



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Escola d'Enginyeria de Telecomunicació  
i Aeroespacial de Castelldefels

# TREBALL DE FI DE GRAU

**TFG TITLE:** Implementation of a flowgraph-based satellite operations software for Earth Observation missions

**DEGREE:** Double bachelor's degree in Aerospace Systems Engineering and Network Engineering

**AUTHOR:** Júlia Alós Mairal

**ADVISORS:** Hyuk Park  
Adrián Pérez-Portero

**DATE:** July 20, 2023



**Títol:** Implementació d'un software d'operacions per satèl·lit basat en diagrames de flux per a missions d'observació de la Terra

**Autor:** Júlia Alós Mairal

**Directors:** Hyuk Park  
Adrián Pérez-Portero

**Data:** 20 de juliol de 2023

## Resum

Aquest projecte té com a objectiu desenvolupar un programari de missió crítica que faciliti la supervisió i automatització del pla d'operacions entre el Centre d'Operacions i els CubeSats. Aquest programari ajudarà els operadors en diverses tasques, com programar les comunicacions amb els satèl·lits, controlar una o diverses Estacions Terrestres per seguir el satèl·lit, preparar plans d'execució amb contingències per a totes les diferents etapes del protocol de comunicació i automatitzar aquests processos.

Per minimitzar els errors introduïts pels operadors, el programari oferirà una interfície d'usuari interactiva per configurar conjunts de missatges i intercanviar informació durant el contacte. També permetrà la configuració de blocs condicionals que depenen de les dades rebudes, creant un bucle de retroalimentació sense problemes i sense errors.

L'objectiu final és reduir gradualment la càrrega de treball de l'operador fins al punt de fer innecessària la seva interacció. Això permetrà la comunicació automatitzada amb el satèl·lit en qualsevol moment del dia. Com a part de les operacions, totes les dades intercanviades s'emmagatzemaran per al seu posterior tractament, amb un processament automatitzat sempre que sigui possible.

El programari es desenvoluparà utilitzant el llenguatge de programació Rust, conegut per la seva velocitat, seguretat de memòria i seguretat de fils d'execució. El compilador Rust detecta una quantitat important d'errors comuns en temps de compilació, això permetrà el desenvolupament d'una aplicació altament fiable i d'alt rendiment.

Si bé el projecte es centrarà inicialment en donar suport al satèl·lit <sup>3</sup>Cat-4, també crearà les bases per operar qualsevol altre satèl·lit en el futur, com ara el RITA Payload.



**Title :** Implementation of a flowgraph-based satellite operations software for Earth Observation missions

**Author:** Júlia Alós Mairal

**Advisors:** Hyuk Park  
Adrián Pérez-Portero

**Date:** July 20, 2023

## Overview

This project aims to develop mission-critical software that facilitates the monitoring and automation of the operations plan between the Operation Center and the CubeSats. This software will assist operators in various tasks, including scheduling satellite passes, controlling one or multiple Ground Stations to follow the satellite, preparing execution plans with contingencies for all the different steps in the communication protocol, and automating these processes.

To minimize errors introduced by operators, the software will offer an interactive user interface for configuring message sets and information exchange during contact. It will also allow for the setup of conditional blocks that depend on received data, creating a seamless and error-free feedback loop.

The objective is to gradually reduce the operator's workload, to the point of making their interaction unnecessary. This will enable automated communication with the satellite at any time of day. As part of the operations, all uploaded and downloaded data will be stored for posterior processing, with automated processing wherever possible.

The software will be developed using the Rust programming language, known for its speed, memory safety, and thread safety. Rust compiler detects a significant amount of common errors at compile-time, this will allow the development of a highly reliable and high-performance application.

While the project will initially focus on supporting the <sup>3</sup>Cat-4 satellite, it will also create the basis to operate any other satellite in the future, such as the RITA Payload.



# CONTENTS

<b>Acknowledgements</b> . . . . .	<b>1</b>
<b>Acronyms</b> . . . . .	<b>3</b>
<b>Introduction</b> . . . . .	<b>5</b>
<b>CHAPTER 1. State of the art</b> . . . . .	<b>7</b>
<b>1.1. Introduction to Satellites and CubeSats</b> . . . . .	<b>7</b>
<b>1.2. Satellite operations</b> . . . . .	<b>7</b>
1.2.1. Ground segment . . . . .	8
1.2.2. Telemetry, Tracking and Command . . . . .	8
1.2.3. Satellite Orbit Analysis and Tracking . . . . .	9
<b>1.3. NanoSat Laboratory</b> . . . . .	<b>15</b>
1.3.1. Missions . . . . .	16
1.3.2. Ground segment . . . . .	17
<b>CHAPTER 2. Conceptual design</b> . . . . .	<b>19</b>
<b>2.1. Ground Segment management</b> . . . . .	<b>19</b>
2.1.1. Satellite management . . . . .	20
2.1.2. Ground Station management . . . . .	20
2.1.3. Scheduling module . . . . .	20
<b>2.2. Flowgraph-based TT&amp;C</b> . . . . .	<b>22</b>
2.2.1. Flowgraph . . . . .	22
2.2.2. Flowgraph edition modes . . . . .	27
2.2.3. Execution manager . . . . .	28
2.2.4. Flowgraph execution modes . . . . .	30
<b>2.3. Data downlink and storage</b> . . . . .	<b>31</b>
<b>2.4. Access levels</b> . . . . .	<b>32</b>
2.4.1. Ground Station Manager . . . . .	32
2.4.2. Satellite Operator . . . . .	33
2.4.3. Telemetry Expert . . . . .	33
<b>2.5. Requirements</b> . . . . .	<b>33</b>

<b>CHAPTER 3. Software design</b>	<b>37</b>
<b>3.1. Architecture</b>	<b>37</b>
3.1.1. Programming environment	38
3.1.2. Database Models	40
3.1.3. Interfaces	47
3.1.4. Date and time format	50
3.1.5. Accesibility	50
<b>3.2. Software structure</b>	<b>51</b>
3.2.1. API Implementation	53
3.2.2. TLE automatic update Implementation	58
3.2.3. Execution Manager Implementation	59
3.2.4. Inter-Process communication	66
<b>3.3. Graphical User Interface (GUI)</b>	<b>70</b>
<b>3.4. Deployment</b>	<b>77</b>
 <b>CHAPTER 4. Case study: <sup>3</sup>Cat-4</b>	 <b>79</b>
<b>4.1. Application layer</b>	<b>79</b>
4.1.1. Type Application layer packet	79
4.1.2. CRC32 (Cyclic Redundancy Check)	80
<b>4.2. Telecommand</b>	<b>81</b>
4.2.1. AES Encryption	82
<b>4.3. Telemetry</b>	<b>82</b>
4.3.1. POCKET+ algorithm	82
 <b>CHAPTER 5. Testing and verification</b>	 <b>85</b>
<b>5.1. Unit tests</b>	<b>85</b>
<b>5.2. Integration tests</b>	<b>85</b>
<b>5.3. Verification tests</b>	<b>86</b>
5.3.1. Orbit prediction and Tracking	87
5.3.2. Graphical User Interface (GUI) Testing: Flowgraph Editing and Execution	88
5.3.3. Data decoding verification	89
<b>5.4. Real case scenario</b>	<b>89</b>



**CHAPTER 6. Conclusions and future work . . . . . 91**

**Bibliography . . . . . 93**

**APPENDIX A. Operation Center Software API . . . . . 97**

**A.1. Authentication . . . . . 97**

**A.2. Scheduling endpoints . . . . . 97**

**A.3. Flowgraph-based TT&C endpoints . . . . . 102**

**A.4. Cap'n Proto Models . . . . . 105**



# LIST OF FIGURES

1.1	Standard CubeSats sizes [18]. . . . .	7
1.2	Keplerian elements describing a satellite orbit in the ECI coords [15]. . . . .	9
1.3	Coordinate Systems [11]. . . . .	11
1.4	Geodetic Coordinate System[14]. . . . .	12
1.5	ECI to ECEF conversion [14]. . . . .	13
1.6	Transformations between ENU and ECEF coordinates [13]. . . . .	14
1.7	Local coordinate frame showing the elevation (E) and azimuth (A). [13]. . . . .	15
2.1	Example implementation of a TT&C flowgraph. . . . .	23
2.2	Flowgraph mode configuration and assignment of flowgraph for each satellite pass based on mode preference. . . . .	28
3.1	Software architecture . . . . .	37
3.2	Scheduling Class diagram . . . . .	42
3.3	Flowgraph-based TT&C Class diagram . . . . .	45
3.4	HTTP vs WebSocket. . . . .	49
3.5	(1) The operator sends an update. (2) The server checks and stores the information. (3) The update is sent to the transmission thread. (4) The updates are sent to the operators. . . . .	57
3.6	Satellite thread task flow. . . . .	60
3.7	Main execution thread task flow. . . . .	61
3.8	Mission Block task flow. . . . .	62
3.9	Manual Mode task flow. . . . .	63
3.10	ZMQ thread task flow. . . . .	64
3.11	Backup thread task flow. . . . .	65
3.12	LOS thread task flow. . . . .	66
3.13	Exchange of Update notification messages and the resulting behavior between the Database thread and the Execution Manager . . . . .	67
3.14	Message exchange between the Operator WebSocket, Execution Manager, and Satellite thread. . . . .	68
3.15	Example exchange of ExecutionMsg during a flowgraph execution . . . . .	70
3.16	Login and SplashScreen Views . . . . .	71
3.17	Programmed passes View . . . . .	72
3.18	Scheduling modes . . . . .	72
3.19	Past passes View . . . . .	73
3.20	Satellites View . . . . .	73
3.21	Ground Stations View . . . . .	74
3.22	Flowgraph menu View . . . . .	74
3.23	Flowgraph Modes Configuration . . . . .	75
3.24	Flowgraph configuration View . . . . .	75
3.25	Flowgraph execution View . . . . .	76
3.26	Execution Logs Window . . . . .	76
4.1	App layer packet format . . . . .	79

4.2 Command packet format . . . . .	81
4.3 Compressed packet <a href="#">[24]</a> . . . . .	83
5.1 Unit Tests results. . . . .	86
5.2 Integration Tests results. . . . .	86
5.3 Ground Track verification ISS. . . . .	87
5.4 Tracking verification ISS. . . . .	88
5.5 Test Scenario. . . . .	89

# LIST OF TABLES

1.1 TLE format. . . . .	10
2.1 Backend Requirements . . . . .	34
2.2 Frontend Requirements . . . . .	35
5.1 Passes prediction ISS 10/07/2023. . . . .	88



# ACKNOWLEDGEMENTS

First and foremost, I would like to express my heartfelt gratitude to Adrián Pérez for his unwavering support and guidance throughout the entire duration of this project. Adrián's expertise and dedication have been invaluable in shaping the project's direction and ensuring its success. I am deeply grateful for his patience, encouragement, and insightful inputs that have greatly contributed to the final outcome.

I would also like to extend my appreciation to Albert Morea for his assistance during the initial stages of the project, and to Luis Contreras for his assistance during the testing process. Their willingness to share their knowledge and offer constructive feedback has been immensely helpful in refining the project.

In addition, I want to express my appreciation to all the individuals who are part of the UPC NanoSat Lab team for welcoming me with open arms and creating a magnificent work environment that allows students like myself to gain experience and knowledge in the space sector.

Lastly, I would like to acknowledge and thank my family and friends for their unwavering support and encouragement throughout this journey. Their belief in my abilities and their continuous motivation have been pivotal in keeping me focused and determined. I am truly grateful for their love, understanding, and sacrifices that have made this project possible.





# ACRONYMS

- AES** Advanced Encryption Standard. 81, 82
- AOS** Acquisition of Signal. 24, 44, 58–61, 65, 68, 88
- API** Application Programming Interface. 20, 38, 48, 49, 52, 53, 59, 66, 68, 71, 77, 85
- ECEF** Earth-Centered Earth-Fixed. 11–14, 55
- ECI** Earth-Centered Inertial. 10–12, 54, 55
- ENU** East, North, Up. 11, 14, 15
- EO** Earth Observation. 5, 7, 8, 15, 16, 31
- ESA** European Space Agency. 16, 82
- GS** Ground Station. 7–9, 11, 12, 14, 15, 17, 19–21, 23–27, 30, 32, 37, 42–44, 49, 51, 53–56, 59–61, 64, 69, 72, 74, 77, 79–82, 85, 87, 89–92
- GUI** Graphical User Interface. viii, 51, 52, 65, 68, 70, 71, 85, 88–90, 92
- HTML** HyperText Markup Language. 52
- HTTP** Hypertext Transfer Protocol. 48, 51, 53
- JWT** JSON Web Tokens. 50, 51
- LDAP** Lightweight Directory Access Protocol. 50, 51
- LLA** Latitude, Longitude, and Altitude. 11–13
- LOS** Loss of Signal. 24, 58, 60–65, 68, 88
- LXC** Linux Containers. 77
- NORAD** North American Aerospace Defense Command. 10, 20, 30, 44, 46, 54, 55, 59, 72–74, 87
- OpCen** Operation Center. 7, 8, 17, 19, 23, 28, 37, 39, 59, 64, 66, 79, 83, 86–92
- REST** Representational State Transfer. 48
- SDR** Software-defined radio. 89
- SGP4** Standard General Perturbations Satellite Orbit Model 4. 5, 10, 11, 54, 87
- SQL** Structured Query Language. 41
- TLE** Two-Line Element. 5, 10, 20, 21, 24, 39–41, 43, 54, 58–61, 85

**TOML** Tom's Obvious, Minimal Language. 55

**TPC** Transmission Control Protocol. 48

**TT&C** Telemetry, Tracking, and Command. 5, 6, 8, 19, 22, 37, 51, 54, 71, 72, 74, 79, 86, 89–91

**UHF** Ultra High Frequency. 17, 26

**UI** User Interface. 52

**URI** Uniform Resource Identifier. 48

**UTC** Coordinated Universal Time. 50, 53

**VHF** Very High Frequency. 17, 26

**WASM** WebAssembly. 70, 71

# INTRODUCTION

Since the beginning of humanity, an insatiable curiosity to explore the unknown has driven humanity to exceed the boundaries of what is deemed possible. This relentless pursuit began the space race and propelled incredible technological advancements, from the development of rockets to the creation of sophisticated satellites orbiting our planet. In recent years, a remarkable surge in satellite launches has been observed. This phenomenon is a result of various factors that have contributed to this growing trend.

Technological advancements in the space industry have played a significant role in driving the increase in satellite launches. Ongoing developments and improvements in satellite technology have enabled the creation of smaller, more efficient, and cost-effective designs. This has opened up new opportunities for organizations and companies to design, build, and launch their satellites.

The rising demand for space services has also been a key driver behind the increase in satellite launches. Telecommunications, Earth Observation (EO), satellite navigation, meteorology, scientific research, and other areas increasingly rely on satellite infrastructure to provide global services and connectivity.

However, the growing number of satellites presents challenges in terms of management and control. It is crucial to have a robust infrastructure that enables efficient monitoring, control, and coordination of these satellites. This involves real-time tracking of their position and status, establishing reliable two-way communications, and processing the large volumes of data generated by the satellites.

As humanity continues to push the boundaries of space exploration and satellite technology, it is essential to ensure the availability of the necessary infrastructure and capabilities to effectively manage this expanding fleet of satellites. This requires collaboration, innovation, and the development of advanced systems and technologies to support the future of space exploration and maximize the benefits that satellites can bring to society.

## Objectives

This final degree thesis focuses on the development of a mission-critical software system that revolutionizes the monitoring and automation of operations between the Operation Center and CubeSats. The main objective is to design an Operation Center software that integrates and manages various tasks handled by the operator during satellite missions. Acting as a central hub and orchestrator, the software will facilitate the coordination of one or more Ground Stations.

The Operation Center software will be designed to fulfill three fundamental tasks: scheduling satellite passes, providing Telemetry, Tracking, and Command (TT&C) capabilities, and enabling data analysis. By leveraging a tracking system based on the Standard General Perturbations Satellite Orbit Model 4 (SGP4) Orbit propagator and satellite's Two-Line Element (TLE), the software will allow operators to predict future passes of the satellites. Furthermore, it will enable the programming of communication with compatible Ground Stations.

In addition, the software will allow operators to configure the sequence of processes and messages to be exchanged during the missions through a user-friendly graphical interface. The interface will be a graph-based operations flowgraph, where each node will correspond to a telecommand, and responses will be analyzed to guide the flow toward

further actions. The software will be capable of generating telecommands, receiving and parsing responses and beacons from any of the implemented satellites, and establishing network socket connections with the Ground Stations.

Upon reception of the messages, the scientific data will be recollected for future analysis and the telemetry data obtained will be stored in an InfluxDB database. This will enable the visualization of telemetry data through the Grafana software.

## Methodology

This project has been carried out at the UPC NanoSat Lab and has had a duration of one year. In the development of this project, three phases have been identified. The first phase focused on learning the Rust programming language while also designing and planning the project. Simultaneously, an analysis of the libraries that should be used for the development was conducted. This established the foundations for the subsequent software development.

The next phase involved the actual implementation of the project, translating the design of the different modules into executable code, and creating a user-friendly graphical interface. In the final stage of the project, tests and verifications were performed to ensure the proper functioning of all software components and validate that they met the defined requirements. Any issues or bugs discovered during testing were carefully addressed and resolved to improve the stability and reliability of the software.

Throughout the project, regular weekly meetings were held to monitor progress, discuss encountered challenges, and make necessary adjustments. Additionally, version control using Git was employed to ensure the integrity of the codebase.

## Contents

This thesis report has been divided into six chapters. Chapter 1 serves as an introduction, providing the necessary background knowledge and context for the project. It offers an overview of satellite operations, describing the equipment used in the ground segment, the TT&C process, and the data processing associated with these operations.

The next two chapters considered the core of the project, focus on the development process. Chapter 2 outlines the scope of the project and presents the software requirements that need to be fulfilled. With a clear understanding of the project's needs, Chapter 3 delves into the software development process, discussing decision-making, employed technologies, and the overall development approach. It provides an overview of the steps and methodologies followed in the software development.

Once the software's functionality is developed and explained, Chapter 4 conducts a specific study for the <sup>3</sup>Cat-4 satellite. This study explores the considerations, characteristics, requirements, and challenges associated with the <sup>3</sup>Cat-4 satellite, to take into account to provide support with the developed software.

Chapter 5 focuses on the validation of the software's functionality and performance. It details the procedures and methods used to conduct verification tests and presents the results obtained. Additionally, it discusses the testing of the software in a real-world scenario.

The final chapter, Chapter 6, concludes the thesis by providing an overview of the project and highlighting potential areas for future improvements and further implementations.

# CHAPTER 1. STATE OF THE ART

## 1.1. Introduction to Satellites and CubeSats

During the last decades, the increasing number of satellites being launched for various purposes, such as communications, EO, navigation systems, and space exploration, has been mainly driven by the development of smaller satellites known as CubeSats. CubeSats are standardized, small-sized satellites that can be built, assembled, and launched in a relatively shorter period of time and at a lower cost compared to traditional larger satellites.

The CubeSat standard introduced a modular and compact design based on a standardized unit known as a 1U. A 1U CubeSat is a 10 cm cube typically weighing around 1 to 1.33 kg. This standardized form factor enables easy integration and compatibility across different CubeSat missions.

Since the inception of the CubeSat concept, larger sizes beyond the 1U have become increasingly popular. These larger sizes, such as the 1.5U, 2U, 3U, 6U, and 12U, have provided a scalable and versatile platform for satellite designers. Each size represents a multiplication of the 1U unit, allowing for greater volume and expanded capabilities [19].

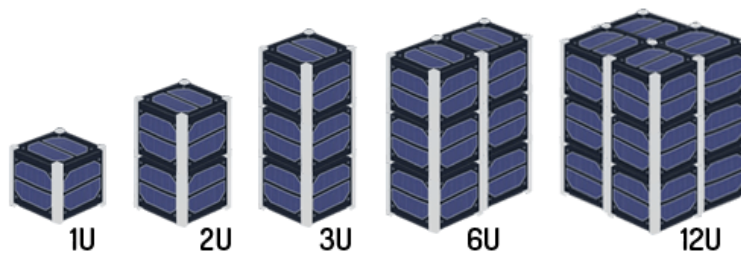


Figure 1.1: Standard CubeSats sizes [18].

## 1.2. Satellite operations

Satellite operations can be divided into two segments that work in coordination: the space segment and the ground segment.

The space segment refers to the satellite itself and its components that operate in the space environment. It operates autonomously or under the guidance of ground control commands, executing various functions to achieve the desired mission outcomes. These functions may include data collection, signal transmission, scientific measurements, or navigation services.

On the other hand, the ground segment contains the infrastructure and systems on Earth that support satellite operations. It includes a network of Ground Stations (GSs) strategically located around the globe, Operation Centers (OpCens), communication networks, and the associated software and hardware. Is responsible for managing and monitoring the satellite's activities, receiving and processing telemetry data, sending commands, and analyzing the data collected by the satellite.

### 1.2.1. Ground segment

The ground segment of satellite operations involves a range of equipment and systems that support the management, control, and monitoring of satellite activities. The primary components commonly found in the ground segment:

- **Ground Stations:** GSs are equipped with antennas and communication equipment, they serve as the direct link between the ground segment and satellites in orbit. Their primary function is to receive telemetry data transmitted by the satellite, including information on the satellite's health, status, and performance. In addition, are responsible for tracking satellites as they move across the sky, by precisely pointing their antennas toward the satellite's location, establish and maintain communication links during the satellite's orbit. Furthermore, transmit commands and instructions to the satellite, allowing for control and operation.
- **Ground Networks:** The ground segment relies on ground networks to establish connectivity between GSs, OpCens, and other facilities involved in satellite operations. These networks facilitate the transmission of data, commands, and instructions between different components of the ground segment and the satellite.
- **Operations Centers:** They act as the central hub for managing satellite operations. OpCens are equipped with sophisticated software systems, hardware infrastructure, and a team of operators. The operators are in charge of monitoring and controlling the satellite's activities, analyzing telemetry data, and making decisions regarding mission objectives and satellites.
- **Remote Terminals:** They serve as user interfaces that enable the retrieval of transmitted information for additional processing.

### 1.2.2. Telemetry, Tracking and Command

Satellite TT&C is an essential part of space missions, encompassing functions and processes for monitoring, controlling, and communicating with the satellite from a ground station or control center.

Telemetry is a key component of TT&C and involves collecting data from the satellite's on-board sensors, such as temperature, power levels, attitude, and orbit details. This telemetry data provides valuable insights into the satellite's health, status, operational parameters, scientific experiments, and EOs. It is transmitted to ground stations, allowing operators to monitor the satellite's performance, detect anomalies or deviations, and conduct data analysis for scientific purposes.

