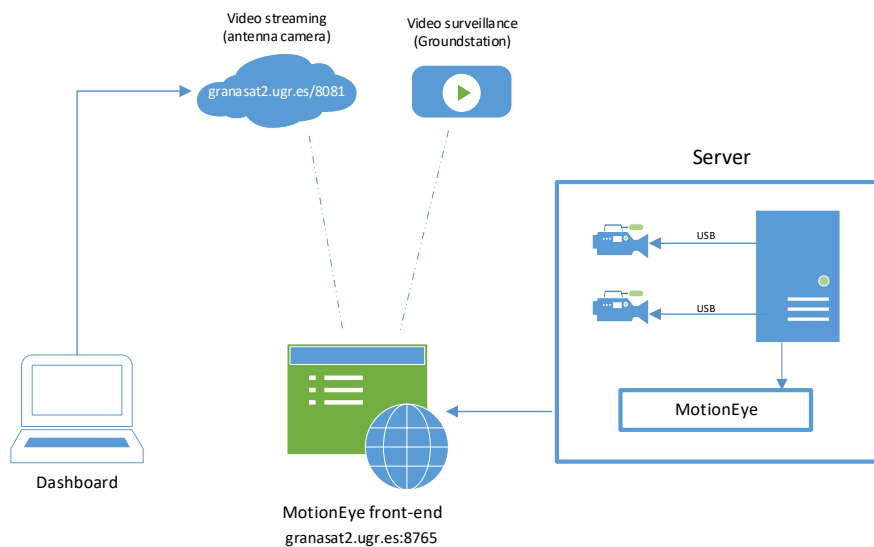


Figure 4.21 – *Streaming video final design*

4.6.3 User account management

The user is expected to modify their profile (username, organization, password, photo). For this, the user can introduce the information that will be updated in the front-end, data that will be verified by the controller before sending this to the server, which eventually will access the data base and will update the user's information.

A flow chart with the process is shown in 4.23.

This updating is made by the following function:

```

1  function modUser(req, res) {
2
3      var salt = createSalt();
4
5      var post = [
6          req.USER_NAME,
7          req.USER_ORGANIZATION,
8          req.USER_MAIL,
9          salt + ":" + hashPassword(req.USER_PASSWORD, salt),
10         req.USER_TYPE,
11         req.USER_BLOCKED,
12         req.USER_IMG,
13         req.USER_ID
14     ];
15
16
17
18
19     database.query('UPDATE USERS SET USER_NAME = ?, USER_ORGANIZATION = ?, USER_MAIL =
20     ?, USER_PASSWORD = ?, USER_TYPE = ?, USER_BLOCKED = ?, USER_IMG = ? WHERE USER_ID = ?
21     ', post, function(err) {
22         if (err) {
23             log(err.toString(), "error");
24             res({
25                 error: "Database error"
26             });
27         } else {
28             res({
29                 status: "Done"
30             });
31         }
32     });
33 }

```

The front-end with the form that allows to modify profiles can be seen in 4.22

Figure 4.22 – User account profile front-end

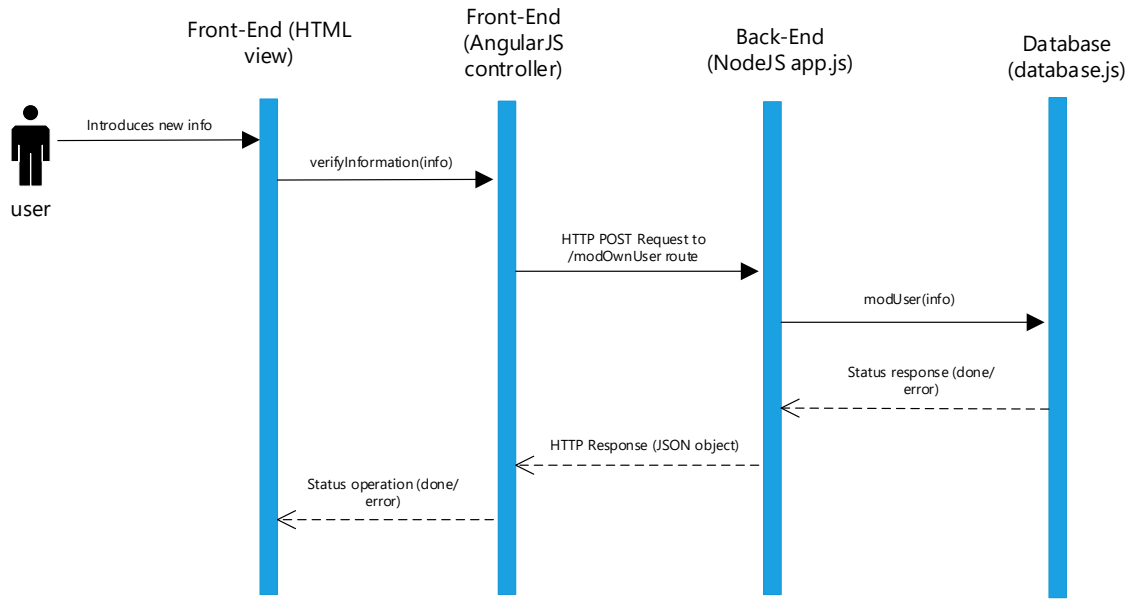


Figure 4.23 – User account management system flowchart

4.6.3.1 Recovery password system

When the user forgets their passwords, they can recover it with a recovery e-mail. The procedure is as follows:

1. The user introduce their e-mail in order to recovery their passwords. If the e-mail does not belong to the system, the user will see an error message.
2. A random password is created with crypto module.
3. A recovery e-mail with a temporary password is sent to the user (see figure 4.24). At the same time, the user's password is modified in the database with this temporary password.

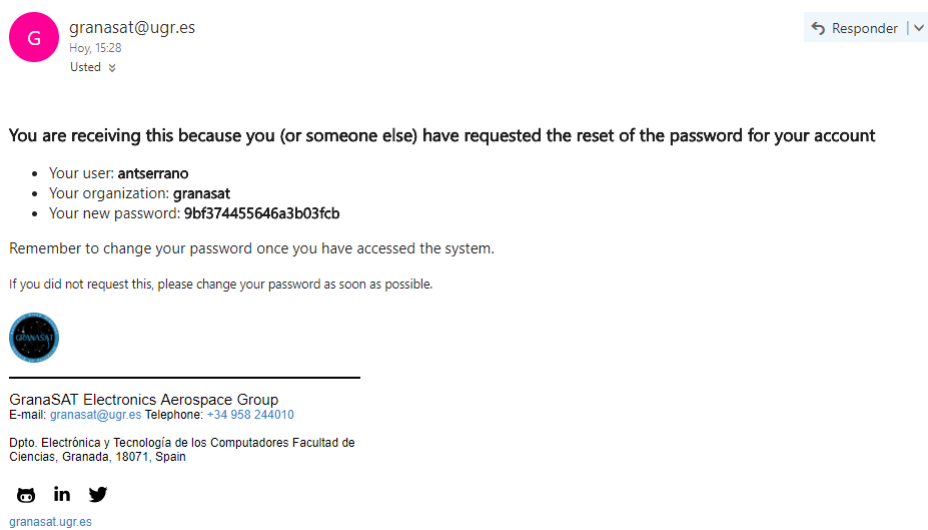


Figure 4.24 – *Recovery password e-mail*

4. The user checks their e-mail and gets the new password.
5. The user logs in the system with the new password and change it again by their convenience.

A flow chart with the process is shown in 4.25.

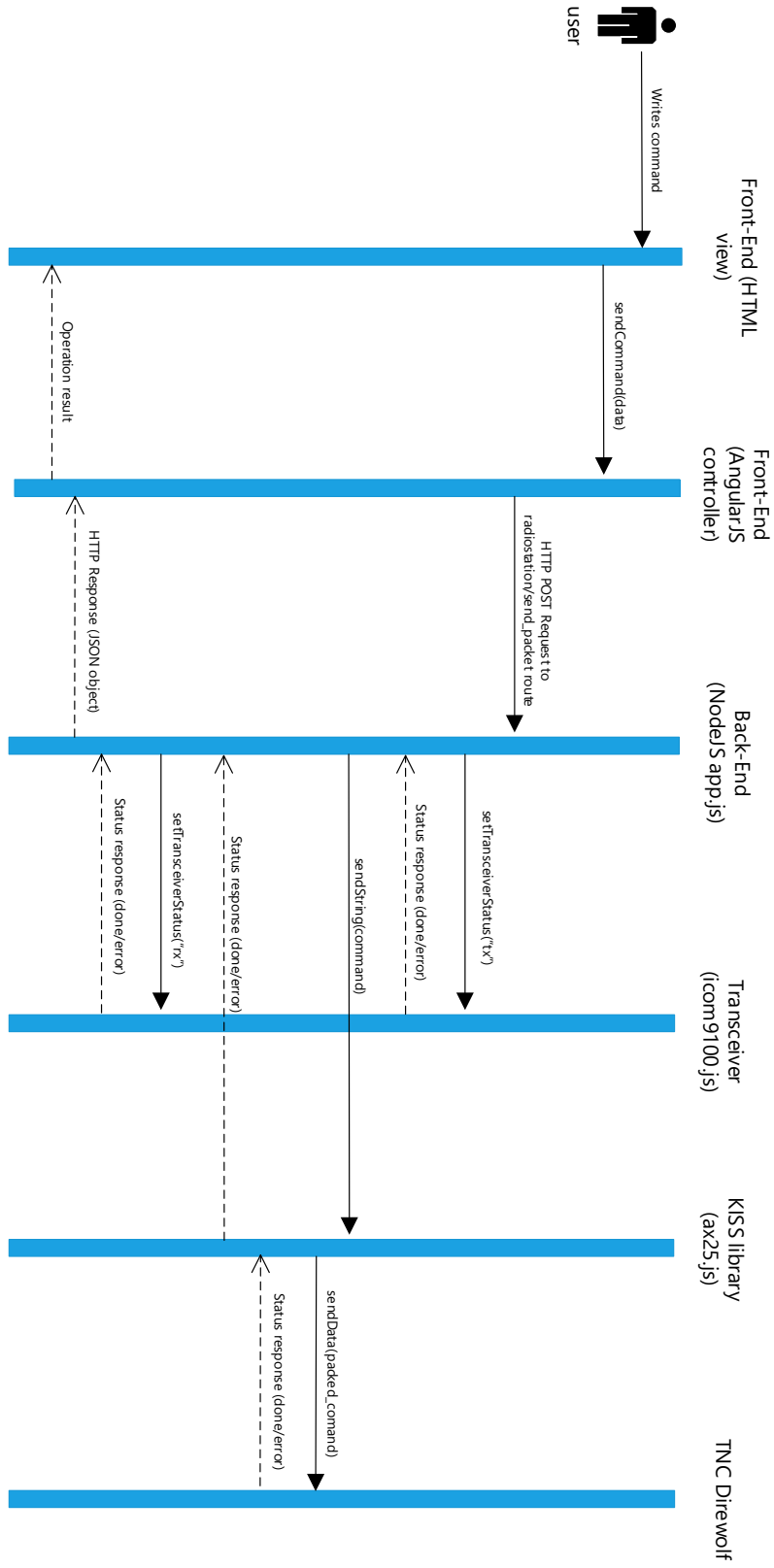


Figure 4.25 – Recovery password system flowchart

CHAPTER

5

TEST AND EVALUATION

This chapter introduces a number of tests that have been performed in order to validate the different designed solutions that have been seen in 4. Once all the test are passed and everything works as expected, the system is expected to go into production so that the real user can use it.

5.1 Transceiver's control verification

All the implemented commands have been progressively tested in both directions, that is, modifying the [Dashboard](#) and checking that the [transceiver](#) responds and adjust its parameters correctly; and other way around, modifying manually the [transceiver](#) parameters and verifying that the [Dashboard](#) updates the front-end (gauges, bars, buttons) correctly and synchronously with the [transceiver](#).

During the development it was very clear that the response time (time between changing something on the [Dashboard](#) and being reflected in the [transceiver](#) and viceversa) should be as short as possible. This has been properly tested and relatively short times has been achieved, with no more than one second of retard (see graphic [5.1](#)).

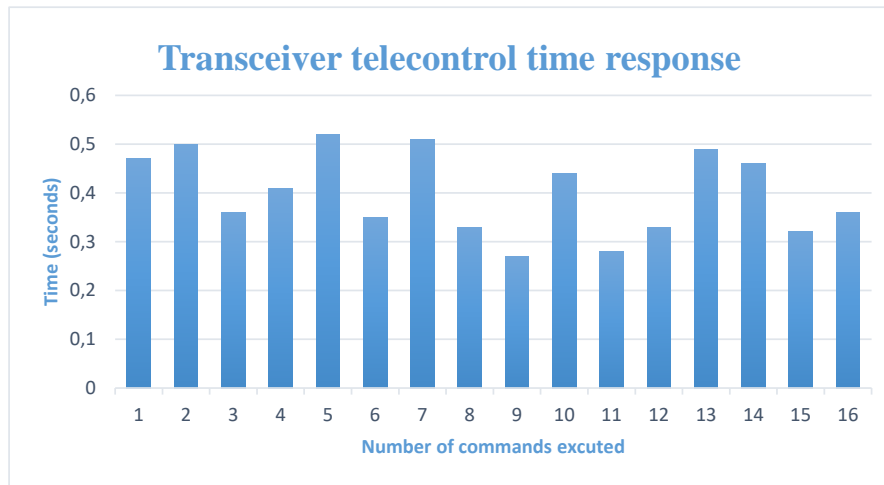


Figure 5.1 – *Transceiver control response time test*

Hence, a remote control of the [transceiver](#) has been correctly designed with virtually real time control, thereby meeting this client's requirement.