Command uplink is another crucial aspect of TT&C, allowing operators to send instructions and commands to the satellite. These commands can include configuring subsystems, executing specific operations, adjusting orbital parameters, or performing diagnostics. The command uplink ensures that the satellite operates according to mission objectives and can adapt to changing requirements.

Maintaining reliable communication between the GS and the satellite is essential throughout the mission. The TT&C subsystem enables bidirectional communication, facilitating the transmission of telemetry data from the satellite to the ground station and the reception of commands and instructions from the GS to the satellite. This communication link ensures effective monitoring, control, and coordination of the satellite's operations.

### 1.2.3. Satellite Orbit Analysis and Tracking

This section focus on studying satellite orbits and exploring the necessary tools and steps for accurate satellite tracking. This includes characterizing satellite trajectories, understanding orbital propagation algorithms, conversions between coordinate systems, and determining Look Angles for precise antenna alignment with the satellite. Which are essential aspects for establishing reliable communication between GS and satellites.

#### 1.2.3.1. Keplerian elements

An orbit is the trajectory that a satellite follows around a celestial body, such as the Earth. These trajectories are defined by Kepler's Laws that describe the motion of planets:

- *Law 1: All planets move in elliptical orbits with the Sun located at one focus.*
- *Law 2: The line connecting any planet to the Sun sweeps out equal areas in equal times.*
- *Law 3: The square of the period of any planet is proportional to the cube of the semi-major axis of its orbit.  $\Rightarrow T^2 = Ca^3$*

An orbit can be characterized based on the Keplerian elements  $(a, e, i, \Omega, \omega, v)$ . These elements are a set of parameters that describe the shape, size, and orientation of an orbit in space [16].

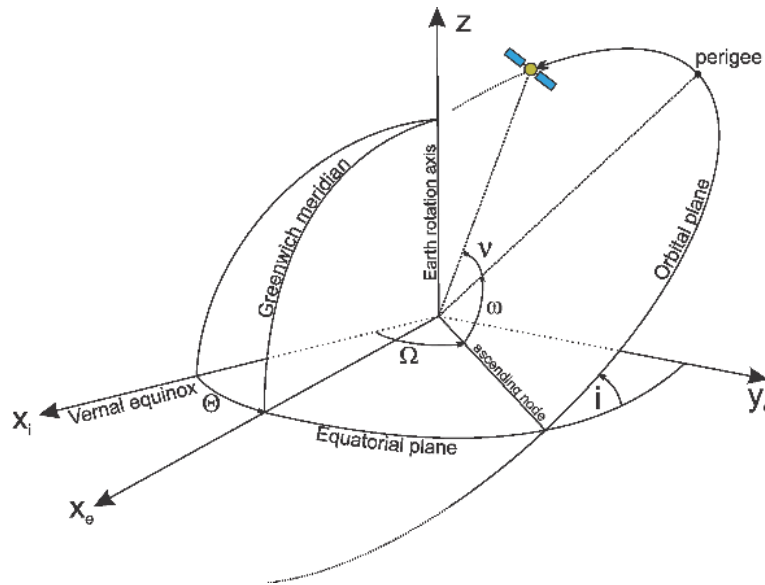


Figure 1.2: Keplerian elements describing a satellite orbit in the ECI coords [15].

The Keplerian elements include:

- $a$  : Semi-major axis, gives the size of the orbit.
- $e$  : Eccentricity, gives the shape of the orbit ( $0 \leq e < 1$ ).

- $i$  : Inclination angle, gives the angle of the orbit plane to the central body's equator.
- $\Omega$  : Right ascension of the ascending node, which gives the rotation of the orbit plane from reference axis.
- $\omega$  : Argument of perigee is the angle from the ascending nodes to perigee point, measured along the orbit in the direction of the satellites motion.
- $\nu$  : True anomaly gives the location of the satellite on the orbit.

### 1.2.3.2. Two-Line Element Set Format

The TLE is a data format used to represent the mean Keplerian orbital elements of Earth-orbiting objects. These elements are made available by North American Aerospace Defense Command (NORAD) and are commonly used for tracking and predicting the orbits of satellites and other space objects.

The mean values for each element are generated using the SGP4 orbital model, which is a mathematical algorithm used to predict the orbital state vectors of satellites relative to the Earth-Centered Inertial (ECI) coordinate system, based on its orbital elements, including parameters such as inclination, eccentricity, mean anomaly, argument of perigee, and mean motion. This model predicts the effect of perturbations caused by the Earth's shape, drag, radiation, and gravitation effects from other bodies such as the sun and moon [17].

The data of the NORAD TLE consists of three lines in the following format:

```

AAAAAAAAAAAAAAAAAAAAAAAAA
1 NNNNNU NNNNAAA NNNNN.NNNNNNNN +.NNNNNNNN +NNNNN-N +NNNNN-N N NNNNN
2 NNNNN NNN.NNNN NNN.NNNN NNNNNNN NNN.NNNN NNN.NNNN NN.NNNNNNNNNNNNNNN

```

Where Line 0 is a twenty-four character name, and Lines 1 and 2 correspond to the standard Two-Line Orbital Element Set Format. Figure 1.1 shows each line's contents in detail.

Columns	Contents	Columns	Contents
01-01	Line Number	01-01	Line number
03-07	Satellite Number	03-07	Satellite number
08-08	Classification	09-16	Inclination (Deg)
10-11	International Designator (Last two digits of launch year)	18-25	Right Ascension (Deg)
12-14	International Designator (Launch number of the year)	27-33	Eccentricity (decimal point assumed)
15-17	International Designator (Piece of the launch)	35-42	Argument of Perigee (Deg)
19-20	Epoch Year (Last two digits of year)	44-51	Mean Anomaly (Deg)
21-32	Epoch (Day of the year and fractional portion of the day)	53-63	Mean Motion (Revs per Day)
34-43	First Time Derivative of the Mean Motion divided by two	64-68	Revolution number at epoch (Revs)
45-52	Second Time Derivative of Mean Motion divided by six	69-69	Checksum (Modulo 10)
54-61	Drag term (decimal point assumed)		
63-63	The number 0 (Originally Ephemeris type)		
65-68	Element number		
69-69	Checksum (Modulo 10)		

Table 1.1: TLE format.



### 1.2.3.3. Orbital Coordinate Systems

Before delving into the topic of tracking satellites and pointing GS antennas towards them, it is important to consider the different coordinate systems that accurately define the position and orientation of objects in three-dimensional space. These coordinate systems include the ECI, the Earth-Centered Earth-Fixed (ECEF), the East, North, Up (ENU), and the Latitude, Longitude, and Altitude (LLA).

The **ECI** coordinate system is centered at the Earth's center, remains fixed in space, and does not rotate with the Earth. The  $z$ -axis runs along the Earth's rotational axis pointing North, the  $x$ -axis points in the direction of the vernal equinox (an imaginary line segment pointing from the center of the Earth towards the center of the Sun at the beginning of Spring of epoch J2000), and the  $y$ -axis completes the right-handed orthogonal system.

When using an orbital propagator like SGP4, the position obtained is typically represented in the ECI coordinate system. This coordinate system provides a fixed reference frame, allowing for an accurate representation of satellite positions in space.

The **ECEF** coordinate system, on the other hand, is fixed with respect to the Earth's surface. Its origin is at the Earth's center, and its axes are aligned with the Earth's rotation. In the ECEF system, the  $z$ -axis aligns with the Earth's rotational axis, pointing towards the North Pole, the  $x$ -axis points to the Prime Meridian (Greenwich Meridian), and the  $y$ -axis, as before, completes the orthogonal system.

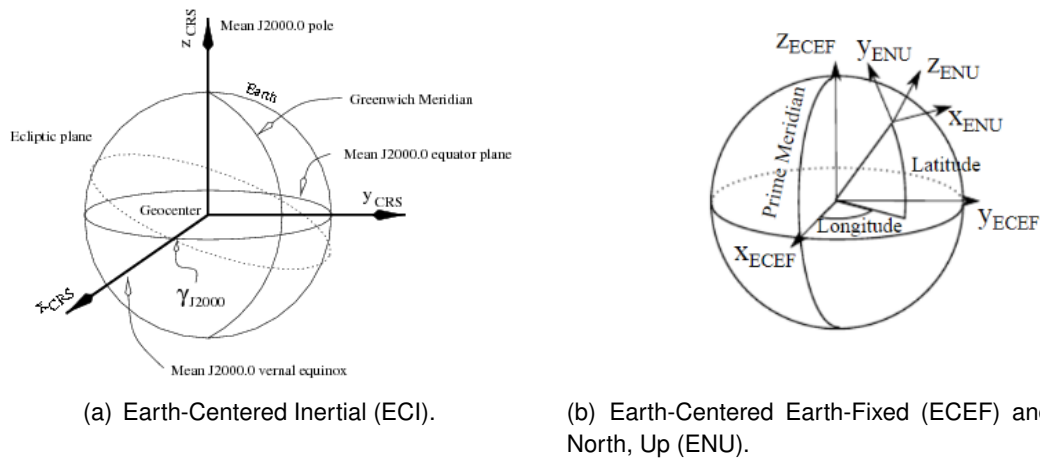
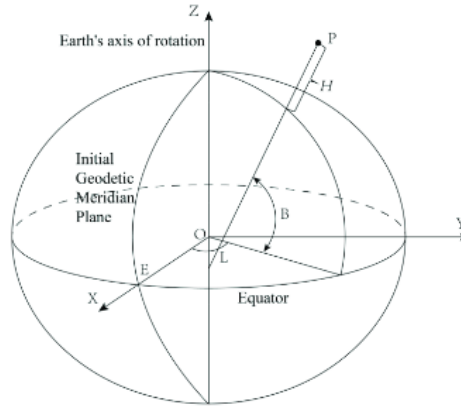


Figure 1.3: Coordinate Systems [11].

In the **ENU** coordinate system, the reference point serves as the origin, and the  $x$ -axis points East, the  $y$ -axis points North, and the  $z$ -axis points up perpendicular to the local tangent plane. The orientation of the axes is determined by the reference point and is aligned with the local horizontal and vertical directions.

Finally, the **LLA**, also known as geodetic coordinates, represents a position on the Earth's surface using latitude, longitude, and altitude (or elevation). LLA coordinates take into account the Earth's shape, specifically using an ellipsoidal model such as the WGS84 ellipsoid, which approximates the Earth's shape more accurately than a simple sphere. This coordinate system is the most commonly used in everyday navigation, maps, and geospatial applications, and it is the one used to describe the location of the GSs.



To accurately point toward the satellite during its trajectory, coordinate system conversions are necessary. As mentioned earlier, the satellite's position obtained from the orbital propagator is typically expressed in ECI coordinates. However, to determine the satellite's position relative to the Earth's surface, a conversion from ECI to ECEF coordinates is necessary. This conversion takes into account the Earth's rotation and aligns the coordinate system with the Earth's surface.

Simultaneously, LLA coordinates of the GS need to be converted to ECEF coordinates. Once both the satellite and GS positions are represented in the same coordinate system, the next step is to calculate the azimuth and elevation angles, commonly referred to as Look Angles. The Look Angles represent the direction in which the ground station antenna should be pointed to establish a line-of-sight connection with the satellite.

To make the conversion from ECI to ECEF we need to take into account that both have approximately the same origin and  $z$ -axis and only differ by an angular component on the  $xy$  plane. This angular difference can be used to rotate a vector from one frame to the other.

This angular component, also known as Greenwich Sidereal Time, can be computed as:

$$\theta_g(T) = \theta_g(0^h) + \omega_e \cdot \Delta t \quad (1.1)$$

where  $\Delta t$  is the UTC time of interest,  $\omega_e = 7.2921151010^{-5} rad/s$  is the Earth's rotation rate, and  $\theta_g(0^h)$  the Greenwich sidereal time at 0h (midnight) UTC, obtained from:

$$\theta_g(0^h) = 24110.54841 + 8640184.812866 \cdot T + 0.093104 \cdot T^2 - 0.0000062 \cdot T^3 \quad (1.2)$$

$$T = d/36525 \quad (1.3)$$

$$d = JD - 2451545.0 \quad (1.4)$$

where  $T$  is in Julian centuries from 2000 Jan. 1 12h UT1 and  $d$  is the number of days of Universal Time elapsed since JD 2451545.0 (2000 January 1, 12h UT1) [10].

To transform between two Cartesian coordinate systems, a common approach is to perform axis rotations. By rotating the axes along one of the axes, the transformation can be achieved. The rotation matrices are [12]:

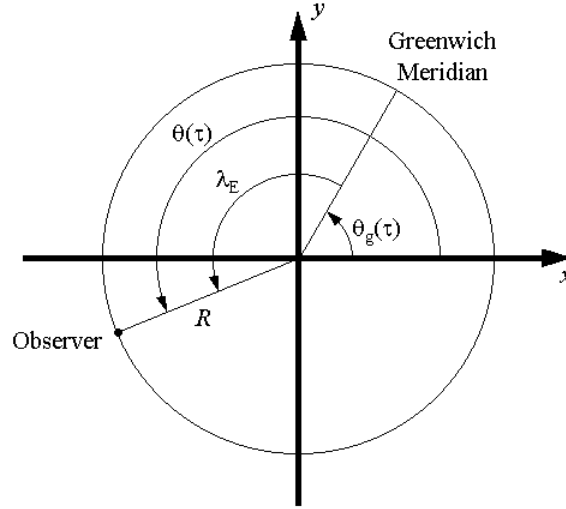


Figure 1.5: ECI to ECEF conversion [14].

$$R_1[\theta] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{bmatrix}; R_2[\theta] = \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix}; R_3[\theta] = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (1.5)$$

Imposing the condition that  $z_{ECI} = z_{ECEF}$ , obtained is:

$$\begin{bmatrix} x_{ECEF} \\ y_{ECEF} \\ z_{ECEF} \end{bmatrix} = R_3[\theta] \begin{bmatrix} x_{ECI} \\ y_{ECI} \\ z_{ECI} \end{bmatrix} \quad (1.6)$$

and finally, the conversion is obtained as:

$$x_{ECEF} = x_{ECI} \cdot \cos \theta + y_{ECI} \cdot \sin \theta \quad (1.7)$$

$$y_{ECEF} = -x_{ECI} \cdot \sin \theta + y_{ECI} \cdot \cos \theta \quad (1.8)$$

$$z_{ECEF} = z_{ECI} \quad (1.9)$$

The conversion between the LLA coordinates of an object to ECEF Cartesian using the ellipsoid WGS84 can be done using the following expressions:

$$x_{ECEF} = \left( \frac{a}{\sqrt{1 - e^2 \sin^2 \phi}} + h \right) \cos \phi \cdot \cos \lambda \quad (1.10)$$

$$y_{ECEF} = \left( \frac{a}{\sqrt{1 - e^2 \sin^2 \phi}} + h \right) \cos \phi \cdot \sin \lambda \quad (1.11)$$

$$z_{ECEF} = \left( \frac{a(1 - e^2)}{\sqrt{1 - e^2 \sin^2 \phi}} + h \right) \sin \phi \quad (1.12)$$

where  $\lambda$  means geodetic longitude,  $\phi$  means geodetic latitude,  $h$  means the height above the ellipsoid and  $a, e$  are the ellipsoid parameters.

Now that both the satellite and the GS are represented in the same coordinate system, it is possible to obtain the vector that points from the GS position to the satellite. This can be achieved by subtracting the vectors of the GS and satellite ECEF coordinates.

$$\bar{V} = \bar{Sat} - \bar{GS} \quad (1.13)$$

To determine the relative position of the satellite with respect to a GS, the GS is considered as the origin of the ENU coordinate system. This coordinate system allows us to calculate the position of the satellite relative to the GS using the ENU axes.

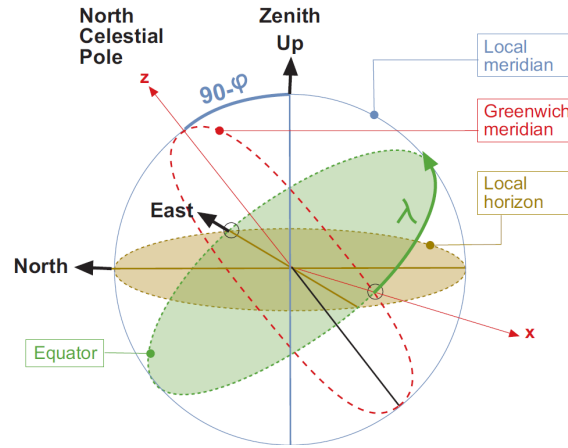


Figure 1.6: Transformations between ENU and ECEF coordinates [13].

This can be done by two rotations, where  $\phi$  and  $\lambda$  are the latitude and longitude of the ellipsoid, respectively. First, anti-clockwise rotation over east-axis by an angle  $90 - \phi$  to align the up-axis with the  $z$ -axis, and then anti-clockwise rotation over the  $z$ -axis by an angle  $90 + \lambda$  to align the east-axis with the  $x$ -axis. That is:

$$\begin{bmatrix} E \\ N \\ U \end{bmatrix} = R_1[(\pi/2 - \phi)]R_3[(\pi/2 + \lambda)] \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (1.14)$$

where the rotation matrix yields:

$$R_1[(\pi/2 - \phi)]R_3[(\pi/2 + \lambda)] \begin{bmatrix} -\sin\lambda & \cos\lambda & 0 \\ -\cos\lambda\sin\phi & -\sin\lambda\sin\phi & \cos\phi \\ \cos\lambda\cos\phi & \sin\lambda\cos\phi & \sin\phi \end{bmatrix} \quad (1.15)$$

The vector computed in the ECEF coordinates can be expressed in the ENU coordinates as:

$$\bar{x} = (-\sin\lambda, -\cos\lambda\sin\phi, \cos\lambda\cos\phi) \quad (1.16)$$

$$\bar{y} = (\cos\lambda, -\sin\lambda\sin\phi, \sin\lambda\cos\phi) \quad (1.17)$$

$$\bar{z} = (0, \cos\phi, \sin\phi) \quad (1.18)$$

Finally, the elevation and azimuth (Look Angles) of the GS antenna, and the slant range can be computed easily from the ENU Cartesian coordinates as:

$$d = \sqrt{E^2 + N^2 + U^2} \quad (1.19)$$

$$el = \arcsin\left(\frac{U}{d}\right) \quad (1.20)$$

$$az = \arctan\left(\frac{E}{N}\right) \quad (1.21)$$

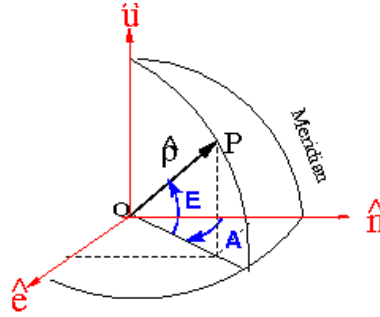


Figure 1.7: Local coordinate frame showing the elevation (E) and azimuth (A). [13].

A satellite is considered visible for the GS when the elevation angle is positive. In order to establish a communication link, the antenna of the GS needs to be positioned based on the Look Angles ensuring that it is in line of sight with the satellite, allowing for successful communication.

### 1.3. NanoSat Laboratory

The Nano-Satellite and Payload Laboratory (UPC NanoSat Lab) is a multidisciplinary research facility located at the Technical University of Catalonia - UPC Barcelona Tech (Campus Nord). It is dedicated to the design, development, and exploration of nanosatellites, with a particular focus on innovative small spacecraft system concepts and EO payloads.

The lab is fully equipped with all the necessary tools for soldering and assembling different boards and subsystems. It boasts advanced features such as Helmholtz coils and an air pad, which are utilized for precise testing and calibration of the attitude control system. Moreover, the lab has been designed to carry out environmental qualification tests (vibrations, and vacuum and thermal cycling), in a clean environment (Class 8 clean room) for the integration of payloads, subsystems, and small satellites.

The lab promotes a collaborative environment among students from diverse academic backgrounds, enabling them to work on missions and gain experience in nanosatellite research and engineering. This collaborative environment promotes knowledge sharing and offers valuable opportunities for learning and participation in real missions.

### 1.3.1. Missions

The UPC NanoSat Lab has participated in a variety of satellite missions, including:

- **<sup>3</sup>Cat-1:** The <sup>3</sup>Cat-1 satellite was the first satellite developed by the UPC NanoSat Lab and the first in Catalonia. This 1U satellite integrates seven different payloads, enabling a wide range of scientific research and experimentation. These payloads include the Eternal self-powered beacon, CellSat Solar Cells, MEMS-based monoatomic oxygen detector, Graphene Transistor in-space characterization, study of plasma effects in Wireless Power Transfer links, low-resolution CMOS camera, and geiger counter. Each payload serves a specific purpose, such as generating power, assessing solar cell performance, detecting monoatomic oxygen, studying graphene transistors, exploring plasma effects, capturing images, and measuring ionized particles and radiation dosimetry. The <sup>3</sup>Cat-1 satellite was launched in November 2018 [20].
- **<sup>3</sup>Cat-2:** The <sup>3</sup>Cat-2 satellite, launched in August 2016, is a 6U CubeSat, is equipped with a GNSS-R payload for EO. It carries four payloads, including the PYCARO GNSS-R main payload for reflectometry, the Mirabilis star tracker for validation purposes, the IEEC AMR eLISA magnetometer for behavior analysis, and the FAPEC compression algorithm for data compression. These payloads enable altitude mapping, wind analysis, magnetometer validation, and efficient data compression for scientific missions [20].
- **<sup>3</sup>Cat-4:** The <sup>3</sup>Cat-4 is a 1U satellite designed to demonstrate the capabilities of nano-satellites for EO using GNSS-R and microwave radiometry, as well as for Automatic Identification Services (AIS). The mission includes scientific experiments focused on assessing GNSS-R observables, studying ionospheric corrections, evaluating GNSS-R applications over land surfaces, assessing RFI detection and mitigation techniques, creating RFI maps, and validating the design of an AIS receiver. The <sup>3</sup>Cat-4 mission is part of the "Fly Your Satellite!" program of the European Space Agency (ESA) Academy [20].
- **<sup>3</sup>Cat-5:** The FSSCat mission, consisting of two federated 6U Cubesats (<sup>3</sup>Cat-5/A and <sup>3</sup>Cat-5/B), in support of the Copernicus Land and Marine Environment services. These CubeSats, carry a dual microwave payload (GNSS-Reflectometer and L-band radiometer) and a multi-spectral optical payload for measuring soil moisture, ice extent, ice thickness, and detecting melting ponds over ice. The mission also includes an Optical Inter-Satellite Link (OISL) technology demonstrator and a proof-of-concept for a Federated Satellite System (FSS). The FSSCat mission was launched in September 2020 [20].
- **<sup>3</sup>Cat-6:** The RITA is a 1U payload of the AlainSat-1 CubeSat, dedicated to studying global warming and vegetation. It includes various instruments such as an L-Band Microwave Radiometry for soil humidity analysis, a hyperspectral camera for vegetation measurements, and RFI detection capabilities. To optimize its operations, the payload is designed with different operational modes that have specific power and data budgets. This allows flexibility in adapting to scientific interests, power availability, and ground station contact.