5.2 Transceiver's audio verification

To verify that the transceiver's audio works correctly, the delay time was checked with ten different connections and this was always between 2-4 seconds (see figure 5.2), which is an appropriate time regarding its purpose.

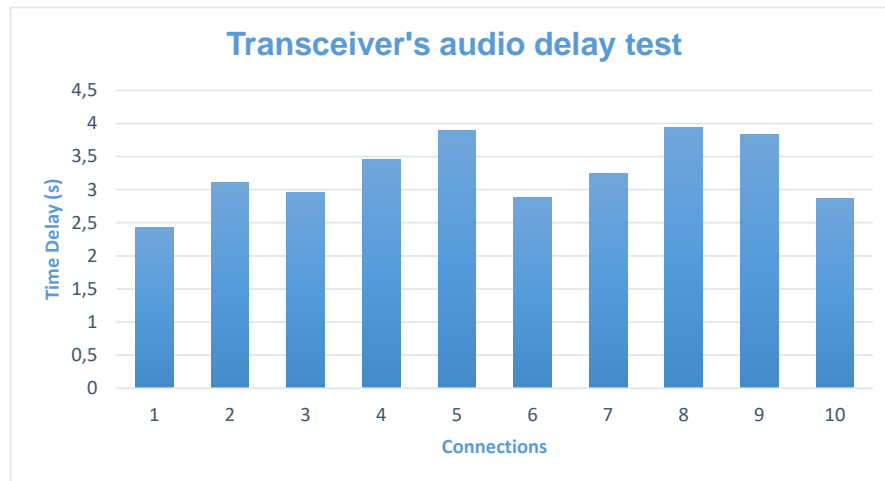


Figure 5.2 – *Transceiver's audio time delay test*

Furthermore, it is important that the audio streaming does not shut down after many hours running. This was correctly verified, being the audio streaming up during more than 5 days without any problem.

From the client's perspective, all the requirements have been met, since the audio can be heard perfectly from the [Dashboard](#) with a minimal time delay.

5.3 TNC (Direwolf) integration test

5.3.1 Receiving and decoding AX25 packets

Once [Direwolf](#) is running on the server, it is necessary to verify that this is working and decoding [AX25](#) packets as expected.

Since there is no real [Cubesat](#) transmitting these packets, some device needs to be used in order to simulate this. Therefore, an [Arduino](#) project will be used, being borrowed from one of our [GranaSAT](#) colleagues. [30]

This [Arduino](#) has several sensors (temperature, barometer, magnetometer, accelerometer and gyroscope). The device measures all of this parameters and transmits it through the [AX25](#) protocol over the 144.800MHz frequency (see figure 5.5).

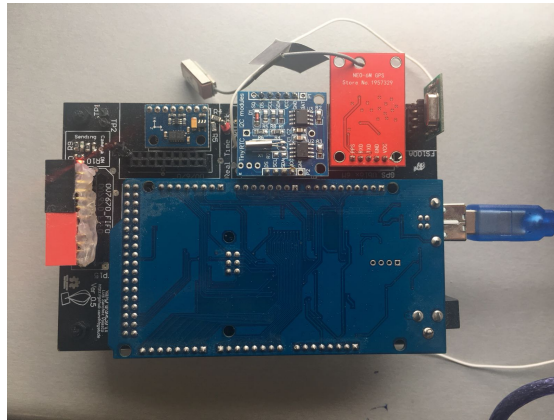
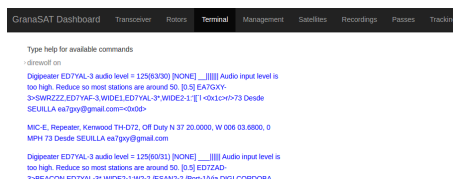


Figure 5.3 – *Arduino AX25 transmitter* [30]

Thus, in summary, the system will retrieve the signal that comes from the [Arduino](#) transmitter, signals that will be demodulated by the [transceiver](#), being converted into audio tones, which will be piped to [Direwolf](#) and decoded in order to get the telemetry that will be displayed on the [Dashboard](#) terminal (see demonstration video 5.1).



```

GransAT Dashboard  Transceiver  Roles  Terminal  Management  Satellites  Recordings  Pages  Tracks

Type help for available commands
direct in
Direwolf ED7YAL-3 audio level = 1256(930) [NONE] ||||| Audio input level is
too high. Reduce so most stations are around 50. [0.5] EA7GKY-
3-5WRZ2Z.ED7YAF-3.WIDE1.ED7YAL-3*WIDE2-1[T] <+>Lev+73 Desde
SEULLA ea7py@gmail.com<+>DSD-
MC-E, Repeat: Kenwood TH-D72, Off Duty N: 37 20.0000, W 008 03.8800, 0
MPH 73 Desde SEULLA ea7py@gmail.com
Direwolf ED7YAL-3 audio level = 1256(931) [NONE] ||||| Audio input level is
too high. Reduce so most stations are around 50. [0.5] ED7ZAD-
3-BEACON.ED7YAL-3*WIDE2-1.W2-2.ESAN2-2 /Port J via DKG CORDOBA

```

Video 5.1 – *Direwolf decoding packets test (double click)*

When setting the [transceiver](#) to the proper frequency where [Arduino](#) is transmitting, in this case 144.800MHz, the system is capable of decoding the signals.

Concretely, [Direwolf](#), which is connected to the audio card of the server, is getting the audio that comes from the [transceiver](#) and demodulating this audio.