### 1.3.2. Ground segment

Currently, the laboratory operates and maintains a GS located at the Montsec Observatory facilities managed by the Institut d'Estudis Espacials de Catalunya (IEEC)<sup>1</sup>. This GS is equipped with S-Band, Ultra High Frequency (UHF), and Very High Frequency (VHF) antennas. However, there is a lack of an OpCen able to centralize all the satellite operations and GSs management tasks.

This project aims to develop an OpCen software able to supply all the deficiencies, by providing the necessary tools and functionalities for planning, scheduling, monitoring, and controlling satellite operations, which will enable operators to efficiently supervise and coordinate the various tasks, such as satellite communication, data storage, telemetry analysis, and mission planning.

---

<sup>1</sup>More information can be found at: <https://montsec.ieec.cat/>





## CHAPTER 2. CONCEPTUAL DESIGN

In this chapter, the reasons behind the development of this software will be explained, along with the necessary requirements it must fulfill and the overall scope of the project. There are three distinct needs that this software aims to address: Scheduling, TT&C, and Data Downlink and Storage. Each of these needs is intended to support the different tasks performed by operators throughout the satellite operation process. For convenience, each of them will be referred to as:

- Scheduling module
- TT&C module
- Data Downlink and Storage module

The satellite operators at UPC NanoSat Lab require a reliable and efficient tool to effectively manage and coordinate the commanding and downlink of the satellites through the GSs and OpCen. To address these challenges, a centralized software is necessary to provide operators with a comprehensive overview of all satellites under their control, allowing them to track and monitor each satellite, ensuring optimal resource utilization and streamlined coordination across various missions, making better decisions, prioritizing tasks, and allocating resources strategically.

### 2.1. Ground Segment management

Without a robust scheduling mechanism, the operators face significant challenges in maintaining centralized control, leading to potential inefficiencies, operational difficulties, and the risk of errors.

One of the key advantages of implementing a scheduling system is its ability to predict and determine when satellites will be visible from specific GSs. By leveraging orbital data, satellite orbits, and the Earth's rotation, operators can identify optimal communication windows, facilitating efficient planning and scheduling of communication sessions. This ensures a reliable and uninterrupted data exchange between satellites and GSs.

Furthermore, a scheduling system will facilitate the management of the GSs, by leveraging knowledge of their operational status, and indicating whether they are available and ready for communication with the satellites or under maintenance. This information will assist in selecting the most suitable GS, based on factors such as availability, operational frequencies, and bandwidth, thereby optimizing overall communication performance within a network of GSs.

In previous works done by the team of UPC NanoSat Lab, this topic has been addressed [9]. By designing an OpCen that centralizes data and enables the control of multiple GSs. This centralized approach allows for efficient management, analysis, and control of satellite operations. Operators can seamlessly monitor and communicate with satellites, and automated scheduling capabilities optimize GS utilization. That is why this new project is presented as an evolution of the previous work, providing greater scope and utility to the system.

### 2.1.1. Satellite management

The scheduling module should support the management and scheduling of multiple satellites. Operators should be able to easily add, remove, and modify satellite information. This includes assigning operational frequencies of satellites, allowing for the selection of the appropriate GS during scheduling. Additionally, it should allow for the modification of the NORAD ID as it may be assigned shortly after the satellite's deployment into orbit. The NORAD ID serves as a unique identifier for the satellite and is crucial for tracking and communication purposes. Thus, updating the NORAD ID is necessary to ensure accurate and coherent information.

Furthermore, during the initial stage after launch, predicting the precise orbit of a satellite can be challenging due to the difficulty of distinguishing individual nanosatellites from other cubesats during the early stages when they are launched together. Additionally, the lack of precision in the injection vector provided by the launcher makes it difficult to determine the precise position and velocity of the orbit. As a result, the software used for tracking and orbit prediction should allow for manual input of TLE data by operators. This ensures that the most up-to-date and accurate information is used for orbital predictions. Moreover, the software should have the capability to automatically update the TLE data periodically using the Celestrak<sup>1</sup> Application Programming Interface (API), which provides reliable and current orbital information.

### 2.1.2. Ground Station management

In addition to controlling the different satellites, the software should also enable the management of GSs to have control over both sides of the communication. The software should provide information about the available GSs, including their locations, operational frequencies, and current operational status. This information will be crucial in determining the feasibility of communication with the satellite.

### 2.1.3. Scheduling module

The scheduling module unifies satellite management and GS management, intending to efficiently plan and schedule communications with the satellites based on their specific requirements and specifications. By integrating satellite management and GS management, the scheduling module enables seamless coordination of resources and tasks. It takes into account various factors such as satellite availability, GS capabilities, communication windows, and satellite-specific requirements.

The scheduling module leverages the specifications provided by the satellites to choose the most compatible GSs taking into consideration the frequency bands and bandwidth of the satellites, ensuring that the communication sessions are planned in a way that meets the specific needs of each satellite. Furthermore, the scheduling module uses the TLE to compute the satellite orbits to determine when the communication can take place.

---

<sup>1</sup>Celestrak: <https://celestrak.org/>

### 2.1.3.1. *Scheduling options*

The software should offer a variety of options to accommodate diverse scheduling needs. Operators will have the flexibility to choose from different scheduling modes based on their specific requirements. These options include:

- **One Pass:** This option schedules the next available pass for a satellite starting from the current time. It allows operators to quickly schedule a single communication session for immediate operations.
- **Multiple Passes:** Operators can specify the number of consecutive passes they want to schedule for a satellite, and the software will search and schedule the nearest passes that will happen.
- **Multiple Passes from Date:** With this option, operators can schedule a specific number of passes starting from a chosen date. This option is useful for planning future communication sessions in advance.
- **All Passes Until Date:** Operators can schedule all available passes for a satellite until a specified future date.
- **All Passes Within Time Interval:** This option allows operators to define a specific time interval and schedule all the passes that will occur within that period.

When making these predictions, several factors must be considered, as they can affect both the performance and accuracy of the software. The predictions are based on the orbital parameters extracted from the TLE of the satellites. Therefore, if the TLE data is not up-to-date, the predictions may not be entirely accurate. Additionally, when scheduling a pass far in advance, the predicted results may not be entirely precise.

Furthermore, the calculation follows an iterative process to identify the moments of acquisition of sight and loss of signal events. If the search period is not properly defined, it could potentially cause the software to become stuck in an infinite loop, in the cases that the desired event may not occur. To prevent this, a time limit is imposed for the first three cases, mentioned above, where there is no specific end date for the search. This time limit can either be a default value set by the software or directly inputted by the operator.

Another factor to consider is what to do when two passes of different satellites occur at the same time and require the use of the same GS. In such cases, if a conflict is found with a previously scheduled pass, the operator will be notified of the conflict, and the scheduling software will prevent the conflicting pass from being scheduled, avoiding overlaps or clashes in the planned passes. By notifying the operator about the conflict, they will have the opportunity to review and adjust the scheduling accordingly to ensure the desired execution.

Finally, all the predictions will be displayed in a graphical interface, providing essential information for the operator. This includes details such as the satellite, GS, start and end time of the pass, maximum elevation, and other relevant information. The interface will allow operators to easily visualize and evaluate the scheduled passes, being able to eliminate those that do not meet the needs or do not want to be carried out. Additionally, the interface will also display the completed passes, allowing operators to review the historical data and performance of the satellite operations.

### 2.1.3.2. *External Accessibility*

Furthermore, an additional application for this software is to promote collaboration and facilitate resource sharing among external operators and organizations. This will enable them to utilize the infrastructure available at the UPC NanoSat Lab.

By providing access to this software, external users can plan and schedule their satellite passes, ensuring the availability of the necessary resources for their specific missions and operations, and enabling them to make the most of the deployed infrastructure at the UPC NanoSat Lab.

In summary, the scheduling module of the software serves both operational and commercial purposes. It facilitates the scheduling of passes for the satellites, ensuring efficient resource utilization. Once the passes are completed, all relevant information is recorded, allowing for the billing of services rendered. This capability enables the UPC Nanosat Lab to track and charge for the utilization of its resources.

## 2.2. **Flowgraph-based TT&C**

Nowadays, the tasks of TT&C in satellite operations are carried out manually by operators at UPC NanoSat Lab, and even though the procedures to be followed during operations are defined in detail, these introduce the possibility of human errors, which may impact the satellite's performance. The TT&C operators are responsible for collecting telemetry data, commanding its operations, maintaining its orbit, detecting anomalies, managing communication links, and monitoring the satellite's health. These tasks are crucial for ensuring the successful operation and management of satellites throughout their lifetimes.

There is a strong need for automated support that simplifies operations and provides comprehensive assistance and guidance to the operators. By implementing advanced automation and intelligent software, this process would effectively reduce the workload and greatly enhance the efficiency of TT&C tasks, leading to smoother and more efficient satellite operations.

### 2.2.1. **Flowgraph**

A flowgraph defines a set of processes and operations to be performed by software. These tasks are grouped into independent components and connected following a predefined logic. This modular approach promotes code reusability, maintainability, and scalability, as components can be easily added, removed, or modified without impacting the entire system.

The goal is to provide a flexible and adaptable design tool for different missions. Operators can easily modify or extend flowgraphs to accommodate changes in mission requirements or operational scenarios. This allows for quick adjustments of components, making it easier to build complex systems from smaller, reusable blocks.

Furthermore, flowgraphs enable automated decision-making by incorporating logic and algorithms within the design of blocks. The flowgraph analyzes data and makes real-time decisions, reducing the need for manual intervention and improving the overall reliability

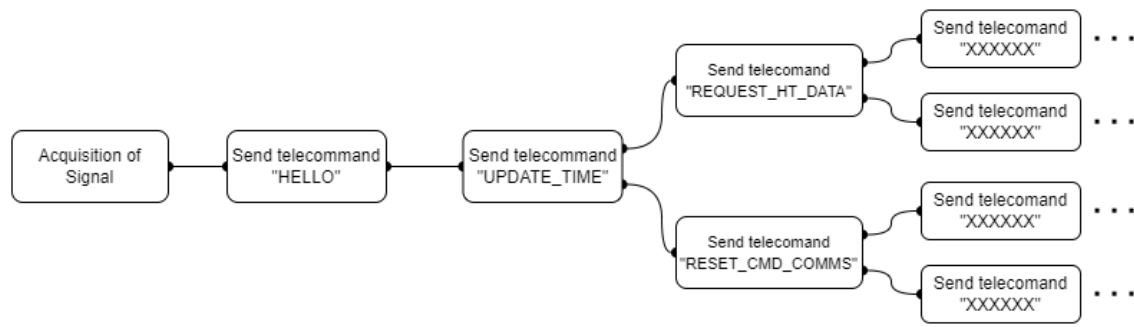


Figure 2.1: Example implementation of a TT&C flowgraph.

and efficiency of satellite operations. Additionally, flowgraphs offer a visual representation of the system's structure and the flow of data between different components.

For those reasons, the integration of flowgraph technology covers all the objectives, as it provides a powerful tool for visualizing, organizing, and managing the complex flow of the data, commands, and processes involved in satellite operations. This provides a clear and intuitive representation of the entire process, enables operators to visually design and configure the sequence of operations, define dependencies between tasks, monitor the flow of data and commands, and make informed decisions in real time.

When designing a flowgraph, it is important to consider that each satellite mission has its own unique objectives, goals, and specific requirements. These missions utilize different protocols and communication systems, making it impractical to reuse the same code in the blocks for every mission. Therefore, a personalized flowgraph is necessary for each mission to accommodate these variations and ensure optimal performance and compatibility.

In this context, three different types of blocks have been distinguished based on their functionalities. These include utility blocks, mission blocks, and background blocks.

- **Utility blocks:** These blocks are designed to be versatile and applicable to various missions. They offer high-level control and functionality, performing generic operations that assist in managing and controlling the flowgraph. Their purpose is to provide a consistent and standardized set of operations that can be used across different satellites, promoting reusability and reducing the need for mission-specific customization.
- **Mission blocks:** These blocks are specific design for each mission and are responsible for the exchange of information between the satellite and the OpCen. They incorporate mission communication protocols, commands, data structures, and other specific functionalities. Also, integrate some logic to take decisions about the next block to be executed. These blocks share the same functionality and visual representation across all the satellites and only vary their execution based on the specifications of the missions.
- **Background blocks:** These blocks handle background processes and tasks that support the overall operation of the satellite. They provide essential functions such as connecting to the GS, managing database storage, and performing auxiliary tasks. While these blocks may not directly contribute to the primary mission objectives, they play a crucial role in ensuring the correct functioning of the satellite's operations.

#### 2.2.1.1. *Utility blocks*

Within this group, there are two blocks that have very similar functionalities: the *Acquisition of Signal (AOS)* and *Loss of Signal (LOS)* blocks. These blocks employ orbital propagation algorithms to predict and calculate various parameters related to satellite passes over a GS, as well as to detect and manage the satellite's visibility to the GS.

The AOS block is responsible for monitoring the satellite's position and determining the moment when it is about to pass over the GSs, preparing for the initiation of communication. By utilizing parameters such as the latitude, longitude, and altitude of the GSs, along with the satellite's TLE data, the AOS block calculates the Look angles necessary for establishing communication.

Once the satellite's elevation angle exceeds a predefined threshold, the LOS block takes over. This block continuously tracks the satellite to determine when it is no longer visible to the GSs, marking the end of the communication.

Additionally, another block, known as the *Tracking block*, can be found in this group. The Tracking block is responsible for computing the necessary antenna movements required to accurately point the antenna toward the satellite and track its path during the pass. Its main function is to generate a file containing the calculated movements and send it to the GSs to follow these instructions.

#### 2.2.1.2. *Mission blocks*

Mission blocks are responsible for generating and receiving data that is transmitted to and from the satellite. These blocks have a unique and customized implementation specifically for the mission they are designed for. This means that each set of mission blocks is executed by a different code to adapt to the communication requirements and protocols of the individual satellite missions. By utilizing mission-specific code, these blocks can efficiently handle data generation, reception, and processing following the unique characteristics and objectives of each mission.

However, it is important to highlight that there is an ongoing effort to standardize the operation of satellites in the future. This initiative aims to eliminate the need for such distinction among mission blocks, as a unified and standardized approach would be adopted across multiple satellites. This will greatly facilitate the operation of new satellites, as they will be compatible with the standardized software and protocols, promoting enhanced interoperability and reducing complexity.

During operations, mission blocks take input data and generate a serialized message for transmission through the GSs interface. They then await a response, which is subsequently decoded and analyzed to determine the block's success. These blocks also incorporate control parameters for communication, such as the timeout time. The timeout defines the duration within which a block considers a package lost if no response is received. The adjustability of the timeout is crucial since the processing speed of commands by the satellite may vary, resulting in different RTT (Round Trip Time) for each case. Additionally, mission blocks include the number of retry attempts for package transmission in case of a timeout or other unsuccessful execution.

Based on the received response or the inability to execute the block successfully due to a timeout or other events, mission blocks determine their output, enabling further decision-making and processing within the flowgraph.

The mission blocks can be categorized into three types, as a function of the input parameters:

- **Send Command:** This block takes a command as input. It generates a data package containing the command to be transmitted to the satellite through the GSs interface.
- **Send File Command:** This block takes both a command and a file as input. The command represents instructions, while the file contains the actual content of the package to be sent.
- **Send Bytes Command:** This block directly accepts a sequence of bytes as input and is utilized in scenarios where no specific data structure has been defined for a particular case. It allows for the transmission of raw bytes as the content of the data package, providing flexibility in handling diverse data formats or situations that do not adhere to predefined structures.

In addition, to provide flexibility for handling various execution scenarios within the mission blocks and providing appropriate output paths depending on the outcome of the block's operation can be differentiated another three different cases:

- **Simple Output:** In this case regardless of the execution result, there is only one possible output.
- **Success-Fail Conditional Output:** In this scenario, the output is determined based on the success or failure of the block's execution. If the execution is considered successful, the Success output is chosen; otherwise, the Fail output is selected.
- **Success-Fail-Timeout Conditional Output:** In this case, if a failure occurs, the selection of the output varies depending on the origin of the failure, which can be attributed to either a timeout or another reason.

Combining both groups, the software supports up to nine different combinations, providing operators with a high degree of flexibility and customization. By linking the blocks together, operators can define the sequence of operations and the data flow within the flowgraph, allowing the coordination and integration of various tasks and ensuring that the execution follows the desired workflow.

#### 2.2.1.3. Background blocks

Background blocks, similar to utility blocks, are executed in the same way regardless of the mission. They are designed to perform complementary tasks to the execution of the flowgraph, providing support in various areas such as data querying and storage in databases and files, as well as establishing connections with the GSs.

The *Ground Station block* acts as a bridge between the flowgraph execution and the GSs. Its primary role is to establish and maintain a connection with the GSs throughout the communication process. Moreover, it should enable GSs roaming, which allows for the flexibility to switch to a different communication GSs during satellite operations. This capability

ensures uninterrupted communication by facilitating a smooth transition to an alternative GSs location if required to maintain the satellite connection. Once the communication is completed, it frees up the channel to allow the next communication session. Currently, the UPC NanoSat Lab operates two antennas in Montsec (including one for UHF and another for S-Band) and an additional two antennas in Barcelona (UHF and VHF). These antennas have a standardized data exchange interface, making them interoperable within the software, which will be explained in detail later.

During the flowgraph execution, the GS block receives packets generated by the mission blocks and sends them to the GSs for transmission to the satellite. Simultaneously, it receives packets sent by the satellite, allowing them to be processed by the mission blocks, facilitating bidirectional communication between the satellite and the software.

The *Database Connection block* provides the functionality to interactively query the configuration of the executing flowgraph. As the execution progresses, the flowgraph can dynamically retrieve information such as which block to execute at each step, the specific command to be sent, the data associated with the command, and any other relevant details necessary for the proper functioning of the execution.

The *Backup block* has the functionality of collecting all the information generated during the execution of the flowgraph. This includes data generated from packet reception and transmission to the satellite, as well as the various decisions and paths taken during the flowgraph execution.

The background block serves two main purposes. Firstly, it serves as a record of everything that has happened during the communication for later analysis and error detection. Secondly, the Background block is used to support the visual representation of the execution. When an operator joins the execution midway, it is essential to provide them with a complete overview of the previous events. By utilizing the information collected by the Background block, the operator can access to the complete representation of the execution history. This ensures that the operator has the necessary context and can make informed decisions based on the entire flowgraph execution, even if they didn't observe it from the beginning.

Finally, there is one last block directly related to Data Downlink and Storage, which is explained in depth in Section 2.3.. This block is the *Data Storage block*, responsible for storing all telemetry data received during the execution. The stored data can be visualized by the operator using Grafana, allowing for monitoring and analysis of the satellite's performance and health, to take decisions during the processes.

Finally, there is one last block directly related to Data Downlink and Storage, which is explained in depth in Section 2.3.. This block is the *Data Storage block*, responsible for storing all telemetry data received during the execution. All the data is sent and saved into a telemetry database, enabling the operator to visualize the stored data using Grafana. This functionality allows for monitoring and analysis of the satellite's performance and health, aiding in informed decision-making during the processes. However, the detailed discussion of the telemetry database falls beyond the scope of this work.



### 2.2.2. Flowgraph edition modes

At this point, the advantages that a modular block design provides when it comes to developing the chain of actions and commands that must be executed in a flowgraph are already known. One of these advantages is undoubtedly the flexibility that it offers and the adaptability to any situation.

However, it is essential to consider how the satellite pass will be connected to the flowgraph that must be executed. The choice of a flowgraph can be conditioned by many factors such as the use of a particular GSs, the time in which the contact with the satellite will occur, or the objectives of the communication.

In this manner, the introduction of different operating modes to the flowgraph enhances another level of customization and flexibility for a given scenario. As a result, four different selection criteria can be defined:

- **Personalized pass:** This mode enables the design of a chain of blocks to be executed by explicitly selected passes (one or more), allowing the design of personalized communications for specific scenarios.
- **Programmed:** In specific scenarios, it is possible that depending on what time the contact with the satellite occurs, the operator is not available. This case could be, for example, during nighttime hours when autonomous execution of the flowgraph becomes necessary. During the configuration, it is essential to specify the start and end time of the prioritized flowgraph selection period.
- **Default Ground Station:** Another criterion to consider in the flowgraph design can be the selection of the GSs responsible for the transmission. Having different flowgraphs, allows customization and adaptation to the specific capabilities of each GSs, optimizing the communication parameters and protocols to maximize performance. In this case, the operator should indicate their preferred GSs for the execution.
- **Default flowgraph:** However, if none of the previously configured modes in a satellite are compatible with the pass, the default design will be selected. It is important to note that unlike the other modes, which may or may not be present, there should always be only one default model.

Before the satellite contact, the server examines the configured flowgraphs for that particular satellite to determine the appropriate flowgraph. Each mode is assigned a preference based on its exclusivity and restrictions. The priority order, from highest to lowest, is as follows: Personalized Pass, Programmed, Default GSs, and Default. This ensures that the server selects the most suitable flowgraph for the given scenario, taking into account the specific requirements and constraints of each mode. An example of the configuration and assignment of these modes can be found in figure 2.2.

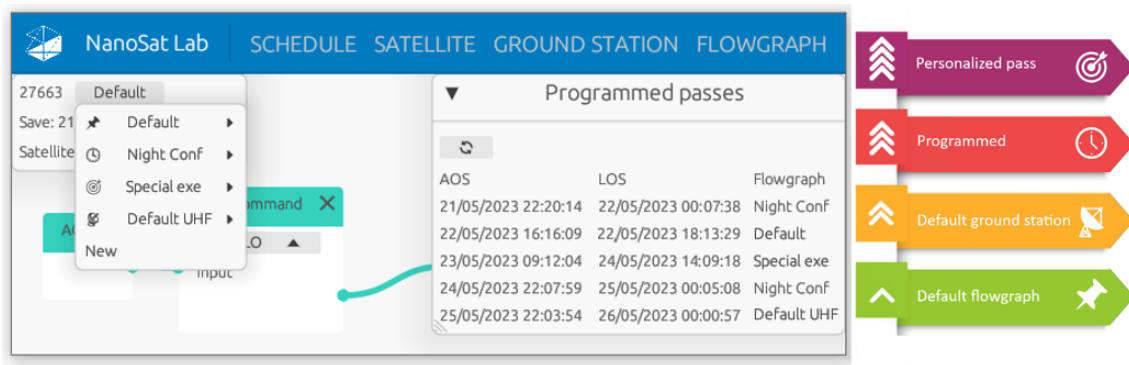


Figure 2.2: Flowgraph mode configuration and assignment of flowgraph for each satellite pass based on mode preference.