The output is the decoded [AX25](#) frame sent from the [Arduino](#), which contains the destination and source callsigns, besides the data information (telemetry) itself separated by “/” (see [A](#) to see [AX25](#) frame content).

```

GSAT-12 audio level = 24(10/11) [NONE] ||||| [0.2] GSAT-12>APRS,WIDE-2::GSAGND-
18:00:02:01/0000.00N/00000.00EO/RoX=0.00/RoY=-3.06/RoZ=-0.65/AcX=0.00/AcY=-0.08/AcZ=9.30/MaX=-0.01/MaY=-0.33/MaZ=-0.21/T=28.77/P=936.21

APRS Message for "GSAGND-18", JEEP, Generic, (obsolete. Digits should use APNxxx instead)
00:02:01/0000.00N/00000.00EO/RoX=0.00/RoY=-3.06/RoZ=-0.65/AcX=0.00/AcY=-0.08/AcZ=9.30/MaX=-0.01/MaY=-0.33/MaZ=-0.21/T=28.77/P=936.21

```

Figure 5.4 – *AX25 decoded frame transmitted from Arduino [30]*

Once the frame is decoded, this can parse it in order to get a more human-readable output with the [Arduino](#) telemetry.

```

> getTelemetry
Pressure: 936.30mbar
Temperature: 28.89°C
Rotation x: 0.00°/s
Rotation y: -3.08°/s
Rotation z: -0.39°/s
Accelerometer x: 0.00m/s^2
Accelerometer y: -0.08m/s^2
Accelerometer z: -0.04m/s^2
Magnetometer x: -0.01T
Magnetometer y: -0.34T
Magnetometer z: -0.20T

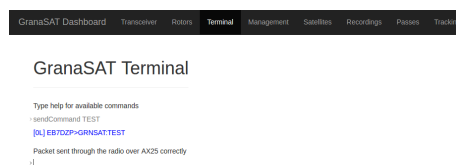
```

Figure 5.5 – *AX25 telemetry received from Arduino [30]*

5.3.2 Sending AX25 packets

On the other hand, it is necessary to verify that commands are being sent correctly, that is, commands sent by the user are really packed by [Direwolf](#) and sent through the [transceiver](#) over radio waves. For this, a walkie-talkie has been used (which is basically a transmitter-receiver) in order to verify that the packets are being sent through radio frequencies (it is important to set the walkie-talkie in the same frequency as the one that the [transceiver](#) will transmit to).

See demonstration video [5.2](#).



Video 5.2 – *Direwolf sending packets test (double click)*

As it is shown in the above video, the packets are being sent correctly. The user introduces on the [Dashboard](#) the command to be sent and this is automatically packed by [Direwolf](#) and transmitted by the radio. Everything works as a black box for the user since this introduces the command to be sent.

The point is that these packets will be received some day by a [Cubesat](#) receiver, instead of being received by a simple walkie-talkie.

In summary, the [TNC](#) has been integrated in the system correctly and it works as expected, hence complying with the client's requirements.

5.4 Tracking satellite verification

In order to verify that the tracking satellite works as expected and it shows the satellites orbits and parameters correctly, this have been compared with current real-time developed trackings, such as <http://www.n2yo.com>, where many satellites can be tracked with several parameters, such as current speed, altitude, latitude, etc.

Several tests were performed with different satellites in order to verify that the system always works. In figures 5.6 and 5.7 it can be seen a comparison between the satellite tracking developed on the [Dashboard](#) and the satellite tracking available at <http://www.n2yo.com>.

For this example, the NOAA 15 satellite was selected.

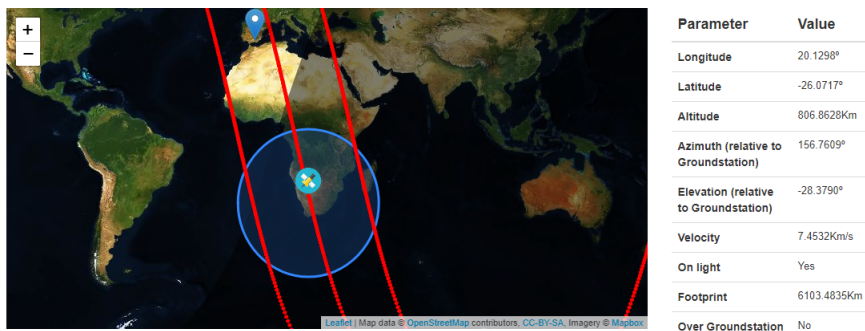


Figure 5.6 – NOAA 15 tracking in GranaSAT Ground Station

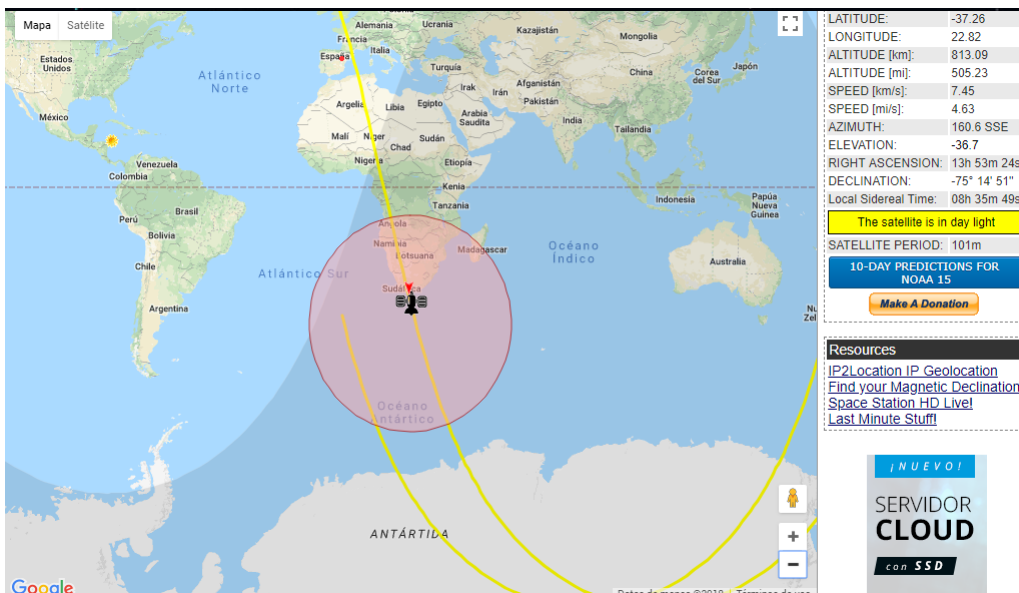


Figure 5.7 – NOAA 15 tracking in www.n2yo.com

As it is shown in the above images, the system works and shows the satellites position similarly as other well known programs do. Hence, the satellite tracking complies with the specified client's requirements and works as expected from the client's perspective.

5.5 Other improvements

- **Cameras system:** as it has been seen during the design stage, a camera system has been deployed over the port 8765 of [GranaSAT](#) server. Here the [Motion GUI](#) is available and it allows to set several cameras, provided their configuration files. In this case, the two required web-cams were correctly installed and they are working properly. With the system deployed, anyone could add a new camera very easily making use of the [Motion GUI](#), which definitely makes [Motion](#) the best solution that could have been chosen.

Hence, this secondary requirement has been achieved.

- **Tooltips design:** from the client's perspective, it was expected a tooltip's module that was easily scalable, allowing in the future the inclusion of more tooltips, in order to make easier the use of the system for amateur users.