### 2.2.3. Execution manager

The Execution Manager is responsible for controlling and regulating communications with satellites, acting as the background task that maintains overall control. It uses the data stored in the OpGen's database to control and schedule the communication timings of the satellites.

The Execution Manager performs the following three main functions:

1. **Maintenance of Scheduled Passes:** Keeps track of the latest version of the scheduled passes by monitoring any changes that may affect the planned communication sessions. Whenever a new pass is scheduled or an older one is removed, the system verifies if any adjustments need to be made to the overall schedule.
2. **Flowgraph Execution Lock:** Prevents the editing of flowgraphs that are currently executing to ensure that no modifications occur during execution. It informs the start of the flowgraph execution and locks any further editing during the execution to maintain the integrity of the flowgraph and avoid potential conflicts or disruptions in the ongoing execution process.
3. **Operator Interaction Support:** Offers information and tools to the user's websocket threads, allowing them to interact with the execution of a specific satellite. It gives the operators detailed information about the flowgraph design that has been chosen for the pass, so they can keep track of the specific configuration and parameters used. Additionally, it provides a dedicated communication channel for monitoring and interacting with the execution in real-time.

For this reason, it must maintain a list of various attributes for each satellite and its next pass. These attributes include:

- **NORAD ID:** The unique identifier of the satellite.
- **Pass ID:** A unique identifier for the pass scheduled for the satellite.
- **Start Time:** The date and time scheduled for the communication session with the satellite. This information is used to determine the waiting time before initiating the

execution of the corresponding flowgraph. In addition, it is also compared with newly scheduled passes to ensure accuracy.

- **Execution Flag:** A flag indicating whether the flowgraph of the satellite is currently executing. It acts as a mechanism to prevent operators from editing the flowgraph while it is running to maintain the consistency of the operation.
- **Broadcast Sender:** This feature allows operators to interact with the execution of the flowgraph (see in Chapter 3.2.4.).
- **Flowgraph Identifier:** During a communication session, it holds the identifier of the flowgraph being executed. This identifier enables operators to have a graphical visualization of the ongoing process.
- **Task Thread:** This thread is responsible for the execution process and remains in a waiting state until the start time is reached. Whenever an update occurs for the next pass of a satellite, the current execution task is aborted, and a new task with the updated start time is initiated.

When the software is initialized, the Execution Manager queries the database to determine the next scheduled satellite pass. For each satellite with a scheduled pass, it generates a list with all the previously mentioned attributes. Simultaneously, it initiates a dedicated thread for each satellite, which will remain waiting until the designated start time is reached.

Once the start time is reached, the thread should notify the Execution Manager that it is ready to begin the execution. It will also provide information such as the flowgraph selected to be executed, the communication channel through which operators can follow and interact with the ongoing execution, and initiates the execution of the selected flowgraph.

After the execution of a flowgraph is completed, the Execution Manager needs to consult the database again to check if there is a scheduled pass for the satellite that just finished executing. If a scheduled pass is found, the Execution Manager should prepare a new thread to execute the flowgraph for the upcoming pass.

At the same time, the Execution Manager will need to receive notifications from the database regarding updates made by the Scheduling software. These updates can have an impact on the execution threads, and the Manager should monitor all of them to ensure continuous operation and timely execution for all satellites. This iterative process ensures that the software will remain responsive to changes in the scheduled passes and will be always prepared to execute the appropriate flowgraph for each satellite's communication session. The changes that must be checked are the following:

- **Scheduling a New Pass:** When this notification is received, the Execution Manager checks if there is any existing execution planned for that satellite. If there is no current execution, a new entry is created with the updated information, and a corresponding execution thread is initiated. However, if there is already an existing execution planned for that satellite, the Execution Manager compares the start time of the new pass with the previous one. If the new has an earlier start time and the ongoing execution has not started yet, the existing execution will be terminated, and replaced with the new pass.

- **Removal of a Pass:** In this case, the Execution Manager checks if the pass being removed is one of the scheduled passes that are still awaiting execution. If the pass is not currently being executed, it is removed from the list. After removing the pass, the Execution Manager consults the database for the next scheduled pass for the corresponding satellite.
- **Removal of a Satellite:** Similar to the previous case, the Execution Manager checks if the satellite is currently executing a flowgraph. If the satellite is in the middle of an execution, the Manager will wait for the flowgraph to complete, ensuring that the ongoing execution is not interrupted. Otherwise, it will delete the thread assigned to it.
- **Change of NORAD ID a Satellite:** This operation carries a potential risk of introducing errors if performed while a flowgraph execution is in progress, as it may affect the references used during execution. Therefore, it is crucial to ensure that no execution is ongoing for the satellite before proceeding with any update or modification. The operator will need to wait until the current execution is complete before performing the NORAD ID update, if not the software will refuse this petition. The Execution Manager, responsible for managing flowgraph executions, will consult the database to determine the next scheduled pass for the satellite with the updated NORAD ID. Once the updated information is obtained, a new entry will be created in the database, ensuring that the updated NORAD ID is properly reflected for future executions.

#### 2.2.4. Flowgraph execution modes

During the execution of a pass, two possible scenarios can occur. The first scenario occurs when a pass is scheduled to reserve the connection with the GS, as there is a desire to execute the message exchange using another software independent of the current one. In this case, the software executes only the necessary Utility Blocks to ensure that the communication channel remains free during the designated time and to provide information about the antenna movements that need to be performed by the GS.

In the second scenario, the communication process follows a planned execution using the flowgraph model. This involves the sequential execution of all the blocks that have been carefully designed and included in the flowgraph to achieve the desired communication objectives. The flowgraph serves as a roadmap for the execution, outlining the sequence of operations and tasks to be performed during the communication session. Each block represents a specific function or action, such as data transmission, telemetry collection, command execution, or other relevant tasks.

In this case, the execution of the flowgraph can either be fully autonomous or monitored by an operator. If an operator is actively monitoring the execution, they can intervene at any moment by switching to manual execution mode. In manual execution mode, the operator can pause the planned execution to send additional commands considered necessary for the current circumstances.

This mode provides the operator with the flexibility to make real-time adjustments during the communication process, controlling the execution and evaluating all the situations. It

allows the customization of the commands to be sent, enabling the operator to respond to unexpected events or optimize the communication process based on real-time information. Furthermore, manual execution mode allows for the interruption of a specific block's execution if the operator believes it is not functioning properly. This capability ensures that any issues or anomalies can be promptly addressed and mitigated, maintaining the integrity and effectiveness of the overall communication session.

## 2.3. Data downlink and storage

The main objective of EO missions is the collection and study of data. Therefore, it is crucial to collect and store all the data exchanged, scientific experiments, and telemetry received from the satellite and logs generated during the mission for later analysis.

During the execution of the flowgraph, a specific background block called the Data Storage Block is responsible for handling the data received from the satellite and storing it.

The messages sent by the satellite contain various types of valuable data, including responses to commands, telemetry beacons, and experiment results. Each message type requires specific treatment due to its unique structure and content. To interpret these messages, they need to be deserialized into different fields. This involves extracting the raw bytes of the messages and converting them into their respective data types. Depending on the specific data type, binary data may need to be converted into integers, floats, strings, or other appropriate formats. Additionally, conversion factors may be applied to obtain the desired information from the data.

Telemetry data provides information about the satellite's performance, health, and status. It serves as a valuable source of information for monitoring and diagnosing the satellite's behavior. By analyzing telemetry data, operators can identify anomalies, detect potential issues, and take corrective actions to ensure the satellite operates optimally. Storing telemetry data enables historical analysis and trend identification. By having historical telemetry records, patterns and trends can be identified over time, aiding in performance optimization, predictive maintenance, and anomaly detection.

Scientific data obtained from experiments conducted by the satellite is also crucial to store. This data may include measurements, observations, or any other experimental results. Storing this scientific data allows researchers and scientists to retrieve and analyze it later, facilitating in-depth studies, trend analysis, and scientific discoveries.

In addition to the data received from the satellite, it is important to store the logs generated during the communication to maintain a complete record of all events and actions related to the satellite's operation. Logs provide a valuable source of information for troubleshooting and post-mission analysis. They can help identify anomalies, track system behavior, and assist in diagnosing issues that may arise during satellite operations.

The files generated by scientific experiments and the logs are stored as separate files, which can be accessed by the operator through the graphical interface. On the other hand, telemetry data requires to be stored in a database capable of handling large volumes of data and providing efficient querying capabilities. InfluxDB<sup>2</sup> is a widely-used option for

---

<sup>2</sup>InfluxDB official webpage: <https://www.influxdata.com/>.

storing time-series data like telemetry due to its scalability and optimized storage structure and in combination with Grafana<sup>3</sup>, offers a user-friendly interface for creating customizable dashboards and visualizations. This combination enables operators to monitor and analyze telemetry data in real-time, enhancing their ability to make informed decisions and optimize operations.

## 2.4. Access levels

Just as the software has been designed with distinct components to fulfill different functionalities, it is equally important to differentiate the roles and responsibilities involved in satellite operations. This differentiation ensures that each operator is assigned specific tasks aligned with their designated role, promoting a secure utilization of the software.

By assigning specific roles to operators, the software can enforce access restrictions and permissions based on their assigned tasks. This helps prevent unauthorized use of the software and mitigates the risk of unintended actions or disruptions.

When managing the satellite, the operators have specific responsibilities that can be divided into three main roles: Ground Station Manager, Satellite Operator, and Telemetry Expert. These roles are assigned to operators based on their assigned tasks and expertise.

By logging into the software with their credentials, operators are assigned access levels according to their roles and responsibilities. This ensures that each operator has restricted requests and actions, as well as limited views and operations available to them in the graphical interface.

### 2.4.1. Ground Station Manager

The Ground Station Manager is responsible for monitoring and managing the availability of the various GSs used for satellite communications. Their primary responsibility is to monitor the status and availability of these GSs, providing essential information regarding their operational status and maintenance activities. This information is vital for determining whether communication sessions can be established with the satellites.

Additionally, they hold the responsibility of managing the satellites, including tasks such as adding new satellites, removing existing ones, and making necessary configuration adjustments. Besides, they have the authority to plan satellite passes using the scheduling module. This involves careful consideration of various factors such as the duration of the passes, the maximum elevation achievable during each pass, and the availability of the GSs, to make sure that the objectives of the communications will be successfully met.

---

<sup>3</sup>Influx + Grafana: <https://docs.influxdata.com/influxdb/cloud/tools/grafana/>.

### 2.4.2. Satellite Operator

The Satellite Operator is responsible for designing the different flowgraphs that will be executed during the scheduled passes. Their role involves carefully planning the communication exchange with the satellite, defining objectives, and determining the sequence of commands to be executed. During the contact with the satellite, the Satellite Operator assumes the responsibility of ensuring the proper execution of the flowgraph. They monitor the communication process closely, ensuring that commands are transmitted to the satellite and received successfully.

They also verify that the satellite responds appropriately to the commands, checking that the desired actions are being carried out. They may also handle error handling, contingency planning, and troubleshooting in case of any issues or anomalies during the communication session.

In addition, they have access to all the data received or generated during the execution of the flowgraphs. This includes telemetry data from the satellite, command logs, execution status, and any other relevant information.

### 2.4.3. Telemetry Expert

The Telemetry Expert plays a complementary role to the Satellite Operator mentioned earlier. Their primary responsibility is to closely monitor the communication with the satellite and ensure the integrity and accuracy of the received telemetry data.

The Telemetry Expert continuously analyzes the telemetry data, which includes various parameters such as satellite health status, sensor readings, power levels, temperature, and other relevant information. They compare the received telemetry data with expected values and predefined thresholds to identify any anomalies or deviations from normal behavior. When unusual or unexpected behavior is detected, the Telemetry Expert alerts the Satellite Operator, who can then take appropriate action to address the issue by adjusting the command sequences, or initiating contingency plans if necessary.

## 2.5. Requirements

All the requirements mentioned in the prior sections have been compiled in the subsequent tables, separating the tasks of the backend (Table 2.1) and the frontend (Table 2.2). Additionally, these tables indicate the procedures to follow in order to verify their correct functionality.

ID	Topic	Description	Verification
100	Scheduler	The operator should be able to add, edit or delete a satellite on a DB.	Test
110	Scheduler	The operator should be able to upload the TLE of a specific satellite.	Test
120	Scheduler	The operator should be able to modify the NORAD of a specific satellite.	Test
130	Scheduler	The operator should be able to modify whether a ground station is operative or not on a DB.	Test
140	Scheduler	The operator shall be able to delete one or multiple satellite passes when they haven't been done yet.	Test
150	Scheduler	The OpCenter shall be able to obtain the TLE from NORAD.	Analysis
160	Scheduler	The OpCenter shall be able to find the most suitable GS for a specific satellite.	Test
170	Scheduler	The OpCenter shall be able to compute the Look Angles from a TLE on a specific date.	Analysis
180	Scheduler	The OpCenter shall be able to find when a satellite, from his NORAD or TLE, is going reach AOS (the time when the elevation becomes greater than zero) and LOS (the time when the elevation becomes negative) respect a specific GS location.	Analysis
190	Scheduler	The operator should be able to schedule a satellite pass and save it on a DB. → The max search time would be 48 hours by default. → Only will save the request that has not any conflict unless the flag 'Force save' is set, in these cases all passes without conflict will be saved.	Analysis
200	FG Configuration	The operator shall be able to upload the available commands of a satellite.	Test
210	FG Configuration	The operator shall be able to configure the chain of commands and processes to be execute during a satellite communication (Flowgraph).	Analysis
220	FG Configuration	The operator shall be able to add, edit or delete a flowgraph.	Test
230	FG Configuration	The OpCenter shall be able to retrieve the configuration stored in the DB of a flowgraph to be sent to the client side. List of instantiated blocks, block parameters, and connections.	Test
240	FG Configuration	The OpCenter shall track the edition of a flowgraph (add or delete a block, add or delete the connection, configure block, upload a file,...) and validate the configurable parameters. Also shall inform about the changes to the interested operators.	Platform Analysis
250	FG Execution	The OpCenter shall wait until it is time to execute a pass, load the flowgraph that must be executed and execute it automatically.	Analysis
260	FG Execution	The OpCenter should send informative messages about the current execution of a flowgraph.	Platform Analysis
270	FG Execution	The intervention of the operator must be allowed in cases of the execution of a pass needs it in a conditional block.	Platform Analysis
280	FG Execution	The PANIC mode should stop the execution of a flowgraph immediately and only execute the commands set by the operator.	Platform Analysis
290	FG Execution	The MANUAL mode should allow the operator to configure and send commands in real-time.	Platform Analysis
300	FG Execution	The OpCenter shall decode all the messages sent by satellite and save the information in an InfluxDB if its telemetry data or in a File in the other cases.	Analysis
310	Utility Blocks	AOS checks every second if the satellite is in sight, when the elevation is higher than zero the execution of the flowgraph starts.	Analysis
320	Utility Blocks	LOS checks every second if the satellite is lost from sight, when the elevation is lower than zero the execution of the flowgraph ends.	Analysis
330	Utility Blocks	For the scheduled passes, the software should calculate all the movements that must be done by the antenna	Analysis
340	Utility Blocks	The software should send the movements of the antenna to the ground station	Analysis
350	Mission Blocks	SendCommand build the data packet from the id of the corresponding command, send it through the ZMQ socket, and wait for the response of the satellite.	Analysis
360	Mission Blocks	SendFileCommand builds the data packet from the id and the conf file of the corresponding command, sends it through the ZMQ socket and wait for the response of the satellite.	Analysis
370	Background Blocks	The OpCenter shall establish a ZMQ connection to send and receive between the OpCenter and the Ground Station.	Analysis
380	Authentication	The operator can identify himself with his username and password (LDAP) and receive a token for his future communications with the server.	Analysis
380	Authentication	A user should have different roles: GS Manager, Satellite Operator, Telemetry Expert.	Analysis

Table 2.1: Backend Requirements



ID	Topic	Description	Verification
100	Schedule	The operator shall be allowed to add a satellite by introducing the NORAD, the name, the uplink, and downlink frequencies, the bandwidth, and the TLE ( introduced manually or configured to automatically fetch at Celestrak) of a satellite through a form.	Platform Analysis
110	Schedule	The operator shall be allowed to modify a satellite from a form filled with the name, the uplink and downlink frequencies, the bandwidth, and the TLE configuration of a satellite.	Platform Analysis
120	Schedule	The operator shall be able to delete a satellite. From the satellite display list, pressing a button will open a confirmation window.	Platform Analysis
130	Schedule	The operator shall be able to change the NORAD of a satellite by introducing it in a form.	Platform Analysis
140	Schedule	The operator can visualize the parameters of the ground stations like name, location, uplink, and downlink frequency range and bandwidth and if its operative or not.	Platform Analysis
150	Schedule	The operator shall be able to set if a ground station is operative or not by a toggle switch.	Platform Analysis
160	Schedule	The operator shall be able to schedule a satellite pass by introducing the NORAD and choosing between the following options: → One pass. * → Multiple passes, by introducing the number of passes. * → Multiple passes from date, by introducing the start date and the number of passes. * → All passes until date, by introducing the final date. → All passes in time intervals, by introducing the start and final date. Also, the operator would be able to set the flag of "Force save" , and the max search period will be the default unless a different one is entered (Only configurable in the * modes).	Platform Analysis
170	Schedule	The operator shall be able to delete one or multiple satellite passes when they haven't been done yet, indicating its pass id.	Platform Analysis
180	Schedule	The operator shall be able to view all the programmed satellite passes and the ones that have already been carried out. The passes will be displayed in a table showing the NORAD, the start time, end time, AOS time, LOS time, and max elevation.	Platform Analysis
190	Schedule	The operator should filter the list of passes by the NORAD, max elevation, gs,...	Platform Analysis
200	Schedule	The operator shall be able to view a list with all the satellites, including the name, NORAD, uplink freq, downlink freq, BW, and TLE.	Platform Analysis
210	Schedule	The operator shall be able to view a list with all the Ground Stations, including name, range of operational freq, location, and operational state.	Platform Analysis
220	Schedule	The operator should be able to view all detailed information of a specific pass.	Platform Analysis
230	Schedule	The operator can visualize the passes in a timeline.	Platform Analysis
240	FG Configuration	The operator shall be able to upload a TOML file with all the available telecommands of a specific satellite.	Platform Analysis
250	FG Configuration	The operator shall view a list of all the satellites that have at least one flowgraph and the date of its last update.	Platform Analysis
260	FG Configuration	The operator shall be able to delete a flowgraph.	Platform Analysis
270	FG Configuration	The operator shall be able to create a new flowgraph (With the AOS block by default). The operator would introduce the name of the flowgraph through a form. If the satellite has at least one flowgraph, would select which configuration will be used when selecting which flowgraph should be executed for a given pass.	Platform Analysis
280	FG Configuration	The operator shall be able to visualize a dropdown with the name of all the flowgraphs of the current satellite.	Platform Analysis
290	FG Configuration	The operator shall be able to edit a flowgraph if it is not currently being executed.	Platform Analysis
300	FG Configuration	The operator shall be able to add a block to send a command.	Platform Analysis
310	FG Configuration	The operator shall be able to add a block to send a command with a configuration file.	Platform Analysis
320	FG Configuration	The operator shall be able to select the command to be sent on a block from the ones available for the satellite.	Platform Analysis
330	FG Configuration	The operator shall be able to upload a configuration file, this will be presented with a file picker in case it is required by the block.	Platform Analysis
340	FG Configuration	The operator shall be able to edit the relation between blocks, through connectors from the output of one block to the input of another.	Platform Analysis
350	FG Configuration	The operator shall be able to delete one block.	Platform Analysis
360	FG Configuration	The operator shall be able to configure timeout time and number of attempts of a block.	Platform Analysis
370	FG Configuration	The operator should be able to see all the configurations updates on a specific flowgraph in real-time (websocket)	Platform Analysis
380	FG Configuration	The operator should be able to add a flowgraph for a specific satellite or pass (save more than one flowgraph configuration for a satellite)	Platform Analysis
390	FG Configuration	The operator should be able to view which flowgraph will be executed for a programmed pass.	Platform Analysis
400	FG Execution	The current execution of a pass should be displayed (visual graphic representation)	Platform Analysis
410	FG Execution	The operator should be able to visualize the informative messages sent by the server about the execution of a flowgraph on an auxiliary window.	Platform Analysis
420	FG Execution	The operator should be able to filter the informative messages of the auxiliary window, by the tags or by contents.	Platform Analysis
430	FG Execution	The operator should be able to download the files generated during the execution of the satellite communication.	Platform Analysis
440	FG Execution	The operator must be allowed to stop the execution and send manual commands.	Platform Analysis
450	Access Levels	The operator shall introduce his credentials in a form to access the application.	Platform Analysis
460	Access Levels	The views shall be restricted as function of the access level of the operator.	Platform Analysis
470	GUI app	The GUI shall work as a desktop app for the OS Windows, Linux, and MAC and also shall be available for web compiled with WASM.	Platform Analysis

Table 2.2: Frontend Requirements



# CHAPTER 3. SOFTWARE DESIGN

This chapter presents the design and implementation of the software, discussing the decision-making process and justifying the chosen technologies based on the previously defined requirements.

## 3.1. Architecture

The software is designed with a client-server structure, where the client requests services or resources from the server, and the server provides those services or resources in response to client requests.