Therefore, this secondary requirement has been correctly met since all the tooltips have been included in a single file, which complies with the system modularity and ease of scalability

- **User profile management:** something that the system lacked was the possibility of modifying the user's profiles, something really important taking into account that nowadays this is possible in almost every web page. With the implemented design, the users are now able to modify their username, organization, profile photo and passwords.

It was verified that when the user updates their information, this is updated in the data base as well. On the other hand, all the possible errors that the users could introduce while modifying their information is handled as well.

Therefore, this secondary requirements complies with the client's requirements.

- **Recovery password e-mail:** in the last version of the system, it was no possible to recover the user password, therefore, when the users forgot their passwords, they were no longer available to access the system. This is something clearly inadmissible.

With the new implementation, when the user forgets their password, they can easily introduce their e-mail and instantly receive a new temporary password for the system.

It was properly verified with different e-mail accounts that the e-mails are received correctly in less than 10 seconds since they introduce their e-mail. On the other hand, if the introduced e-mail does not belong to the system, no e-mail is sent, being displayed an error instead.

Hence, this secondary requirement has been met as well.

5.6 Browser compatibility testing and performance

The final version of the [Dashboard](#) was tested and it has a correct user experience with the following browsers:

- Google Chrome 66.0.3359.139 (64 bits)
- Mozilla Firefox 60.0 (32 bits)
- Safari 11
- Microsoft Edge 41.16299.402.0

Regarding the application performance, this has been tested in Microsoft Edge v.41 (considered slower than its competitors) in Windows 10, with a medium/high-end computer, concretely a Intel(R) Core(TM) i7-3520M CPU @ 2.90 GHz.

The results are reasonable good taking into account that the system is constantly making requests to the server in order to obtain the current [transceiver](#) status, rotor's status, [transceiver](#)'s audio, information of the satellite tracking, etc.

The obtained results are the following:

- **CPU:** the CPU performance oscillates between 5 and 20%, which is a good performance taking into account that there are more processes in the computer than the [Dashboard](#). See figure [5.8](#) for further details.

- **RAM:** the RAM keeps stable most of the time the application is running, which is around the 40-50%, something reasonable. See figure [5.9](#) for further details.

- **Disk:** the disk performance is about 30%, having some peaks when the satellite tracking was active, something reasonable taking into account the amount of calculations. See figure [5.10](#) for further details.

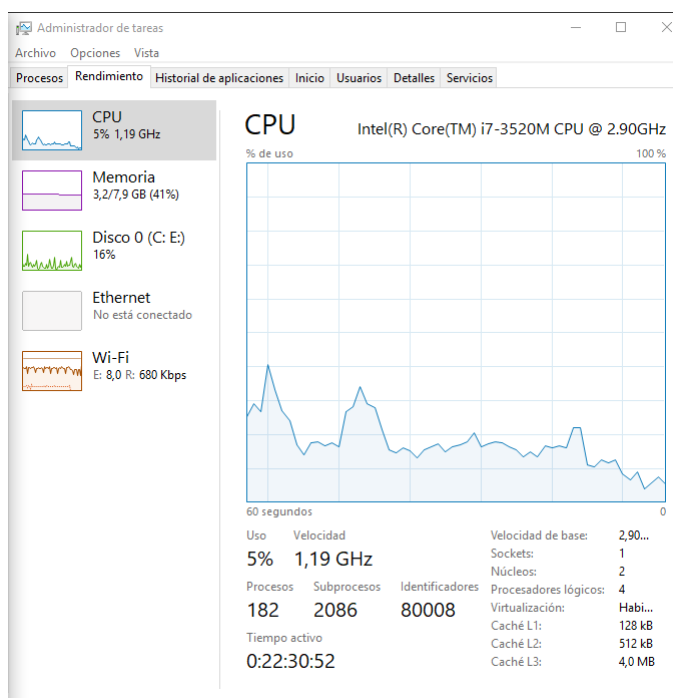


Figure 5.8 – Dashboard CPU performance test

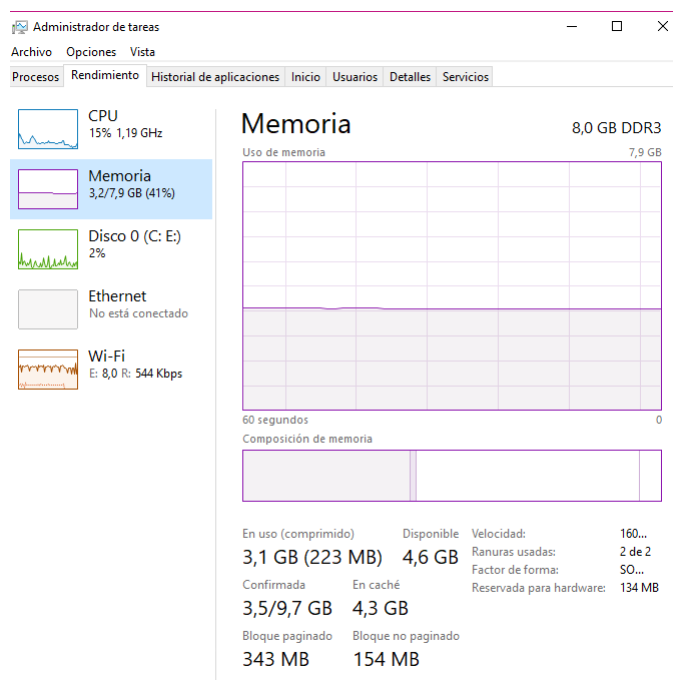


Figure 5.9 – Dashboard memory performance test



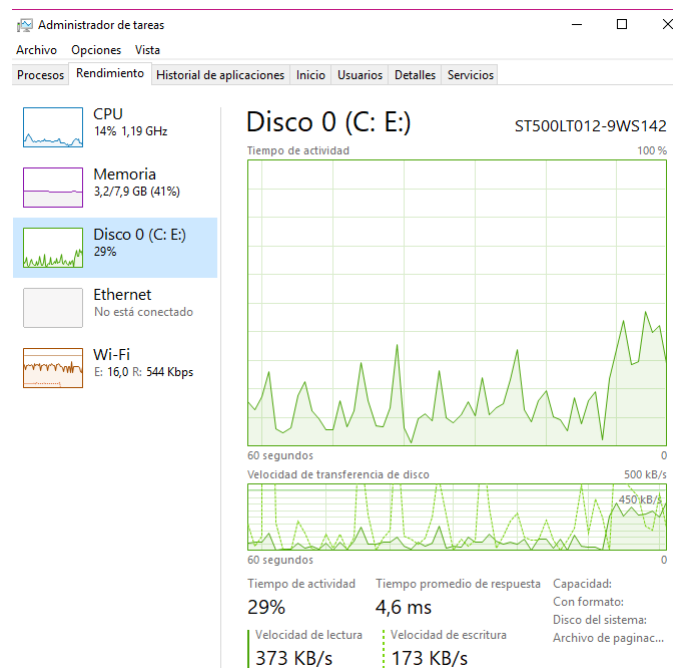


Figure 5.10 – *Dashboard disk performance test*

Lastly, once all the mentioned tests have been successfully verified, the system is expected to go into production, that is, being definitely running on the server for its usage. Therefore, the application will be running in granasat2.ugr.es and it will be managed by PM2 (see appendix D).