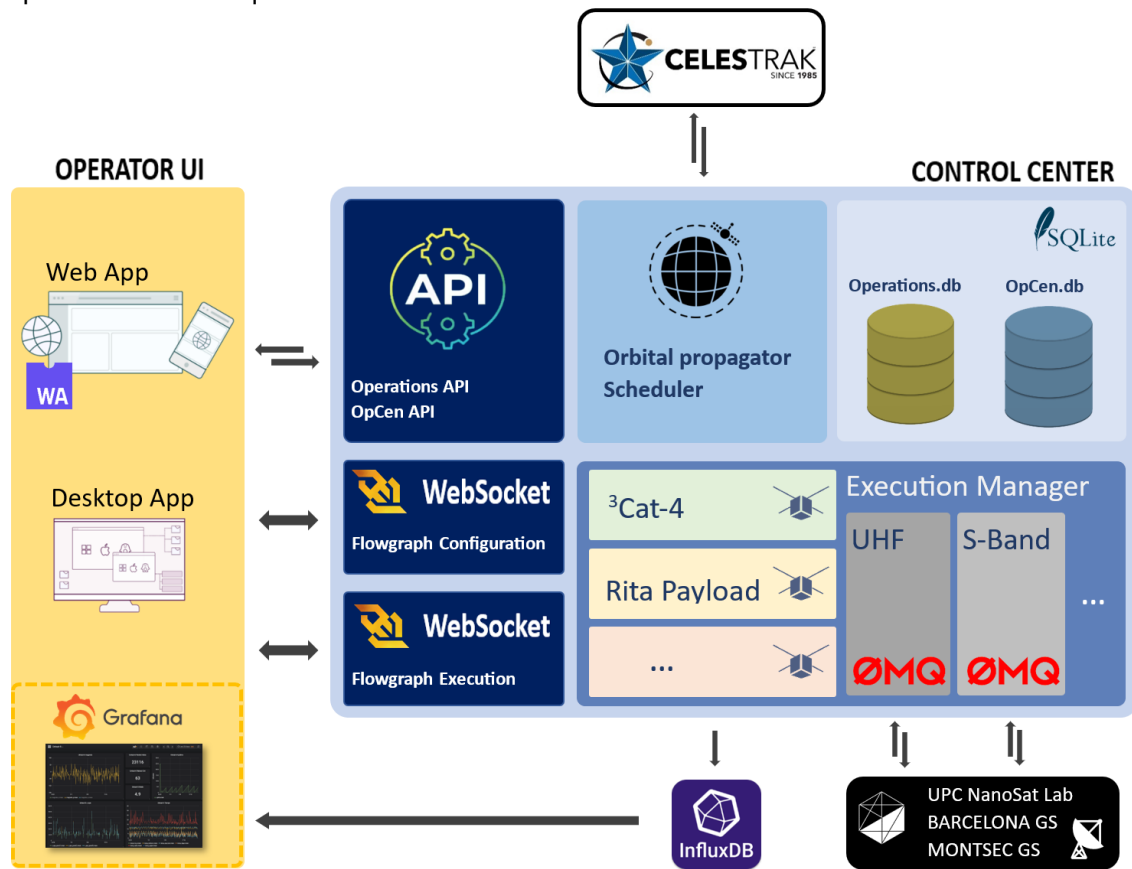


Figure 3.1: Software architecture

The server, located at the UPC NanoSat Lab's OpCen, offers three primary services: Scheduling, Flowgraph-based TT&C, and Data downlink and storage. Each service has its independent database for data storage and management, all hosted on the same server. This separation ensures data isolation and enables independent management and processing of the services.

To ensure synchronization between the Flowgraph and the Scheduling module, a communication channel is established between the two. This allows for the notification of any updates or changes in the satellite pass planning, enabling the Flowgraph module to adjust its execution accordingly and ensuring seamless coordination. Additionally, during the execution of flowgraphs, the software establishes a connection with any of the GS in the network.

For the client, two graphical user interface options have been developed. The first option is a web application hosted on the server, providing easy accessibility from any device with a web browser, while the second one is a desktop application. Both interfaces offer the same set of functionalities, allowing users to choose the one that best suits their needs and preferences.

To allow interaction between the clients and the server, a RESTful API and a WebSocket interface are utilized. This standardized interface allows clients to make queries, retrieve data, and fully utilize the software's capabilities.

Additionally, telemetry data obtained during communications can be visualized through the interface provided by Grafana.

In the following sections, a more in-depth discussion will be provided about the process and the functionalities of the previously introduced aspects.

### 3.1.1. Programming environment

For the development of the program, the Rust<sup>1</sup> language has been chosen due to its reputation built on its speed, memory safety, and thread safety. Rust features enable the detection of numerous common errors during compilation, ensuring code correctness and reducing the possibility of runtime errors. Its focus on memory safety, which helps to prevent issues like null pointer dereferences and buffer overflows, enhancing the overall reliability and security of the software [8].

The reason for choosing Rust for the development of the ground segment in satellite communications is that it involves critical tasks such as command execution, telemetry processing, and data storage. By preventing programming errors, Rust helps minimize the risk of crashes, data corruption, and security vulnerabilities.

Furthermore, this project requires concurrent handling of multiple tasks simultaneously, such as handling operator requests while communicating with satellites, receiving telemetry, sending commands, and managing data storage. Rust, combined with the Tokio<sup>2</sup> framework, provides efficient and scalable solutions for managing concurrent operations in an async environment.

Also, real-time processing and low-latency communication are crucial in the ground segment. Rust's emphasis on low-level control and zero-cost abstractions enables efficient memory management and optimal performance, making it suitable for these demanding requirements.

Considering these factors, Rust emerges as the perfect choice for developing the ground segment in satellite communications, as it provides the necessary safety, concurrency, performance, and memory management features required for such critical and complex tasks.

Additionally, the Rust community offers an extensive ecosystem of third-party crates, enabling developers to create clean, maintainable, and reusable code. Some notable examples used in this project include: **serde**<sup>3</sup>, used for the serialization and deserialization of

---

<sup>1</sup>Rust official webpage: <https://www.rust-lang.org/>.

<sup>2</sup>Tokio Docs: <https://docs.rs/tokio/latest/tokio/>.

<sup>3</sup>serde official webpage: <https://serde.rs/>.

data structures; **tracing**<sup>4</sup>, which provides a flexible framework for instrumenting, collecting, and analyzing diagnostic information in applications; **anyhow**<sup>5</sup>, for error handling, and **endian\_codec**<sup>6</sup>, for encoding and decoding data in different endianness formats, making it easy to work with binary data in big-endian or little-endian byte order.

#### 3.1.1.1. Async ecosystem with Tokio

Tokio is an asynchronous runtime for the Rust programming language, offering a powerful solution for handling concurrent and asynchronous tasks. It provides a multi-threaded runtime for executing asynchronous code, an asynchronous version of the standard library, and a large ecosystem of libraries [7].

The following code demonstrates the usage of Tokio. In the example, two asynchronous tasks are concurrently executed using the `tokio::spawn`, which means that both tasks are running at the same time. The handles vector stores the tasks, and the program awaits the completion of each task. Tokio's runtime manages the scheduling and execution of these tasks, ensuring that they make progress concurrently without blocking each other.

```
#[tokio::main]
async fn main() {
    let mut handles = Vec::new();

    handles.push(tokio::spawn(async {
        // Async task1
    }));

    handles.push(tokio::spawn(async {
        // Async task2
    }));

    for handle in handles.into_iter() {
        let _ = handle.await;
    }
}
```

In the project, this behavior is applied to the whole software. The main execution of the Op-Gen program utilizes multiple threads to handle different tasks concurrently. Three threads are distinguished at the top of the program.

The first thread is dedicated to supporting the server, which allows operators to make queries and requests. Inside this thread, each time an operator makes a query, a new thread is created and destined to handle their request. This approach enables transparent service provision to multiple operators simultaneously.

The second thread is dedicated to periodically updating the TLE of the satellites registered on the system. This ensures that the latest orbital information is available for accurate satellite tracking and management. Most of the time, this thread remains in an idle state,

---

<sup>4</sup>tracing Docs: <https://docs.rs/tracing/latest/tracing/>.

<sup>5</sup>anyhow Docs: <https://docs.rs/anyhow/latest/anyhow/>.

<sup>6</sup>endian\_codec Docs: [https://docs.rs/endian\\_codec/latest/endian\\_codec/](https://docs.rs/endian_codec/latest/endian_codec/).

without consuming system resources, and it only becomes active when it's time to perform the TLE updates according to the predefined refresh interval.

Finally, there is a dedicated thread for the Execution Manager, which is responsible for managing the execution of satellite operations. This thread creates and controls individual threads for each satellite, ensuring proper coordination and management of their respective tasks. More detailed information about this process will be provided later.

Also, the complexity of this project requires handling many tasks in parallel, and while some tasks can be executed independently of each other, in most cases, there is a need for a mechanism to communicate and exchange information between different threads of the program. To facilitate these communications, channels are used. The Tokio library enables asynchronous communication, which allows for non-blocking execution. For example, if a thread is waiting for a message, it remains inactive, waiting for the arrival of a new message, freeing up resources that can be dedicated to other tasks. The Tokio library provides four different types of channels, three of which will be used throughout the project:

- **mpsc** (multi-producer, single-consumer): This type of channel allows communication between multiple producers and a single consumer. It is useful when multiple threads need to send information to a single receiving thread.
- **oneshot**: This type of channel allows sending a value or an error once from a sending thread to a receiving thread. It is useful when a one-time signal needs to be sent from one thread to another.
- **broadcast**: This type of channel allows sending messages to multiple consumers. Each time a message is sent, all connected consumers will receive a copy of the message. It is useful when a message needs to be sent to multiple recipients.

These channels provide efficient and safe mechanisms for communication between threads, enabling effective coordination of tasks and maintaining synchronization in a concurrent environment.

### 3.1.2. Database Models

#### 3.1.2.1. *SQLite Database*

When deciding on the type of database to use for data storage, the choice between a relational or non-relational database depends on several factors, such as the nature of the data, project requirements, and query needs. A relational database offers a well-defined structure with tables, rows, and columns, and supports complex queries. On the other hand, a non-relational database can handle large volumes of data with a flexible schema and allows queries with high read and write performance.

Given the well-defined structure of the data models that will be discussed in the following sections, a relational database has been chosen as the preferred storage solution. Specifically, SQLite<sup>7</sup> has been selected due to its lightweight nature and simplicity. These qualities make it an ideal choice for efficient storage and management of structured data.

---

<sup>7</sup>SQLite webpage: <https://sqlite.org/index.html>.

This type of database provides a structured way to store and manage data. It consists in tables, which are organized structures of rows and columns. Each table represents a specific entity or type of information. Rows contain individual items, and columns represent the attributes or characteristics of those items. Tables can be related to each other to retrieve related information.

The **rusqlite**<sup>8</sup> library provides a convenient way to integrate SQLite databases into Rust projects. One of its important features is the ability to prevent Structured Query Language (SQL) injection attacks. By using placeholders in the SQL statement and prepared statements, **rusqlite** ensures that user-provided values are treated as data rather than executable SQL code. This effectively mitigates the risk of SQL injection vulnerabilities, as the user input is properly escaped and quoted.

However, in situations where concurrent access to the database is required, such as in an asynchronous environment, proper concurrency mechanisms are necessary to ensure data integrity. This is particularly important because SQLite is an embedded database system that directly reads from and writes to the database file. To address this problem, the combination of the **tokio-rusqlite**<sup>9</sup> library and the **rusqlite** library provides mechanisms for locking and managing concurrent access to the SQLite database.

The following example illustrates how a query is performed asynchronously using the aforementioned libraries. The `conn.call()` method is used to execute the query asynchronously, while the `query_row()` method is employed to retrieve a single row from the result set. The purpose of this specific query is to fetch the TLE of a satellite based on its `norad_id`. If everything goes as planned, the function will return its corresponding TLE.

```
async fn get_tle_satellite(conn: &Connection, norad_id:
    i32) -> anyhow::Result<TLE, Error> {

    let res = conn.call(move |conn| {
        conn.query_row(
            "SELECT tle1, tle2 FROM satellites WHERE norad_id =
              ?1",
            params![norad_id],
            |row| {
                let line1: String = row.get(0).unwrap();
                let line2: String = row.get(1).unwrap();
                Ok(TLE { line1, line2 })
            },)
    }).await.context("Failed get TLE")?;

    Ok(res)
}
```

<sup>8</sup>rusqlite GitHub repository: <https://github.com/rusqlite/rusqlite>.

<sup>9</sup>tokio-rusqlite Docs: [https://docs.rs/tokio-rusqlite/latest/tokio\\_rusqlite/](https://docs.rs/tokio-rusqlite/latest/tokio_rusqlite/).



### 3.1.2.2. Scheduling models

Scheduling is responsible for the forecasting and scheduling of satellite contacts. The necessity of establishing a data structure capable of storing all relevant information on such an event is reflected in the table *Passes*. This table should store relevant information about each pass, including the satellite and GS involved, the timing of the communication, and additional details like the maximum elevation of the satellite.

Two additional Database models, *Ground station* and *Satellite*, must also be defined. The *Ground station* table includes fields such as localization, operating frequency range, available bandwidth, and other pertinent information. Similarly, the *Satellite* table stores data on the satellite's operating frequencies and bandwidth. This information facilitates the identification of compatible GSs for each satellite based on their respective operational requirements.

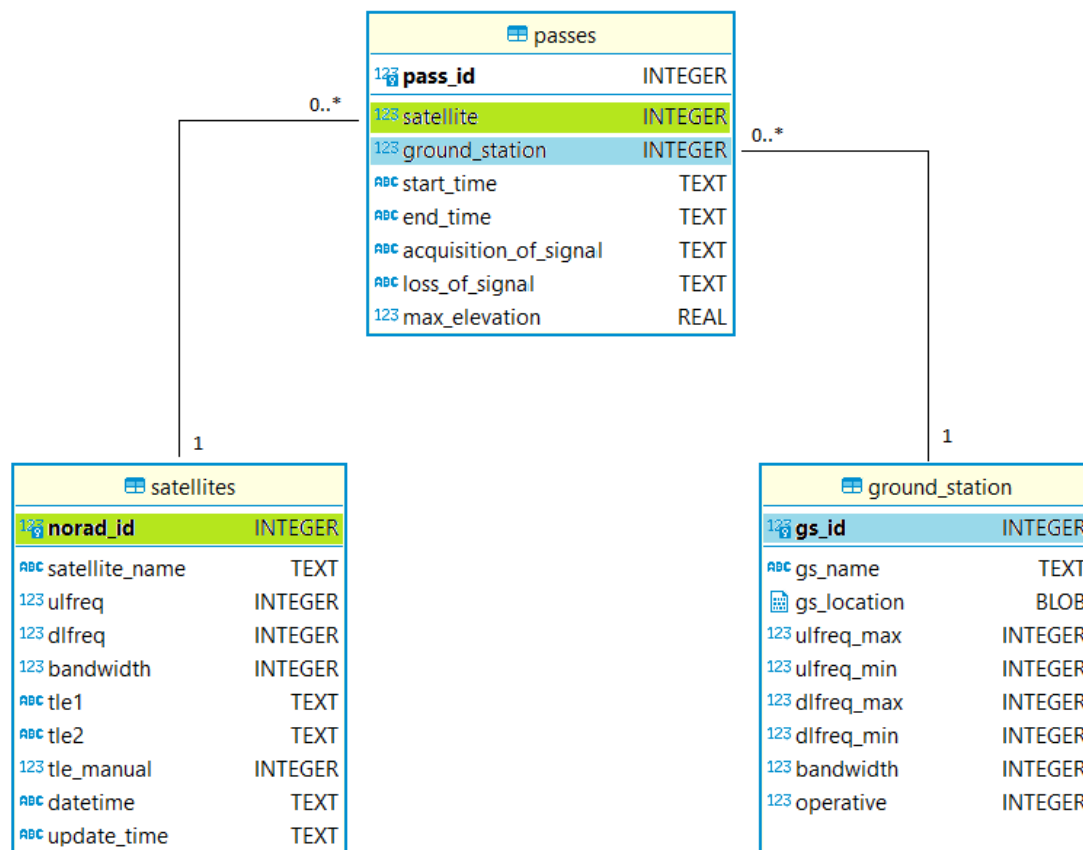


Figure 3.2: Scheduling Class diagram

The Class diagram shown in figure 3.2, defines how the different models will be related. Both, *Satellites* and *Ground station* are considered independent entities, while *Passes* is related to a single satellite and a GS.

Taking a closer look at the *Satellite* database model, the following attributes are established:

- **norad\_id:** This field is mandatory, and corresponds to the Satellite Catalog Number. This will be the unique identifier of the satellite, which will be used in the *Passes* table to link a pass to its satellite.



- **satellite\_name**: This field is only used as a complement to the `norad_id`, this will make the satellites easy to identify on the user's platform, providing human readability. It corresponds to the object name.
- **ulfreq**: The uplink frequency of the satellite is expressed in Hz, which is the frequency at which the ground transmitter sends the signal to the satellite.
- **dlfreq**: The downlink frequency of the satellite expressed in Hz, refers to the frequency at which the satellite transmits signals back to Earth for reception by the GS antenna.
- **bandwidth**: The spectrum of the signal transmitted by the satellite. It serves to identify which GSs are compatible for data transmission with the satellite, together with the *ulfreq* and *dlfreq*.
- **tle1**: Corresponds to the Line1 of the TLE, which contains essential information about the satellite, including its identification number, classification, and orbital information.
- **tle2**: Like the previous field, this corresponds to Line 2 of the TLE. It provides detailed orbital information about the satellite, such as the satellite's inclination, right ascension of the ascending node, and other information (see Chapter 1.2.3.2.).
- **tle\_manual**: A flag that indicates whether the origin of the TLE is of manual origin or has been obtained from the Celestrack API.
- **datetime**: Refers to the DateTime in a TLE, that indicates the specific moment in time at which the orbital elements are valid or applicable.
- **update\_time**: This field, unlike the previous one, provides the date and time at which the TLE data was obtained. It is important to update the TLE used to calculate the satellite's orbits since it has a limited validity period due to the dynamic nature of orbital motion and the influence of many factors. To achieve precise calculations and predictions, a more up-to-date set of orbital elements is required [2].

Also, for the *Ground station*, following attributes are defined:

- **gs\_id**: A unique identifier for the GS. This will later be used in the *Passes* table to link the pass to its GS.
- **gs\_name**: Indicates the name of the GS that should be displayed on the user's platform.
- **gs\_location**: This field contains the coordinates of the antenna, which are used to calculate the look angles. The latitude, longitude, and altitude are stored in a serialized object.

```
struct Coords {  
    latitude: f64,  
    longitude: f64,  
    altitude: f64,  
}
```

- **ulfreq\_max**: This field corresponds to the maximum operating frequency of the uplink. Together with the minimum frequency, it determines the range of operability of the GS.
- **ulfreq\_min**: As before, this field corresponds to the minimum frequency that a signal can transmit in the case of the uplink.
- **dlfreq\_max**: This field indicates the maximum frequency at which a signal can be received by the GS in the downlink.
- **dlfreq\_min**: This field represents the minimum frequency at which a signal can be received by the GS antenna in the case of the downlink.
- **bandwidth**: The bandwidth of the GS determines the maximum amplitude of the signal that can be sent or received.
- **operative**: A flag that indicates whether the GS is operative or not. Regardless, this does not affect when a pass is scheduled; it will only be checked when starting a flowgraph execution.

Finally, the *Passes* model contains the following elements:

- **pass\_id**: Unique identifier of the pass.
- **satellite**: A foreign key that connects a pass to the corresponding satellite, refers to the NORAD ID of the satellite.
- **ground\_station**: A foreign key that connects a pass to the scheduled GS.
- **start\_time**: The date and time when the preparation of the pass will begin. The edition of the flowgraph corresponding to the satellite will be blocked and the first block will begin the execution of the AOS block, aiming to obtain the most precise start of contact with the satellite.
- **end\_time**: The date and time at which the pass is considered complete, this indicates that the GS is available for starting a communication with another satellite.
- **acquisition\_of\_signal**: It refers to the moment when the communication between the satellite and the GS begins.
- **loss\_of\_signal**: It refers to the moment when satellite communication is expected to be interrupted due to the orbital trajectory.
- **max\_elevation**: This field refers to the highest point in the sky that the satellite will reach, this will help to determine the visibility and signal strength of the satellite at the given pass.

### 3.1.2.3. Flowgraph-based TT&C models

The operations software has two basic functionalities, on the one hand, it must provide support for editing and visualizing flowgraphs, as well as all the specific information of each block for its complete and correct execution during communication with the satellite.

In this way, the *Flowgraph* entity must group all the block instances that form it, let's call them *Block instance*. These instances are unique to each flowgraph, but their operation is common to other blocks, therefore, they must be referenced to a specific *Block definition*. Each instance can also have additional fields and variables beyond what would be the basic block, called *Block params*. To generate the chain of commands, the blocks must be able to be related to each other, this entity will be a *Connection*.

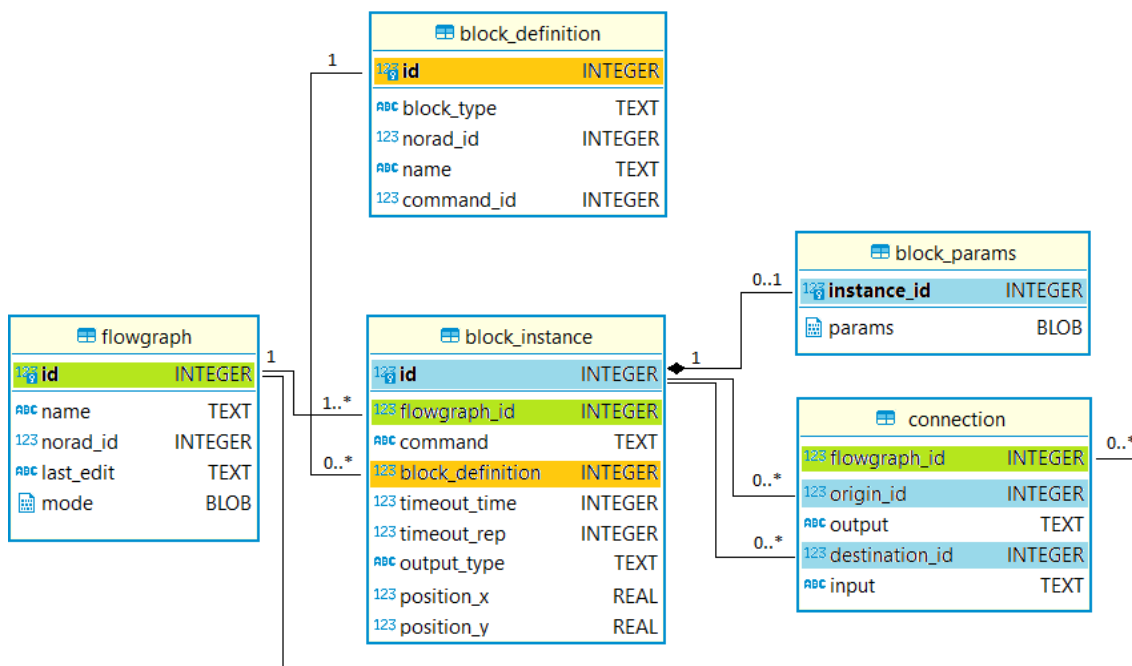


Figure 3.3: Flowgraph-based TT&C Class diagram

The Class diagram of figure 3.3 summarizes all the relationships that will be mentioned in this section. It is important to note that *Block definition* is an entity related to a specific satellite, and can be presented in multiple flowgraphs since a satellite can have different designs depending on the modes of operation. However, an *Block instance* is restricted to a flowgraph.

From the requirements, the following attributes are defined for *Flowgraph*:

- **id**: A mandatory unique identifier used to relate the blocks and connections with the corresponding flowgraph.
- **name**: This field refers to the name of the flowgraph in question, which allows differentiating the different designs of the same satellite.
- **norad\_id**: This field indicates to which satellite this design belongs, in such a way that it allows to identify of the specific blocks and commands that can be configured.
- **last\_edit**: Indicates when the last accepted update was made.

- **mode:** This field indicates which configuration mode has been assigned to this flowgraph, and stores all the additional information for the correct identification of the passes that must execute this flowgraph. This information is input via a serialized enum. Each of the different modes was discussed in the previous chapters (see Chapter 2.2.2.).

```
enum FlowgraphMode {
    Default,
    GS(i32),
    Time((i32, NaiveTime, NaiveTime)),
    Pass(Vec<i32>),
}
```

As mentioned before, all the blocks that can be used for the design of a flowgraph are previously defined and stored on a model of type *Block definition*. This one has the following attributes:

- **id:** A unique identifier to link an instance with the corresponding block definition.
- **block.type:** A field that serves both utilities, one to identify the type of widget that must be displayed on the user's platform, and used to identify the code to implementation during the execution of the desired block.
- **norad\_id:** The blocks that are supported for all satellites, this field is set to zero. Other than that, some commands are unique to each satellite. That is the case of Send Command and Send File Command, which will use NORAD to identify to which satellite they belong.
- **name:** This field corresponds to the name that must display the operation platform.
- **command.id:** As mentioned before, in the *norad.id* field, some blocks had a command, so this will store the corresponding id to set while the command is generated.

Furthermore, when a block is configured on a flowgraph, it will be an instance of a *Block definition* database model. This instance will be called as *Block instance*, and has the following attributes:

- **id:** A unique identifier to match the instance with its parameters and connections.
- **flowgraph.id:** A foreign key that links the instance to the flowgraph that it belongs.
- **command:** This field corresponds to the name that must display in the operation platform and is the same as the *name* field of *Block definition* at which corresponds.
- **block\_definition:** Link to the block definition.
- **timeout.time:** Defines the maximum time limit within which a response or completion of the communication process is expected to be received. This time limit serves as a threshold for determining if a communication operation has exceeded the acceptable duration.

- **timeout\_rep**: Specifies the number of attempts that will be made in the event of a timeout occurrence during the communication process. When a timeout happens and no response is received within the expected time, the block can be programmed to retry the operation multiple times before considering it as a failure.
- **output\_type**: This field indicates the mechanisms and conditions that will be used to determine the output of the block.
- **position\_x**: This field specifies the position of the x-axis on which the block must be displayed.
- **position\_y**: As before, this field corresponds to the position of the y-axis on which the block must be positioned.

As a complement to the *Block instance*, the *Block params*, is in charge of saving all that complementary information as:

- **instance\_id**: Identify the instance to which it belongs.
- **params**: A flexible field that accommodates data storage for various data structures in a serialized format. It serves as a container for different types of information, required for the blocks. For instance, in the case of the Send file command, would include fields like *filename* and *data*, while the Send bytes command would utilize this field to store the raw bytes of the content of the packet to be sent by the block.

Lastly, *Connection* is the entity that allows us to define the order and relationships between blocks, and has the following attributes:

- **flowgraph\_id**: This field might seem redundant since the connections are made from the instances, which are unique for each flowgraph. However, this reference to the flowgraph increases the search efficiency, making it faster and easier to find a connection between blocks.
- **origin\_id**: A foreign key that identifies the parent block of the connection.
- **output**: This field is only used in the operator platform when displaying the design and indicates the name of the output pin.
- **destination\_id**: A foreign key that identifies the child block of the connection.
- **input**: This field is only used in the operator platform when displaying the design and indicates the name of the input pin.

### 3.1.3. Interfaces

Given the nature of the software, a distributed client-server ecosystem, it is crucial to establish well-defined interfaces to ensure effective, fluid, and efficient intercommunication between the various components. The selection of appropriate messaging protocols for each scenario has been influenced by their ability to meet specific requirements such as real-time communication, scalability, reliability, and security.

The software has incorporated three different messaging protocols: Representational State Transfer (REST) API, WebSockets, and ZeroMQ. Each protocol serves a distinct purpose and offers unique advantages in facilitating communication between the server and the edge components.

#### 3.1.3.1. *RESTful API*

REST API is an architectural style for building distributed systems. It provides a set of well-defined endpoints that can be accessed by edge components through standard Hypertext Transfer Protocol (HTTP) methods. These methods request standard functions like creating, reading, updating, and deleting records (also known as CRUD) within a resource.

The choice of this protocol is based on its uniform interface and independence between the client and server. REST APIs allow all requests to have the same format regardless of their origin, and clients only need to know the Uniform Resource Identifier (URI) of the requested resource. This makes the APIs platform-independent and easily consumable by clients running on different platforms. Additionally, REST APIs are stateless, meaning the server does not store any client-specific information between requests. This simplifies server-side management and enables better scalability[3].

Thus, the API interface will be used for all communications that require sending, receiving, or manipulating information without direct user interaction. By utilizing the API interface, an efficient and standardized mechanism is established for exchanging information between the user platform and the server.

#### 3.1.3.2. *WebSocket*

WebSocket is a protocol that allows a persistent Transmission Control Protocol (TCP) connection, this means that a full-duplex connection is established allowing real-time communication between the server and the user. Unlike REST API, where the client is responsible for initiating requests to receive data updates, WebSocket allows the server to push updates to connected clients instantly[4]. This behavior is shown in figure 3.4.

This bi-directional connection makes WebSocket more suitable for scenarios that require continuous and cooperative communication, such as real-time execution and editing of flowgraphs. One notable advantage is the ability to collaboratively design and configure flowgraphs, allowing multiple users to work together and instantly verification of the design changes by the server.

For both the REST API and WebSockets interface, **Axum**<sup>10</sup> has been chosen as the framework. Axum is a web application framework known for its emphasis on ergonomics and modularity and it is specifically designed to leverage the capabilities of the Tokio ecosystem.

Axum offers an intuitive API that simplifies the process of defining routes, handling HTTP requests and responses, and implementing middleware. The modularity of Axum allows for flexibility in building and organizing web applications, making it suitable for projects of various sizes and complexities.

---

<sup>10</sup>Axum GitHub repository: <https://github.com/tokio-rs/axum>.

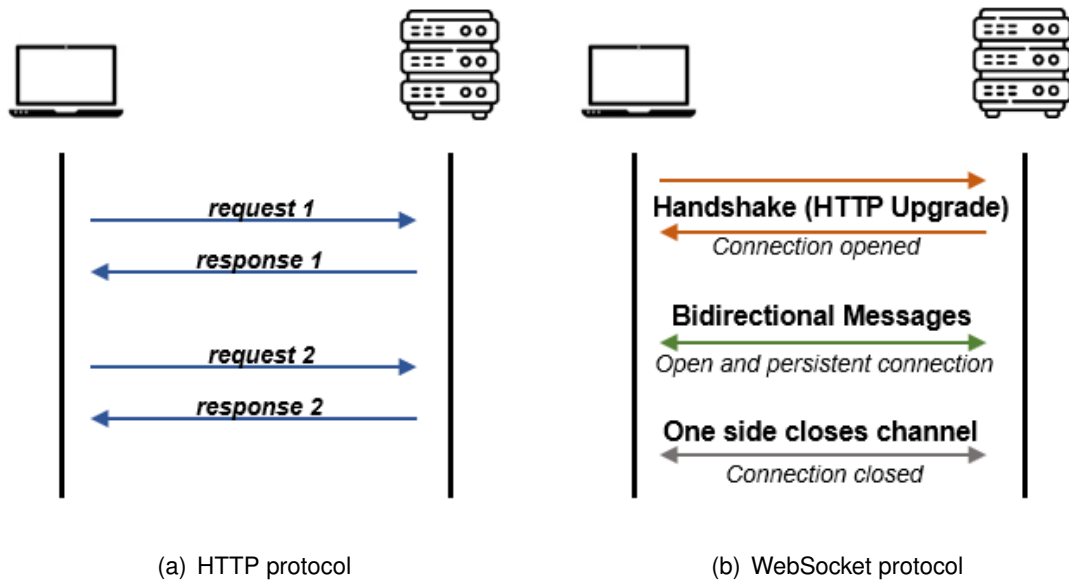


Figure 3.4: HTTP vs WebSocket.

### 3.1.3.3. ZeroMQ

ZeroMQ<sup>11</sup> is a high-performance asynchronous messaging library aimed at use in distributed systems. Unlike conventional sockets, that present a synchronous interface to either connection-oriented reliable byte streams, ZeroMQ sockets present an abstraction of an asynchronous message queue, with the exact queueing semantics depending on the socket type in use. This provides a lightweight and flexible infrastructure for building scalable and reliable messaging patterns [5].

ZeroMQ has been chosen as the communication framework for the interactions between the Operation center and the GS. This decision is based on its ability to provide high throughput, low latency, and fault tolerance, ensuring efficient and reliable data exchange between the two endpoints.

In Rust, the crate that provides a binding to the ZeroMQ library is **zmq**<sup>12</sup>. It offers a high-level API for creating sockets, sending and receiving messages, and implementing various communication patterns such as publish-subscribe, and push-pull, which are the ones that are going to be used in this project.

### 3.1.3.4. Cap'n Proto

In addition to the aforementioned messaging protocols, it is crucial to establish the format in which data will be exchanged between different components of the system. Cap'n Proto is a serialization protocol that has been chosen as the data format for its efficiency, compactness, and schema evolution capabilities.

Cap'n Proto provides a binary representation for serialized and deserialized data structures. One advantage is its code generation tool, which automatically generates code in various programming languages based on a defined schema. This capability not only

<sup>11</sup>ZMQ official webpage: <https://zeromq.org/>.

<sup>12</sup>Rust ZMQ Crate documentation: <https://docs.rs/zmq/latest/zmq/>.

saves development time but also allows for the seamless production and consumption of structured data without the need for manual implementation of serialization and deserialization logic. As a result, it simplifies the creation, manipulation, and access of values defined in the schema, making the process easier and more efficient.

However, the advantages of Cap'n Proto go much further. One notable feature that it offers is random access, which allows reading just one field of a message without parsing the entire data structure, also it supports zero-copy data transfer, which improves performance and reduces memory consumption. It also checks the structural integrity of the message just like any other serialization protocol would [1].

For Rust, the **capnp**<sup>13</sup> library contains the necessary facilities for reading and writing Cap'n Proto messages. Additionally, the **capnpc** tool enables the compilation of the data structures, generating Rust code that can be used in the project. Considering the need to share the models between the backend and the frontend, a library has been created to contain these models. This approach enables more efficient and practical utilization of the models in both projects, reducing redundant code and enhancing code reuse.

### 3.1.4. Date and time format

Date and time formats play a critical role in ensuring consistency and interoperability between systems. Inconsistencies in date and time representations can result in software malfunctions, incorrect calculations of satellite orbits, operator confusion, and loss of passes. To mitigate these issues, the RFC 3339 standard format has been chosen as the reference for handling date and time information.

RFC 3339: *YYYY-MM-DDTHH:MM:SSZ*

In this way, all dates processed on the server adhere to the Coordinated Universal Time (UTC) zone, providing a common reference point for data consistency. Additionally, this standardization also enables smooth operations with dates in the database. To avoid confusion in the user interface, dates will be displayed in their respective local time zones. However, it's important to note that before sending requests to the server, these dates must be properly parsed and converted to UTC to maintain consistency and accuracy.

### 3.1.5. Accessibility

The accessibility of the system has been implemented to ensure secure and controlled access, following the access levels defined by the requirements. To achieve this, a combination of Lightweight Directory Access Protocol (LDAP) for authentication and JSON Web Tokens (JWT) for authorization has been employed.

LDAP serves as the authentication mechanism, allowing users to verify their identities by validating their credentials against a centralized directory. This directory securely stores user information such as usernames and passwords. When a user attempts to login, their credentials are compared against the LDAP directory to authenticate their identity.

Once a user is authenticated, JWT comes into play for authorization purposes. When a successful authentication, the user is issued with a JWT token, which contains encrypted

---

<sup>13</sup>Cap'n Proto Github repository: <https://github.com/capnproto/capnproto-rust>.



information about their identity and role. This token is introduced on the header of the subsequent requests, so the server can validate the JWT token to ensure that the user has the necessary permissions and access rights to perform the requested actions.

By combining LDAP for authentication and JWT for authorization, the system establishes a robust and secure access control mechanism. LDAP ensures that only authenticated users with valid credentials can have access to the software, while JWT guaranty access based on the user's role and permissions encoded in the token.

## 3.2. Software structure

The software is structured into two main modules: the server module and the GUI module. The server module is responsible for handling server-side logic and it is the core of the project. While the GUI module focuses on presenting a user-friendly interface and handling user interactions.

The server software, named **operations**, contains both the Scheduling and the Flowgraph-based TT&C backends:

- **operations**
  - **src**: This folder holds all the files of the main program.
    - **api**: In this folder, the HTTP routes for the different services are defined, including the authentication mechanisms, along with various methods necessary for their proper functioning.
    - **pass\_scheduling**: This folder contains the implementation of the HTTP routes, database queries, and methods related to the Scheduling service.
    - **flowgraph**: This folder contains the necessary files for flowgraph editing and visualization. It includes HTTP routes, database queries, and threads for managing WebSockets used in editing and executing flowgraphs.
    - **execution**: This folder contains the code for the Execution Manager, Utility, and Background blocks, along with a subfolder that contains the code for the Mission blocks of each satellite mission to be operated. Additionally, the code for the different missions is organized into separate folders, each containing the necessary functions adapted to the specific characteristics, protocols, and requirements of each satellite.
    - \* *main.rs*: This file contains the code responsible for initializing the different services offered by the program. It includes the initialization of the database connection, starting the server, and the execution manager.
    - \* *lib.rs*: This file defines several modules that encapsulate different functionalities of the software.
  - **tests**: In this folder, all the tests are grouped together, including unit tests, integration tests, and any other tests related to the software.
  - \* *gs.conf.toml*: This configuration file defines the endpoints of the interfaces for the different GSs.
  - \* *Cargo.toml*: Configuration file that specifies various aspects of the package, including its name, version, dependencies, build settings, etc.

The frontend module, named **operations\_egui**, has the following structure:

- **operations\_egui**

- **api**: This folder contains a library that implements all the API endpoints. It is used for the frontend to request information from the server and interact with it.
- **frontend**: This folder contains the code of the GUI.
  - **assets**: This folder contains the static assets such as images and js files used in the frontend.
  - **src**: This folder contains the frontend source code.
    - **componets**: This folder contains reusable User Interface (UI) components that are used throughout the application. It also includes the different pages and views of the application, providing the necessary components for rendering and displaying the user interface.
  - \* *main.rs*: It is the main entry point of the frontend application.
  - \* *lib.rs*: This file is responsible for managing the views and pages to be displayed.
  - \* *app.rs*: This file handles the application's state and manages the responses received from the server.
  - \* *utils.rs*: This file contains styles, visuals, color palettes, and themes used in the application.
  - \* *index.html*: Is the main HyperText Markup Language (HTML) file that serves as the entry point for the frontend application, when it is compiled for web application.
  - \* *Cargo.toml*: The manifest file specifying dependencies and build configurations.
- \* *Cargo.toml*: The manifest file for the frontend module.

In addition to these modules, there is an auxiliary library is used to share functionalities, utilities, and resources between the server and GUI modules. It includes models built with Cap'n Proto for communication between the server and user interfaces, as well as functions for DateTime format conversions. This approach simplifies development, reduces code duplication, and ensures consistency throughout the software, as these functions are used in various parts of both modules.

The library named **schemas** has the following file structure:

- **schemas**

- **models**: This folder holds all the Cap'n Proto models used in the program.
  - \* *auth.capnp*: This file contains the bodies of the messages exchanged during the login process.
  - \* *configure\_flowgraph.capnp*: This file contains the request bodies used during the message exchanges in the WebSocket for configuring the flowgraphs.

- \* *execute\_flowgraph.capnp*: This file contains the request bodies used in the WebSocket for executing the flowgraphs.
  - \* *ground\_station.capnp*: This file contains the models related to the GS requests.
  - \* *satellite.capnp*: This file contains the models related to the satellite requests.
  - \* *schedule.capnp*: This file contains the models used for the Scheduling API.
- **src**: contains the autogenerated code for the models defined in the Cap'n Proto files, as well as the "**lib**" file.
- \* *lib.rs*: This file serves as an entry point for utilizing the Cap'n Proto models and includes the necessary import statements to make these modules accessible within the project. Additionally, it provides a collection of methods that simplify datetime manipulation tasks. These methods handle operations such as converting datetimes between UTC and local time, parsing datetime strings into objects, and formatting datetimes into desired string representations.
  - \* *auth.capnp.rs*
  - \* *configure\_flowgraph\_capnp.rs*
  - \* *execute\_flowgraph\_capnp.rs*
  - \* *ground\_station\_capnp.rs*
  - \* *satellite\_capnp.rs*
  - \* *schedule\_capnp.rs*
  - \* *build.rs*: Script executed during the build process. Responsible for compiling the Cap'n Proto files located in the "**models**" folders and generating the corresponding Rust code in the "**src**" folder.
  - \* *Cargo.toml*: The manifest file for the Rust package.

### 3.2.1. API Implementation

One of the primary threads within the server is the API thread, which has been implemented using the Axum framework. The API thread acts as the entry point for client requests, processing incoming HTTP requests and generating appropriate responses. It handles routing, ensuring that each request is directed to the corresponding endpoint based on the defined routes. The endpoints are responsible for executing the necessary logic to fulfill the client's request and generate the desired response. The following routes had been defined:

- */auth*: This route handles the authentication, it contains the endpoint to perform the login and consequently obtain an authentication token.
- */schedule*: This route is responsible for managing the scheduling of satellite missions and tasks. It provides endpoints to do all the operations related to the satellites, GS, and schedule passes.

- `/flowgraph`: This route is responsible for handling operations related to the flowgraph configurations. It provides various endpoints to facilitate the management of flowgraph-related tasks within the TT&C software.
- `/ws/configuration/flowgraph_id`: This route provides WebSocket endpoints for real-time configuration of the flowgraph identified by the parameter.
- `/ws/execution/norad_id`: Similar to the previous route, this route provides WebSocket endpoints for interacting with the execution of a specific satellite identified by its NORAD ID.

A more detailed analysis of each route, method, message body, and access level can be found in Appendix A. Most of the functions that handle operator requests involve reading or writing data from the database. However, some functions have more complex functionality, such as pass prediction, upload mission blocks, and the functionality of the WebSocket. The following sections will provide more detailed explanations of these functionalities.

#### 3.2.1.1. *Pass schedule*

In Section 2.1.3.1., the different options available for predicting passes were discussed, including One pass, Multiple passes, Multiple passes from a specific date, All passes until a specific date, and All passes within a time interval. When making a pass prediction request using the RequestPasss Cap'n Proto model, the operator needs to provide the NORAD ID of the satellite they want to predict passes for, along with their desired prediction option.

The data required for the prediction depends on the selected option. For example, if the operator chooses the Multiple passes from a specific date option, they would need to provide the number of passes and the start date for the prediction. In addition to the required data, there are also optional parameters that can be provided. One parameter is the maximum search duration, which is only necessary when the prediction does not have an end date, if this parameter is not provided, a default value of 48 hours is used for the calculation. Another optional parameter is the forced save flag. When conflicts arise between the predicted passes and the saved passes, the default behavior is to not save the predicted passes. However, if the forced save flag is set, only the passes that do not have conflicts will be saved.

Once the request is received by the server, and as long as the operator has a valid token with the role of Satellite Operator, the server performs several steps. First, it queries the database to retrieve the TLE data of the satellite. If the satellite exists in the database, the server proceeds to determine the compatible GS and obtains its coordinates.

Using the TLE data, the server employs the Rust library called **sgp4**<sup>14</sup> to retrieve the Keplerian elements of the satellite's orbit, as described in Section 1.2.3.2.. It then utilizes the SGP4 model to propagate the orbit, enabling accurate predictions of the satellite's position and velocity over time.

Based on the specified start time provided in the request body or using the current time, depending on the chosen option, the orbital elements of the satellite are propagated to that specific moment in time. By propagating the orbital elements, the position and velocity of the satellite are calculated in the ECI coordinate system.

---

<sup>14</sup>SGP4 Docs: <https://docs.rs/sgp4/latest/sgp4/>.

To determine if the satellite is visible from the GS at that particular moment, the process described in Section 1.2.3.3. is executed. Initially, a conversion from ECI to ECEF coordinates takes place. This conversion takes into account the rotation of the Earth and its orientation in space. Once the satellite's position is in the ECEF coordinate system, further conversion is carried out to obtain the Look Angles. Look Angles coordinates provide the azimuth (horizontal angle), elevation (vertical angle), and slant range (distance) of the satellite relative to the GS, enabling the determination of whether the satellite is within the line-of-sight of the GS's antenna.

This process is continued iterated by incrementing the propagation time in one-second intervals until the satellite's elevation angle with respect to the GS becomes positive. At this moment, it is considered that the satellite is in line-of-sight of the GS. The next step is to identify when the satellite will no longer be visible. This occurs when the elevation angle becomes negative. The objective is to determine the specific window of time during which communication with the satellite can take place. This process of predicting visibility and communication windows is repeated until reaching either the specified end time or the requested number of passes.

Once all the passes have been computed, the next step is to check if any of them conflict with the already stored passes. If no conflict is found, the predicted passes will be stored in the database. However, if a conflict occurs, the passes will only be saved if the force save flag has been requested. In this case, only the passes that do not have any conflicts will be saved.

After completing this process, a response containing all the resulting passes is sent back to the operator. This response provides the operator with a list of all the predicted passes and the conflicts found if any.

### 3.2.1.2. Upload Mission Blocks

The program has been designed to be as modular and adaptable as possible in order to operate different satellites with minimal, simple, and fast configuration. In some cases, as mentioned before, it is not always possible to achieve complete modularity because certain satellites require the use of specific protocols for their communications. Therefore, new code must be added to support those satellites. However, in other cases, the code for mission blocks can be used for multiple satellites. The only difference between operating with one satellite or another will be the commands to be sent, as the packet structure and communication protocols are common among these satellites. That's why it is necessary to be able to upload files with the appropriate configuration. These files are in Tom's Obvious, Minimal Language (TOML)<sup>15</sup> format, tailored to each satellite, and they specify the names and identifiers of the commands and other specifications for their proper functioning.