On the other hand, an [SSL](#) certificate provided by [GranaSAT](#) has been installed (see appendix E) so all the traffic of the web application is safe.

CHAPTER

6

CONCLUSIONS AND FUTURE LINES

During this document has been displayed the analysis, design and verification of a web-based system ([Dashboard](#)) that allows to control the [GranaSAT Ground Station](#) remotely. The project was intended to follow a product-oriented philosophy, where the final system is not the result of a simple programming process, but from a deep analysis of the client's requirements and how to approach the later design and implementation. This philosophy has allowed the student to get closer to a real and professional problem.

Hence, from the student's perspective, the presented project has supposed a big challenge. During the project development, a vast amount of problems had to be solved and therefore, a big amount of new knowledge had to be acquired. The first problem came with the fact that all the project relies on the web programming, something that the student had not seen before during his Bachelor's Degree. Furthermore, other electronics aspects related to different devices that are used in satellite communications were totally new for the student. All in all, the student acquired new concepts related to Aerospace and Electronics, besides Computer Engineering itself.

As final conclusion, it must be specially remarked that this Final Project has contributed to arouse the student's interest about Aerospace Engineering, a field that the student did not have the opportunity to research about during his Degree. This final system contributes to the [GranaSAT](#) project and its purpose of developing a [Cubesat](#), which makes all the faced problems and difficulties worth it.

Regarding future lines for this project, there is still work to do in future projects, for instance:

- Developing a native application for Linux, Windows or iOS, making use of tools such as Electron
- Developing a complete mission control with defined commands and telemetry, (once the [Cubesat](#) is developed) similar to FUNCube Telemetry Dashboard
- Developing different modules on the [Dashboard](#) that allow communication with other satellites, such as [ISS](#)
- Integrating other hardware devices on the [Dashboard](#), such as other radios
- Improving the transceiver's audio streaming, achieving real time
- Developing a module that allow users to transmit voice remotely, as though they were using physically the microphone in the [Ground Station](#)
- Developing a social media module that allow possible organizations and users to interact with each other (possible [Cubesat](#) community)

APPENDIX

A

AX25 PROTOCOL

[AX25](#) (Amateur X.25) is a data link layer protocol derived from the X.25 protocol suite and designed for use by amateur radio operators. It is used extensively on amateur packet radio networks. [AX25](#) is basically an amateur radio specification that describes how to encode digital data in order to transmit it over radio frequencies. This specification mandates a bit rate of 1200 baud and uses [AFSK](#) in order to represent binary values 0 and 1 with tones of 1200Hz and 2200Hz, respectively. [35]

[AX25](#) packets are sent in small blocks of data called frames, which are made up of 9 fields (see figure A.1). [AX25](#) have different frames format but specifically, the Unnumbered Information (UI) frame format is utilized in most [Cubesat](#) communications schemes. [35].

Flag	AX.25 Transfer Frame Header (128 bits)				Information Field	Frame-Check Sequence	Flag
	Destination Address	Source Address	Control Bits	Protocol Identifier			
8	56	56	8	8	0-2048	16	8

Figure A.1 – [AX25](#) frame structure [15]

- Flag: it identifies the beginning and end of a frame so that the receiver can detect and identify each received frame. This is made of the bit sequence 0x7e
- Destination Address: this field contains the destination station (six upper-case letters,

1

also known as CALLSIGN) and SSID (four-bit integer, extra identification in case there is more than one station using the same CALLSIGN).

- Source Address: this field contains the CALLSIGN and SSID of the transmitting station. Specifically, [GranaSAT Ground Station](#) CALLSIGN is D70DZP.
- Digipeater Addresses: from zero to 8 digipeater callsigns may be included in this field (digipeaters are stations that repeat the frames over other stations)
- Control Field — this field is set to 0x03
- Protocol ID — this field is set to 0xf0 (no layer 3 protocol).
- Information Field — this is the most important field, since it contains the information (message) to be sent (no more than 256 bytes). Here the telecommands are introduced, as well as the telemmetry is contained in this field.
- Frame Check Sequence: this a sequence of 16 bits used for checking the integrity of a received frame.

In summary, this set of bytes (0's and 1's all in all) are codified into tones of 1200Hz and 2200Hz, which allows to transmit this data over radio frequencies.

APPENDIX

B

DARKICE AND ICECAST2: INSTALLATION

Icecast2 can be easily installed in Linux with the following command:

```
$sudo apt-get install icecast2
```

After this, open the file `/etc/default/icecast2` and change the last line to:

```
ENABLE=true
```

If necessary, modify the server admin user and password, besides the port where it will run and other parameters from the file `/etc/icecast2/icecast2.xml`

Once everything is ready, start Icecast2:

```
$/etc/init.d/icecast2 start
```

To stop it:

```
$/etc/init.d/icecast2 stop
```

Regarding Darkice, this is installed with the following command:

```
$sudo apt-get install darkice
```

2

APPENDIX

C

DIREWOLF: INSTALLATION

In Linux, [Direwolf](#) can be easily installed from the git repository with the following command:

```
$git clone https://www.github.com/wb2osz/direwolf
```

```
$cd direwolf
```

It might be necessary to install the following sound library:

```
$sudo apt-get install libasound2-dev
```

After this, compile the application:

```
$cd /direwolf
```

```
$make
```

```
$sudo make install
```

```
$make install-conf
```

This will install [Direwolf](#) in the system. Concretely, while running [Direwolf](#) on the [Dashboard](#), the following options are used:

- `-p`: It creates a virtual serial port (`/tmp/kisstnc`) in order to let [KISS](#) applications connect
- `-n 2`: It sets number of channels, in our case 2, one for output and one for input

- `-c direwolf.cfg`: It specifies the configuration file to be taken
- `-t 0`: [Direwolf](#) output is displayed with colors, this option removes any color and displays the output as a plain text

APPENDIX

D

PM2: INSTALLATION AND USE

Once the web application is finished, the production environment needs to be set up so that the [NodeJS](#) application is always running on our server, attending the users requests. To do this, [PM2](#) has been used [23], an open source production process manager for Node.js. With this software all the services of our server can be monitored.