In the following example, a fragment of the TOML configuration file for mission blocks is shown for the 3Cat4 satellite. This file is sent in the request body along with the NORAD ID to specify the desired configuration. The first line specifies that the configuration to be executed for this satellite is the one classified as CubeCat4, and it includes the protocols and message formats required for executing its flowgraph. The subsequent lines define the names and identifiers of the commands, indicating whether they are *send command* or *send file command* types. Additionally, the *option* field can be used to specify any special

---

<sup>15</sup>TOML official website: <https://toml.io/en/>

execution requirements for a block. For instance, the "UPDATE\_TIME" block should include the timestamp of the last packet received by the satellite and the actual timestamp of the GS as the message content. This special behavior is indicated by assigning a number to the *option* field specifying the processes to be followed in this special case. These processes are defined for each different configuration.

```
[CubeCat4]

[[CubeCat4.send_command]]
    name = "HELLO"
    id = 0

[[CubeCat4.send_command]]
    name = "UPDATE_TIME"
    id = 1
    option = 0

[[CubeCat4.send_file_command]]
    name = "MANAGER_SET_CONF"
    id = 12
```

### 3.2.1.3. Configuration WebSocket

The main objective of the configuration WebSocket is to facilitate real-time control of updates and configurations made by the operator on a specific flowgraph. Simultaneously, these changes are reflected in the graphical interfaces of other operators who are also editing the same flowgraph. This serves as a central hub for managing configuration updates, the server allows immediate visibility and synchronization, empowering users to work in collaboration and coordination for the configuration of the flowgraph.

The server-side implementation of WebSockets establishes and maintains a continuous connection by dedicating a separate thread to each user. Each thread is further divided into two parts: one exclusively for sending messages and the other for receiving user requests. These tasks are executed independently.

When a user sends a request, it is received by the receiving thread, which processes the request and performs the necessary tasks. The result of these actions needs to be transmitted not only to the thread responsible for transmission but also to all users who want to receive updates on the flowgraph. To facilitate this, a communication channel is established to enable the notification of changes to all interested parties. To achieve this multi-producer, multi-consumer communication, a broadcast channel is used.

This channel is created when the server starts and is stored in the program's state to be accessible by all clients. Since the channel is common to all clients, every message carries an identifier indicating the corresponding flowgraph to which the message applies. This allows the transmission thread to filter the messages and send only the relevant information to each client. The broadcast of changes allows for immediate visibility and synchronization, empowering users to work together efficiently on the flowgraph. The server's role in facilitating real-time communication and dissemination of configuration updates enhances the overall editing experience and ensures consistency across all connected clients.

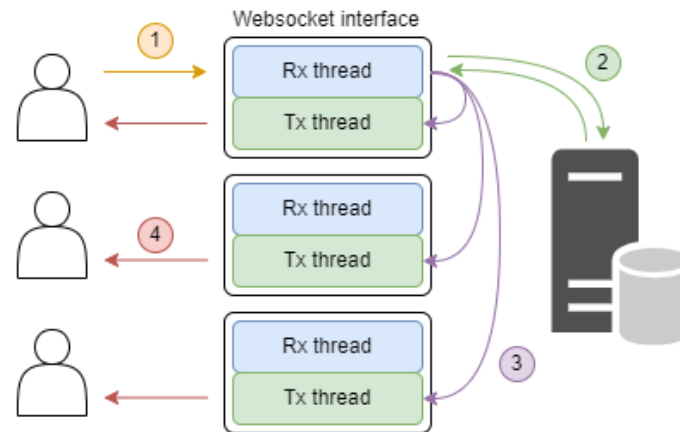


Figure 3.5: (1) The operator sends an update. (2) The server checks and stores the information. (3) The update is sent to the transmission thread. (4) The updates are sent to the operators.

The actions that can be performed using this functionality can be grouped into different categories based on their objectives:

- **Load data:** This group includes two messages, the *Load Flowgraph* and *Mission Blocks*. These messages are sent directly from the server to the client once the WebSocket connection has been established, providing the necessary information for the client to start working with the flowgraph. The Load Flowgraph contains all the necessary information to load and display in the client interface the saved configuration of the flowgraph. It includes details such as the nodes, connections, settings, and parameters of the flowgraph. While the Mission Blocks message contains a collection of commands and blocks that can be configured for the satellite.
- **Update configuration:** This group contains all actions related to the dynamic editing of the flowgraph. It includes operations such as inserting new blocks, creating connections between them, deleting blocks or connections, moving blocks within the graph, configuring block commands, uploading payload content for block commands, and setting timeouts. When the server receives a configuration update request, it validates the provided data to ensure its correctness. Once validated and saved the changes, the server broadcasts the modifications to all users who are interested in the particular flowgraph. This ensures that the graphical interfaces of all connected users reflect the updated configuration in real-time.
- **Configure flowgraph modes:** This group contains actions related to the configuration of the modes of execution of the flowgraph mentioned in Section 2.2.2., such as adding a new flowgraph to the satellite's configuration, deleting an existing flowgraph, or updating an existing flowgraph.
- **Control:** This group comprises a single message known as *Block edition*. This message is specifically intended to block the edition of the flowgraphs when communication with the satellite is initiated. This message prevents any errors that may occur due to parameter modifications during execution. Also, this message aware the operators of the imminent start of satellite communication so that they can make the necessary preparations for it.

#### 3.2.1.4. Execution WebSocket

For the Execution WebSocket, the functionality becomes more complex. In the previous scenario, the operator would send a message, it would be processed, the changes would be saved in the database, and all interested users would be notified. However, in this case, the actions performed have direct implications for the execution flow of satellite communication.

The main objective of this WebSocket is to enable real-time monitoring of the processes occurring during satellite communication. This allows the operator to actively control and interact with the execution while simultaneously visualizing the ongoing processes on their graphical interface. Therefore, in this case, it is not the reception thread of the WebSocket that is responsible for notifying other interested operators. Instead, it is the thread executing the satellite's flowgraph that processes the requests and generates the messages to be sent by the transmission threads of the WebSockets.

The messages that can be sent through this WebSocket can be classified into the following groups:

- **Load data:** This group includes messages that facilitate loading the graphical representation of the flowgraph onto the operator's interface. It consists of the *Load Flowgraph* message, which contains all the blocks, connections, and configurations, and the *Info Time* message, providing information about the timing of AOS and LOS events. These messages are sent by the server once the communication is established.
- **Events:** This group comprises messages that are sent by the server when an event occurs, which should be displayed on the operator's interface to keep them informed about the ongoing processes. The messages in this group include *Start Block*, *End Block*, *Send Log*, and *Download File*. The *Download File* message is sent when a new file is generated as a result of an execution event, like the reception of a scientific experiment result, and the operator can download it for further analysis.
- **Execution control:** These messages allow interaction and control over the ongoing execution. The *Req Manual Next Block* message is sent by the server when a block completes its execution, and there are multiple connected blocks on one of its output pins. In response, the operator needs to provide a *Res Manual Next Block* message indicating which configured block should be executed next. Additionally, some messages enable stopping the automatic execution of the flowgraph and switching to manual mode. The *Manual Execution* message is used to transition between automatic and manual modes, while *Manual Block* creates a block to be executed manually, and *Delete Manual Block* removes a block from the waiting list. Finally, the *Finish* message indicates the end of the execution, triggered either by the operator's request or the occurrence of the LOS event.

### 3.2.2. TLE automatic update Implementation

There is a second thread in the server's execution responsible for updating the TLE of the satellite at regular intervals over 48 hours. To accomplish this, the server checks which



satellites have the automatic update flag enabled. For each of these satellites, a request is made to the Celestrak API<sup>16</sup> to retrieve the updated TLE data. Regular TLE updates ensure an up-to-date view of the satellite's orbit and guarantee that the prediction of the passes, the tracking of the satellites, and communication operations will be performed with the most accurate information available.

The query to obtain the TLE information from Celestrak is the following, by replacing the {norad.id} placeholder with the correct five-digit NORAD ID of the satellite [25].

```
https://celestrak.org/NORAD/elements/gp.php?CATNR={norad_id}&FORMAT=TLE
```

The response of Celestrak's API includes on Line 0 the name of the satellite, while Lines 1 and 2 contain the first and second lines of the TLE data, respectively.

### 3.2.3. Execution Manager Implementation

The thread containing the Execution Manager can be considered the software's core, as it houses the most critical components of its logic. It is responsible for overseeing and managing the execution of satellite flowgraphs. Based on the information inputted by operators through the API and the scheduled communications, the Execution Manager takes charge of resource allocation and task planning. This involves coordinating the efficient utilization of execution threads dedicated to each satellite, optimizing the distribution and allocation of resources, and ensuring that there are no conflicts or blocks in the concurrent execution of flowgraphs.

At the program's start, the Execution Manager queries the OpGen database for scheduled passes of the satellites, selecting the one that will occur earlier. Based on this information, it generates a thread for each satellite, which remains in a waiting state until the communication starts time. Additionally, it listens for notifications whenever there are modifications to the pass schedule, ensuring that there is always a thread prepared to execute the communication for the next upcoming pass of each satellite. Simultaneously, it informs the operators about the satellites that are being executed and provides them with the channels through which they can monitor and interact with the execution process.

The satellite thread operates by waiting until the execution start time is reached, which is configured to be three minutes before the satellite enters line-of-sight. This three-minute window takes into consideration the operator's preparation time and the antenna alignment process, as this duration allows the operator to adequately prepare for the execution, and additionally, it provides sufficient time for the GS antenna to align it with the satellite's position and ensuring an optimal line-of-sight connection, according to the procedures explained in Section 1.2.3.3..

The first step is to inform the Execution Manager that the satellite communication is about to take place so that it can notify the operators and lock the flowgraph configuration for that satellite. Simultaneously, the flowgraph to be executed is selected, taking into account the configured modes and preferences specified in the requirements (see Section 2.2.2.).

Along with selecting the flowgraph, the AOS block is obtained, serving as the initial point for executing the mission blocks and establishing the chain of connections. In the case that the satellite does not have any configured flowgraph, the same procedure is followed

---

<sup>16</sup>Celestrak webpage: <https://celestrak.org/>

anyway. Obtaining the coordinates of the assigned GS for the pass and the satellite's TLE from the database, the Tracking Block calculates the required antenna movements. It generates a file that includes timestamps, azimuth angles, and elevation angles, which serve as guidance for aligning the antenna with the satellite, and this file is transmitted to the GS using a ZMQ socket.

At this point, two scenarios can occur. If the satellite doesn't have any configured flowgraph, the AOS block is executed followed by the LOS block. Once these are completed, the Execution Manager is informed to proceed. This ensures the reservation of necessary resources for satellite communication from another software.

In the case where the satellite has a configured flowgraph and communication is to be carried out from this software, four threads are initialized. Firstly, the main execution thread is responsible for executing the different mission blocks. Secondly, the ZMQ interface thread establishes a connection with the GS, enabling the sending and receiving of packets to and from the satellite. The backup thread collects all events and execution details to generate files for later analysis. It also stores past events so that operators viewing the execution in real-time can access previous occurrences. Lastly, the LOS trigger thread executes the LOS block, which starts once the AOS event occurs.

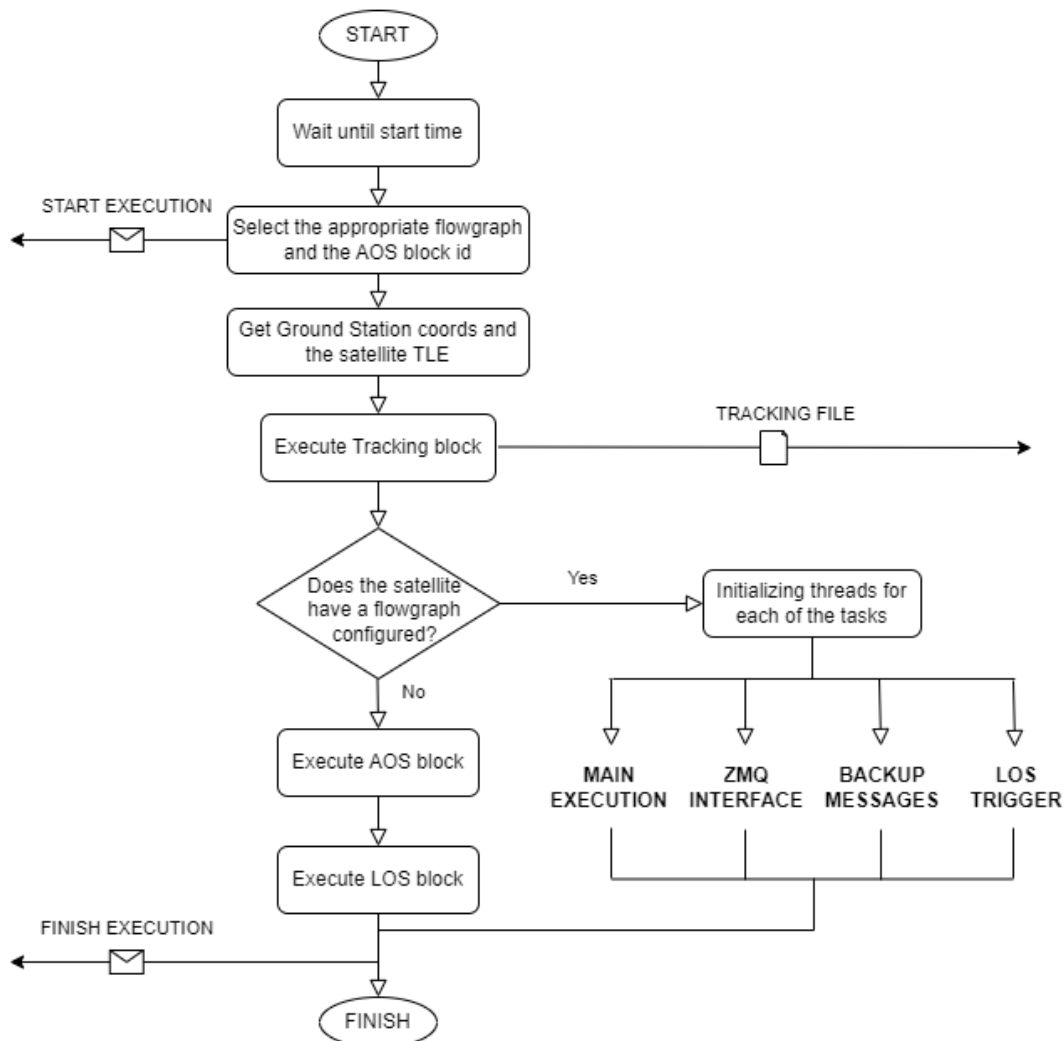


Figure 3.6: Satellite thread task flow.

### 3.2.3.1. Main execution

This thread is responsible for sending commands to the satellite based on the operator's configuration and processing incoming packets from the satellite. The procedures to be followed depend on the specific satellite being executed. These procedures are predefined during the satellite configuration and rely on the appropriate structure and protocols for successful communication with the satellite.

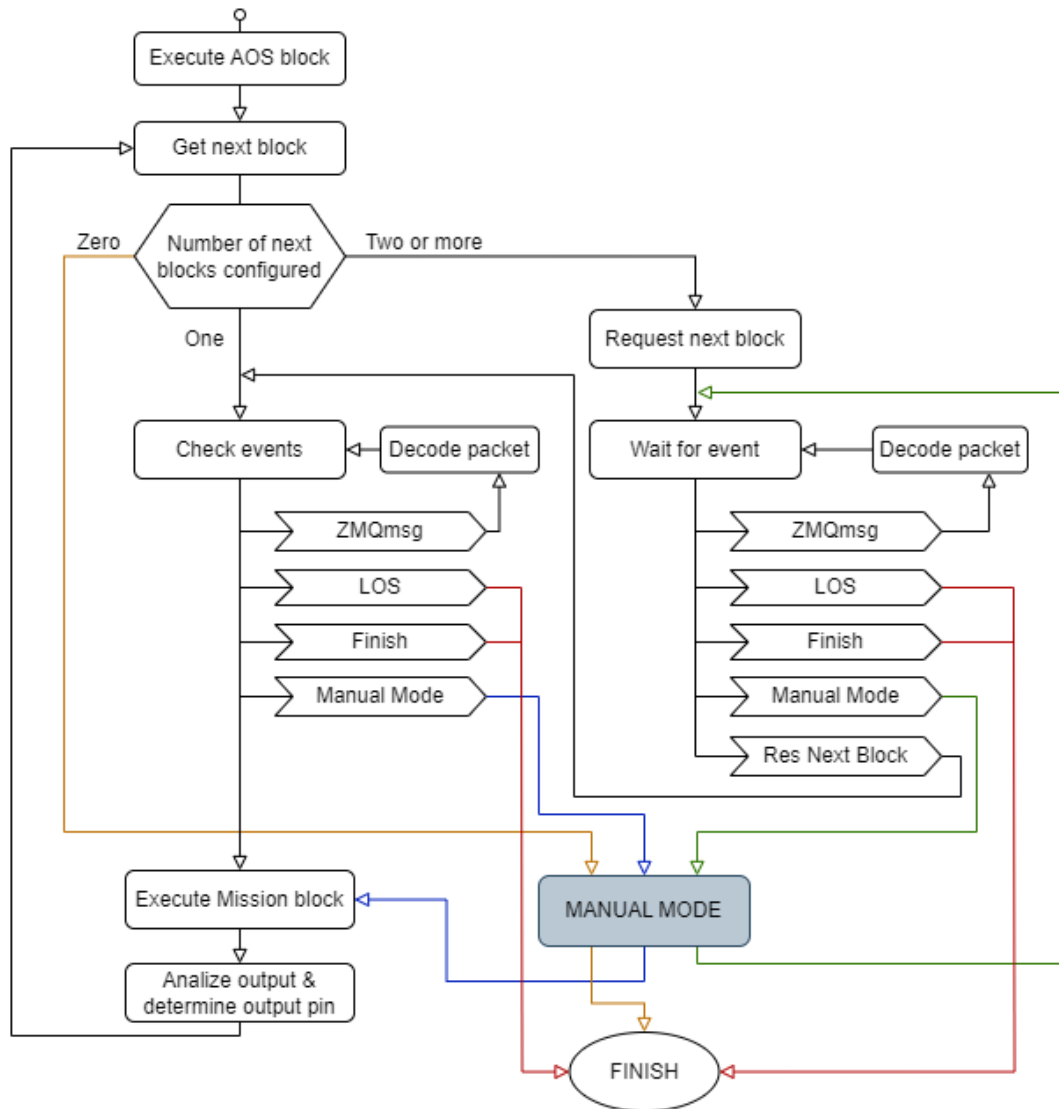


Figure 3.7: Main execution thread task flow.

The first task of this thread is to execute the AOS block. Using the satellite's TLE data and the GS's coordinates, it determines the exact moment when the satellite becomes visible, which means the start of the communication. Once this event occurs, a notification is sent to the thread responsible for executing the LOS block, signaling that it can initiate its execution. This thread takes charge of monitoring the satellite's visibility window from the GS and accurately determines when the satellite moves out of range.

Simultaneously, the database is consulted to determine the next block to be executed. There are three possible scenarios: no blocks are configured for continuous execution,

resulting in manual satellite operation; a single block is configured, allowing for automatic execution of the flowgraph; or multiple blocks are configured, requiring the operator to select the block to proceed with execution.

In the case of multiple configured blocks, the thread waits for operator commands specifying which block should be executed. While waiting, other events can occur, such as receiving a packet from the satellite, the LOS event, or the operator deciding to finish the execution or switch to manual mode. If a packet is received, it needs to be decoded before resuming the waiting state. Furthermore, the occurrence of the LOS event or the finish event indicates the end of the communication process.

Once the command specifying the block to be executed is received, it is validated to ensure that it is one of the eligible candidates. The execution process then proceeds as if only a single block were configured. Before executing the block, a check is performed to ensure that no LOS, Finish, or Change to Manual Mode events have occurred, and that there are no pending messages that need to be processed. After these verifications, the selected mission block is executed.

Depending on the output generated by the block (success, failure, or timeout), the corresponding output pin is selected, and the process repeats by consulting the database for the next block.

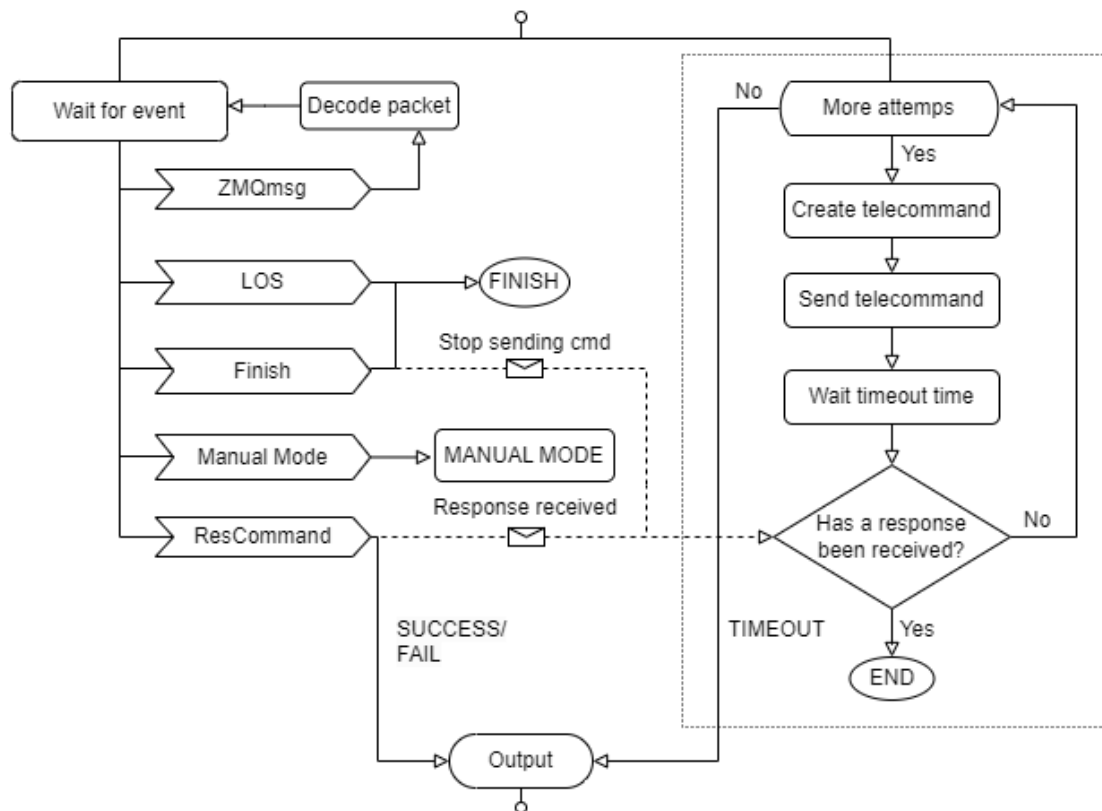


Figure 3.8: Mission Block task flow.