The last version of [PM2](#) can be installed from the terminal:

```
$npm install pm2@latest -g
```

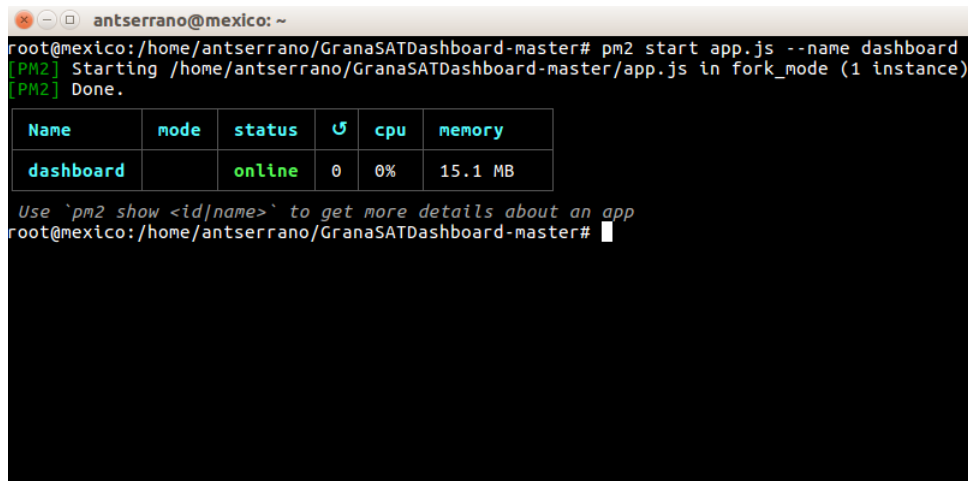
To start our application and assign it a recognizable name ("dashboard" in this case):

```
$pm2 start app.js --name dasdhdboard
```

Hereinafter, the application will be running on the server and it can be monitored and managed with [PM2](#) commands:

- `$pm2 list`: Display all processes status
- `$pm2 monit`: Monitor all processes
- `$pm2 describe 0`: Display all informations about a specific process
- `$pm2 stop all`: Stop all processes
- `$pm2 restart all`: Restart all processes
- `$pm2 reload all`: Will 0s downtime reload (for NETWORKED apps)

- `$pm2 stop 0`: Stop specific process id
- `$pm2 restart 0`: Restart specific process id
- `$pm2 delete 0`: Will remove process from pm2 list
- `$pm2 delete all`: Will remove all processes from pm2 list



```
anterrano@mexico: ~
root@mexico:/home/anterrano/GranaSATDashboard-master# pm2 start app.js --name dashboard
[PM2] Starting /home/anterrano/GranaSATDashboard-master/app.js in fork_mode (1 instance)
[PM2] Done.

  Name      mode  status  ⬆  cpu  memory
  dashboard  -    online  0   0%  15.1 MB

Use `pm2 show <id|name>` to get more details about an app
root@mexico:/home/anterrano/GranaSATDashboard-master#
```

Figure D.1 – *PM2 Execution*

APPENDIX

E

SSL CERTIFICATE INSTALLATION

Another important aspect of our application is that this should allow user access it making use of the [HTTPS](#) protocol (over port 443), allowing secure connections. This certificate has been provided by the University of Granada.

The installation is made along with [nginx](#), an open-source high performance web server. First of all, the server configuration is defined in the file `/etc/nginx/sites-available/default`:

```
1 server {
2
3     listen 80;
4     return 301 https://$host$request_uri;
5 }
6
7 server {
8
9     server_name granasat2.ugr.es;
10
11     listen 443 ;
12
13     ssl on;
14     ssl_certificate /certificados/bundle.crt;
15     ssl_certificate_key /certificados/granasat2_ugr_es.key;
16
17
18     ssl_session_cache builtin:1000 shared:SSL:20m;
19     ssl_session_timeout 180m;
20     ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
21     ssl_prefer_server_ciphers on;
22     ssl_ciphers HIGH:!aNULL:!eNULL:!EXPORT:!CAMELLIA:!DES:!MD5:!PSK:!RC4;
23
24     location / {
25         #try_files $uri $uri/ =404;
26
27         proxy_set_header Host $host;
28         proxy_set_header X-Real-IP $remote_addr;
29         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
30         proxy_set_header X-Forwarded-Proto $scheme;
31
32
33         proxy_pass http://granasat2.ugr.es:8002;
34         proxy_read_timeout 90;
35
36         proxy_redirect http://granasat2.ugr.es:8002 https://granasat2.ugr.es;
```



```
37 }
38 }
39
40
41 location /camera {
42     proxy_pass http://localhost:8081;
43 }
44
45 location /direwolf {
46     proxy_pass https://localhost:8003;
47     proxy_http_version 1.1;
48     proxy_set_header Upgrade $http_upgrade;
49     proxy_set_header Connection "upgrade";
50 }
51
52 location /audio {
53     proxy_pass http://localhost:8000/streaming;
54 }
55
56 }
```

Here the [SSL](#) certificate location must be defined, besides the configuration that redirect the [HTTP](#) traffic to [HTTPS](#) (including the cameras, audio and [Direwolf](#) websocket).

After modifying this file, restart [nginx](#):

```
$/etc/init.d/nginx restart
```

APPENDIX

F

MOTIONEYE: INSTALLATION AND CONFIGURATION

To install MotionEye on our server (Debian 9) [6], the following steps are followed (notice that these commands need sudo mode).

- 1. Install motion, ffmpeg and v4l-utils (packages needed for handling media devices, recording, etc).

```
$apt-get install motion ffmpeg v4l-utils
```

- 2. Install dependencies from repositories:

```
$apt-get install python-pip python-dev curl libssl-dev libcurl4-openssl-dev libjpeg-dev
```

Note: python 2.7 is required.

- 3. Install motion eye:

```
$pip install motioneye
```

- 4. Prepare the configuration directory:

```
$mkdir -p /etc/motioneye
```

```
$cp /usr/local/share/motioneye/extra/motioneye.conf.sample  
/etc/motioneye/motioneye.conf
```

- 5. Prepare the media directory (folder where videos will be saved):

```
$mkdir -p /var/lib/motioneye
```

- 6. Add an init script, configure it to run at startup:

```
$cp /usr/local/share/motioneye/extra/motioneye.systemd-unit-local  
/etc/systemd/system/motioneye.service  
$systemctl daemon-reload  
$systemctl enable motioneye
```

After this, it is possible to:

- Start motioneye: `$systemctl start motioneye`
- Stop motioneye: `$systemctl stop motioneye`
- Restar motioneye: `$systemctl restart motioneye`
- Upgrade motioneye: `$pip install motioneye --upgrade`

APPENDIX

G

PROJECT BUDGET

G.1 Hardware Cost

Regarding hardware costs, as detailed during the project, the [Ground Station](#) is made up of several devices, besides the connectors needed between these devices and the server. In addition, some cameras were bought.

The costs of all the hardware components are detailed in table [G.1](#).

Hardware	Costs (€)
Intel(R) Xeon(R) CPU 5110 @ 1.60GHz, 4 cores	199
Icom IC-9100 transceiver HF/VHF/UHF	3486
Yaesu G5500 Rotors	650
HUB USB	7.99
USB wires 5m	10
Logitech webcam	9.99
LG webcam	12.99
TOTAL	4375.97 €

Table G.1 – *Hardware costs*

G.2 Software Cost

This project has required the use of many different software, frameworks and applications. The prices of these are detailed in table G.2.

Software	License Owner	Cost (€)
IntelliJ IDEA	Author	Student License (Free)
NodeJS (express)	Author	Free
Twitter Bootstrap	Author	Free
AngularJS	Author	Free
MotionEye	Author	Free
TeXnicCenter	Author	Free
Miktex	Author	Free
SumatraPDF	Author	Free
Visio	Author	Trial (Free)
Direwolf	Author	Free
Inkscape	Author	Free
icecast2, darkice	Author	Free
Microsoft Excel 2016	Author	Trial (Free)
	TOTAL	0 €

Table G.2 – Software costs

G.3 Human Resources Cost

Regarding human resources costs, the presented project has required hiring two people. The first one is a **junior engineer**, (10 €/h), hired as a part-time worker (3h/day) during ten months. Secondly, as project supervisor a **senior engineer** is hired, (50 €/h), computing 2 hours per week. Therefore, human resources amounts to **10160 €**, as detailed in table G.3.

Position	Time (hours)	Cost (€)
Junior Engineer	600	6000
Senior Engineer	80	4160
	TOTAL	10160 €

Table G.3 – Human resources costs

REFERENCES

- [1] ACIÉN, F. Github account. <https://github.com/acien101>.
- [2] ALSA. Advanced linux sound architecture. Available at: https://www.alsa-project.org/main/index.php/Main_Page.
- [3] AMSAT. Keplerian “two line element” set format. Available at: <https://www.amsat.org/keplerian-elements-formats/>.
- [4] AULAFORMATIVA. Javascript libraries for interactive maps. Available at: <http://blog.aulaformativa.com/librerias-de-javascript-plugin-para-crear-mapas-interactivos/>.
- [5] CCRISAN. Motioneeye github. Available at: <https://github.com/ccrisan/motioneye>.
- [6] CCRISAN. Motioneeye installation on debian. Available at: <https://github.com/ccrisan/motioneye/wiki/Install-On-Debian>.
- [7] DARKICE. Darkice audio streamer. Available at: <http://www.darkice.org/>.
- [8] DORON, T. Google style gauges using d3.js. Available at: <http://tomerdoron.blogspot.com.es/2011/12/google-style-gauges-using-d3js.html>.
- [9] ECHICKEN. A kiss and ax.25 stack for node.js. Available at: <https://github.com/echicken/node-ax25>.
- [10] GARRIDO, P. Github account. Available at: <https://github.com/pablogs9>.

References

- [11] GAUGE.JS. Javascript gauge library. Available at: <http://bernii.github.io/gauge.js/>.
- [12] GITHUB, J. A javascript port of the popular predict satellite tracking library. Available at: <https://github.com/nsat/jspredict>.
- [13] ICECAST. Icecast streaming media server. Available at: <http://www.icecast.org/docs/icecast-2.4.1/>.
- [14] ICOM. Ic-9100 instruction manual. Available at: http://www.icom-australia.com/products/amateur/ic-9100/Amateur_IC-9100_Instruction_Manual.pdf.
- [15] JOE FITZGERALD, A. Available at: <https://www.amsat.org/why-is-there-so-much-tle-confusion-when-new-cubesats-are-launched/>.
- [16] JOERGDIETRICH. Leaflet terminator. Available at: <https://github.com/joergdietrich/Leaflet.Terminator>.
- [17] JUSTGAUGE. Javascript plugin for gauges. Available at: <http://justgage.com/>.
- [18] KEYCDN. Top 10 front-end frameworks of 2016. Available at: <https://www.keycdn.com/blog/front-end-frameworks/>.
- [19] LEAFLET. Javascript library for interactive maps. Available at: <http://leafletjs.com/>.
- [20] MARÍN, A. Github account. Available at: <https://github.com/albertomn86>.
- [21] NODE-SERIALPORT. Serial-port library. Available at: <https://www.npmjs.com/package/serialport#opening-a-port>.
- [22] ORG, A. O. Dsnoop. Available at: <https://alsa.opensrc.org/Dsnoop>.
- [23] PM2. Pm2 webpage. Available at: <http://pm2.keymetrics.io/>.
- [24] PYEPHEM. Astronomical algorithms library in python. Available at: <http://rhodesmill.org/pyephem/>.
- [25] SATELLITE.JS GITHUB. A library to make satellite propagation via tle's possible in the web. Available at: <https://github.com/shashwatak/satellite-js>.
- [26] STACKOVERFLOW. Developer survey results 2018. Available at: <https://insights.stackoverflow.com/survey/2018/#most-loved-dreaded-and-wanted>.
- [27] TOOLTIPS.JS. tooltips.js website. Available at: <https://popper.js.org/tooltip-examples.html>.
- [28] TWITTER. Tooltips bootstrap documentation. Available at: <https://v4-alpha.getbootstrap.com/components/tooltips/>.

- [29] UNIVERSITY, A. Aau cubesat. Available at: <http://www.space.aau.dk/cubesat/>.
- [30] VELASCO, L. S. Design of an embedded camera with an ax.25 transmitter.
- [31] WIKIPEDIA. Ground station and ground segment. Available at: https://en.wikipedia.org/wiki/Ground_station#Telecommunications_port & https://en.wikipedia.org/wiki/Ground_segment#Ground_stations.
- [32] WIKIPEDIA. Terminal node controller (tnc). https://es.wikipedia.org/wiki/Protocolo_AX.25.
- [33] WIKIPEDIA. Packet radio. Available at: [https://es.wikipedia.org/wiki/Packet_\(Radio\)](https://es.wikipedia.org/wiki/Packet_(Radio)).
- [34] WIKIPEDIA. Two-line element set. Available at: https://en.wikipedia.org/wiki/Two-line_element_set.
- [35] Y. A. AHMAD1, N. J. N., AND YUHANIZ, S. S. Design of a terminal node controller hardware for cubesat tracking applications. Available at: <http://iopscience.iop.org/article/10.1088/1757-899X/152/1/012031/pdf>.