When executing a mission block, it is crucial to handle the telecommand transmission effectively. In the event of not receiving a response from the satellite, whether due to a lost packet or any other communication issue, the telecommand should be resent based on the operator's configuration. This configuration includes the number of retry attempts

and the time interval to wait for a response. If no response is received within the specified attempts and time interval, the execution of the block is considered failed.

Additionally, some checks are performed to verify that the communication has not been terminated due to LOS or Finish events, as these events indicate the end of communication and signal the need to quit further execution or switch to Manual Mode. Furthermore, it is required to decode all messages sent by the satellite until a response to the sent command is received. This decoding process ensures that the received data is properly interpreted and processed, enabling appropriate actions to be taken based on the information received.

To handle these tasks, one thread is dedicated to managing timeouts and command retries, while another thread handles the reception of ongoing events.

At the end of the mission block execution, three possible scenarios can occur: success, if the satellite's response matches the expected outcome; fail, if the response does not meet the expected criteria; and timeout, if no response is received within the specified attempts and time interval.

When the main thread transitions to manual execution, it starts executing mission blocks at the operator's request. These blocks are configured during the ongoing communication. Operator requests are stored in a queue and executed in the order they are received. To facilitate this process, a dedicated thread is assigned for creating and sending telecommands, similar to the automatic execution of mission blocks. While another thread is responsible for receiving events.

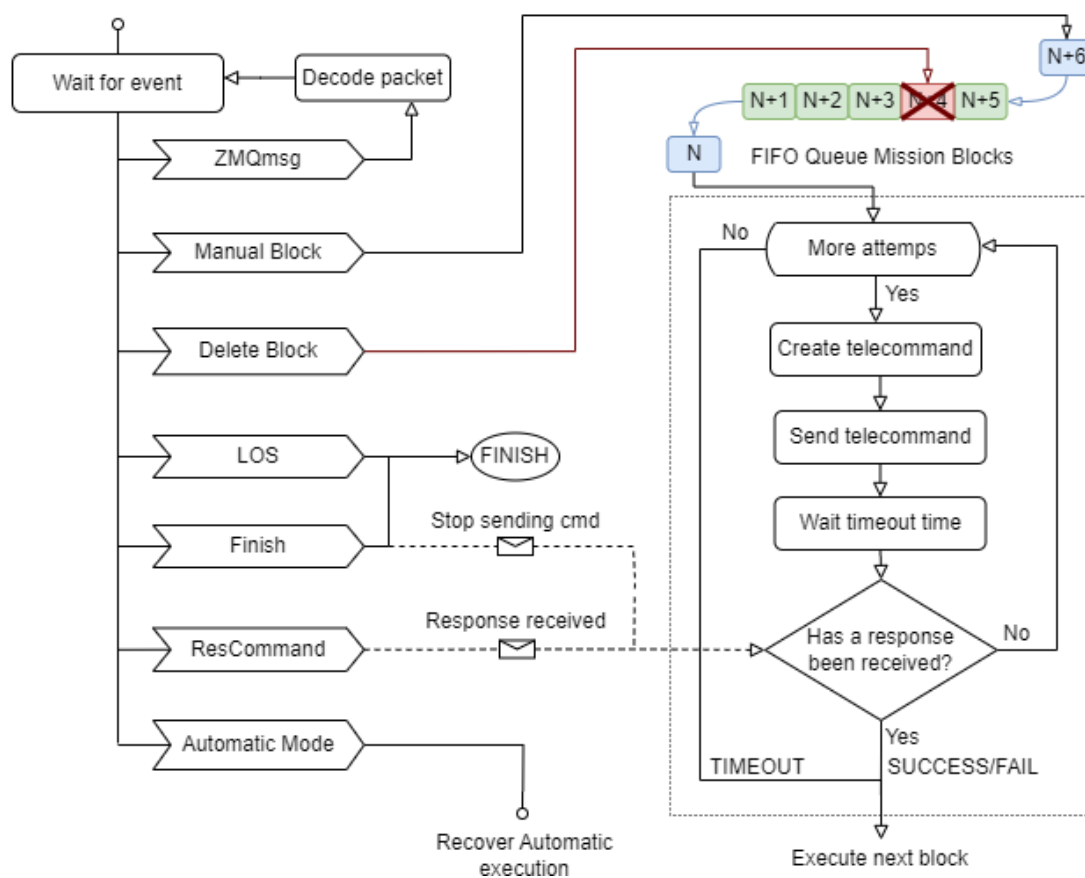


Figure 3.9: Manual Mode task flow.

After each mission block is completed, the next block in the queue is executed until either automatic execution is restored or communication is terminated. The communication can be terminated either by the operator or when the satellite is no longer visible (LOS event).

### 3.2.3.2. ZMQ Thread

Parallel to the main thread, the ZMQ thread is dedicated to establishing and maintaining communication with the GS. This allows the transmission of telecommands from the Op-Cen to the antenna for reception by the satellite, and vice versa. The ZMQ interfaces are configured with a PUSH-PULL pattern, with the server located on the GS side. This interface type accepts only a single client and server on each side of the channel. Therefore, it is crucial to occupy the channel only during satellite communication and keep it free the rest of the time for other purposes. Hence, the connection is established and closed at the beginning and end of the flowgraph execution.

Since ZMQ PUSH-PULL interfaces are unidirectional, a channel is set up for sending telecommands from OpCen to the GS and another channel for receiving telemetry from the GS to OpCen. Each channel operates on a different port. Since the transmission of these messages is independent of each other, separate threads are dedicated to perform both tasks.

The thread is responsible for receiving telemetry waits to receive packets sent by the GS. Its function is simple: receiving a packet of bytes and notifying the main thread to decode it based on the satellite protocols and take appropriate actions. On the other hand, the thread responsible for sending telecommands receives packets from the main thread and sends them to the GS. Additionally, it remains alert to the completion of the flowgraph execution to close the ZMQ connection releasing the channels for future communications.

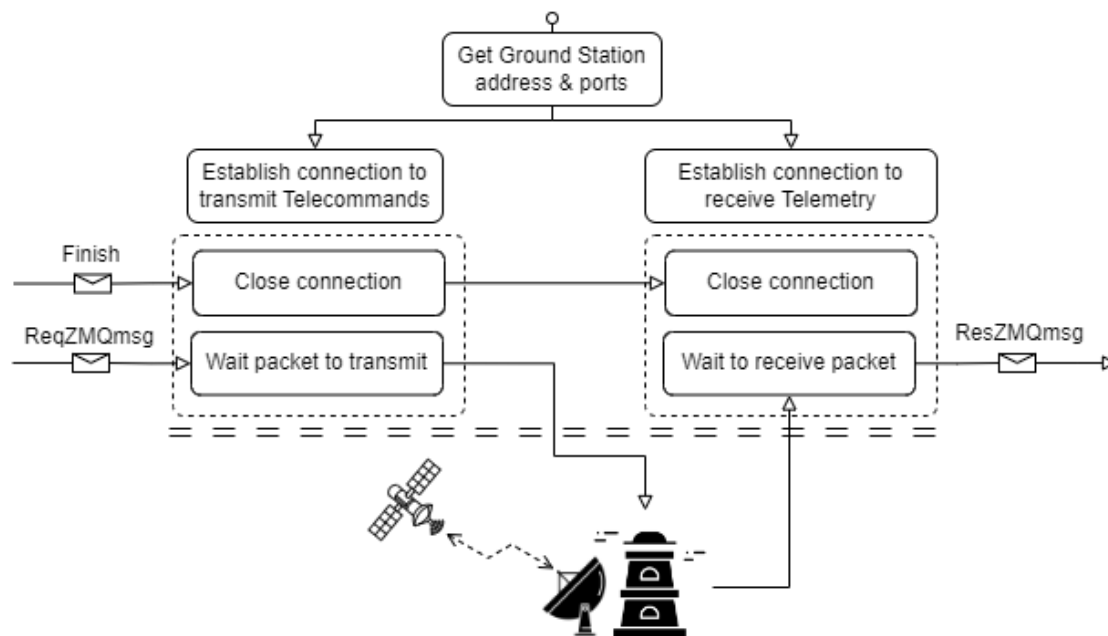


Figure 3.10: ZMQ thread task flow.

### 3.2.3.3. Backup Thread

The backup thread serves two main purposes. Firstly, it collects and records all events that occur during the execution, allowing them to be stored and analyzed later, such as telemetry, experiment results, or execution logs.

Secondly, the backup thread provides support to the operator when they initiate a connection with the server. It informs the operator of all events that occurred before their connection, enabling them to reproduce the events in their GUI. Additionally, the backup thread provides the communication channel through which the operator can intervene and interact with the flowgraph execution. This allows the operator to actively participate in the execution process and make real-time adjustments as needed.

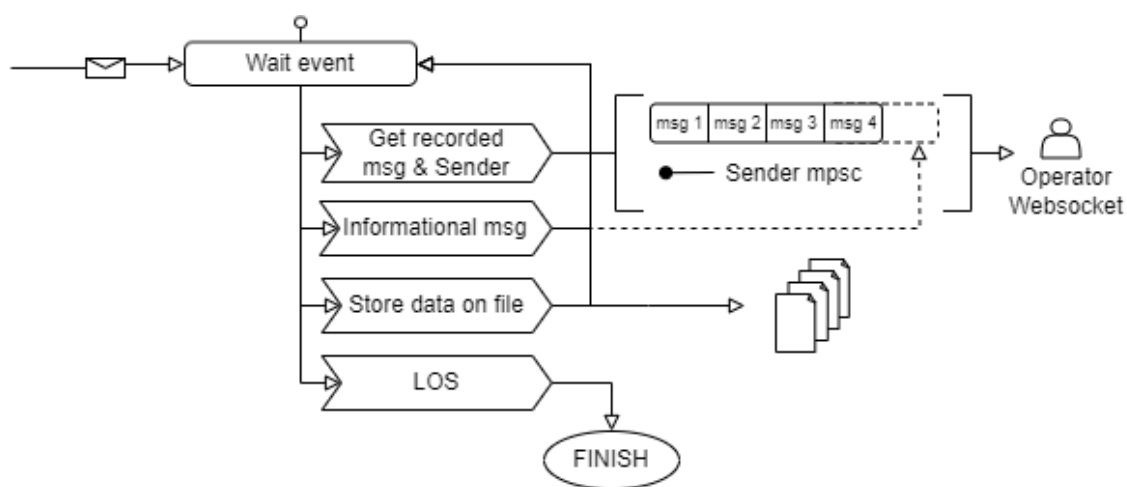


Figure 3.11: Backup thread task flow.

### 3.2.3.4. LOS Thread

In the LOS thread, the utility block responsible for handling the loss of sight events is executed. This thread starts its execution once the AOS event occurs. When the satellite goes out of the range of visibility, it notifies all other threads that the communication has ended, allowing them to conclude their executions and free up the resources they were utilizing for the next communication.

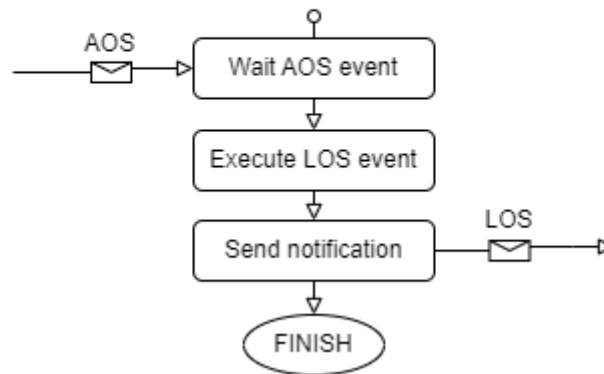


Figure 3.12: LOS thread task flow.

### 3.2.4. Inter-Process communication

Throughout the explanation of the implementation, it has been mentioned that the server consists of multiple threads to accommodate the program's requirements, and information exchange between them is essential for the project's proper functioning. This necessitates the establishment of communication channels between various software components, as described in Section 3.1.1.1.. Two distinct types of messages can be identified: notification messages (NotifyMsg) and execution messages (ExecutionMsg). Notification messages serve to interact with the Execution Manager, including notifying updates on the OpCen database, responding to operator inquiries regarding the satellite execution state, and providing notifications about the execution status of a flowgraph. On the other hand, execution messages are specifically designed to control the execution processes of specific flowgraphs.

The NotifyMsg messages can be classified into three categories:

- **Update notification:** These messages are sent from the Database thread when an operator requests a specific action through the API, which results in a modification in the database that directly affects the satellites controlled by the Execution Manager. These are NextPass, DeletePass, DeleteSatellite, and UpdateNorad and were explained in depth in Section 2.2.3.
- **Operator messages:** This group includes the ReqExecuting and ResExecuting messages, which are exchanged between the operator connected via WebSocket and the Execution Manager. The ReqExecuting message is used to ask about the execution status of a satellite, and the Manager responds accordingly. If the satellite is currently executing or the execution event has started, the Manager informs all connected users by sending a ResExecuting message.

In addition, there are the ReqChannelFG and ResChannelFG messages. Once the operator has been notified that a satellite is executing, the operator's WebSocket



thread requests information about the specific flowgraph that is being executed and the channel through which communication can be established with it. The ResChannelFG message provides the operator with the requested information, allowing them to interact with the ongoing execution.

- **Execution notification:** This group includes the StartExecution and FinishExecution messages. These messages are sent from the satellite execution thread to inform the Execution Manager about the initiation and completion of the satellite communication process. The StartExecution message is sent when the communication with the satellite begins, providing information about the selected flowgraph that will be executed. It serves as a notification to the Execution Manager that the execution process has started for a particular satellite. On the other hand, the FinishExecution message is sent when the communication with the satellite is completed and consequently, the Execution Manager marks the current pass as finished and initiates a new thread for executing the next pass of the satellite.

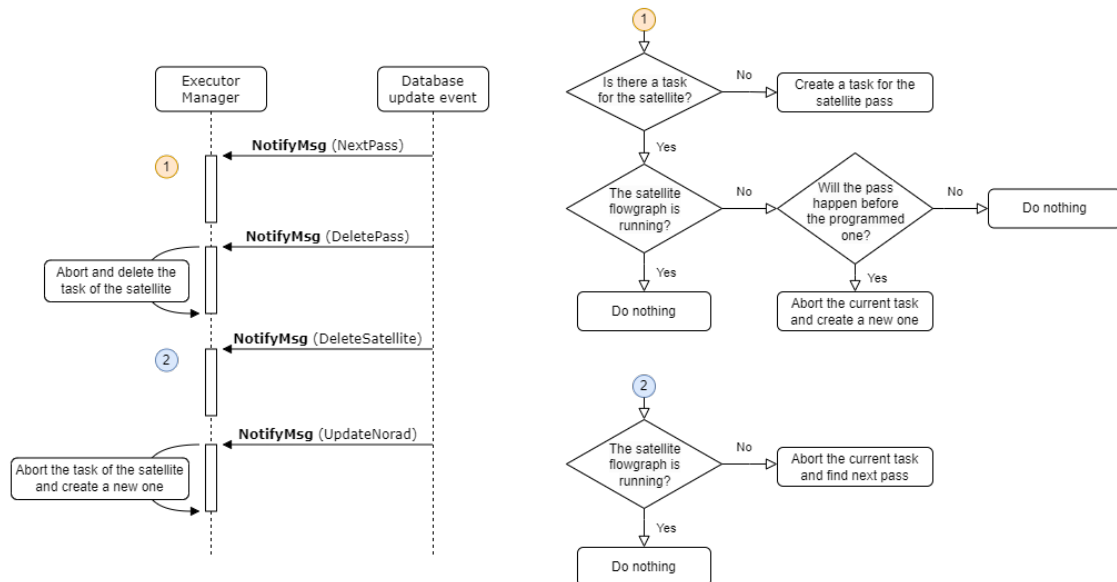


Figure 3.13: Exchange of Update notification messages and the resulting behavior between the Database thread and the Execution Manager

In the example in Figure 3.14, the operator first queries if a specific satellite is being executed, and the Execution Manager responds that it is not. Then, the communication with the satellite is initiated, and this event is notified by the Satellite thread to the Execution Manager. The Execution Manager, in turn, informs the operator of this event. Subsequently, the ChannelFG messages are exchanged, allowing the operator to start monitoring the execution of the satellite task.

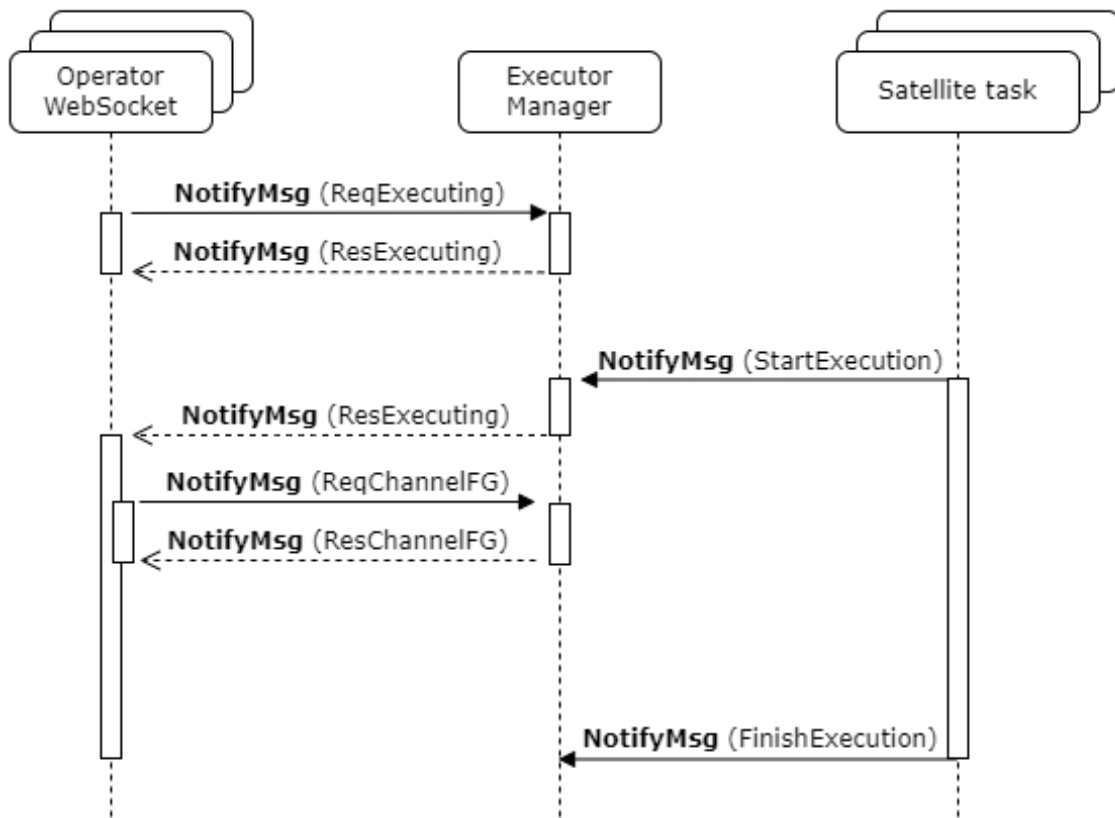


Figure 3.14: Message exchange between the Operator WebSocket, Execution Manager, and Satellite thread.

To facilitate communication and enable the transmission and reception of information from various components, it should be set up as a broadcast channel. This allows multiple threads, including those serving operators through the API and those dedicated to satellite tasks, to send information to the Execution Manager. The Execution Manager can then relay relevant information to all connected operators via WebSockets. Using a single channel for these communications simplifies the tasks of the Execution Manager, as it only needs to listen to the channel and respond accordingly when receiving a message.

On the other hand, `ExecutionMsg` can be classified into three groups. The first group is intended for visual and monitoring purposes, allowing operators to see the ongoing processes in their graphical interface. The second group is used to control and interact with the execution, while the last group comprises messages sent from various satellite task threads to inform about events that affect the normal flowgraph execution. These messages are sent through different channels based on their origin and purpose.

- **Informational messages:** These messages are generated by the main thread of the satellite. These messages include informative text to be displayed in the GUI (`LogMsg`), indicating the start or end of block execution (`StartBlock` and `EndBlock`), requesting the selection of the next block to execute (`ReqNextBlock`), or providing predictions of AOS and LOS events. These messages are received by connected operators through WebSockets and by the backup thread.

This channel is created by the Execution Manager, allowing operators to easily ac-

quire it through message exchange of type `NotifyMsg`, specifically using `ReqChannelFG` and `ResChannelFG`. This process is necessary because these channels are unique to each thread and dynamically created. Initially, operators only have access to the communication channel that enables them to communicate with the Execution Manager. Furthermore, when an operator establishes their connection with the flowgraph execution, they send a message through the same channel to the Backup thread (`ReqExecution`). In response, via a dedicated mpsc channel established between the operator and the backup thread, it transmits a list of previously received messages and the sender endpoint, enabling interaction with the main satellite execution.

- **Intervention messages:** These messages influence the flowgraph execution through operator intervention, and can be sent by different operators connected via the Web-Socket interface. These messages are exclusively received and processed by the main flowgraph thread. Therefore, they are sent through an mpsc channel, and the sender information is stored in the backup thread, allowing operators to access them when their connection is established. The operations that can be performed through this channel include sending a `ResNextBlock` message in response to a `ReqNextBlock`, indicating the desired next block for execution. Also, operators can send a `ManualMode` message to switch between automatic and manual execution modes, a `ManualBlock` message containing the configuration of a mission block to be executed, or a `Finish` message to terminate the flowgraph execution and communication with the satellite.
- **Control messages:** Control messages are sent from the thread of the flowgraph. These messages, just like informational ones, are transmitted through the same broadcast channel. Despite sharing the channel, it doesn't pose any issues because each endpoint will only process the messages that are relevant to them while disregarding the rest. The control messages include `ReqZMQ` and `ResZMQ`, used to exchange messages sent to or received from the satellite between the flowgraph execution thread and the ZMQ interface; `ResCommand`, which is received when a ZMQ message is a response to a sent telecommand; `Timeout`, triggered when the mission block being executed has reached the maximum number of telecommand resend attempts; `ITelemetry` or `HTelemetry`, received when new telemetry is obtained from the satellite; `ExperimentFile`, received when the result of an experiment is obtained; and finally, `LOSMsg`, which indicates that the satellite is no longer visible to the GS, resulting in the termination of communication.



applications tend to have better performance compared to those generated with WASM. This is because the browser needs to interpret and translate the code, which can impact performance and resource consumption.

Desktop applications can directly access system resources and take advantage of the full power of the user's device, including CPU, memory, and graphics processing. This direct access often results in better performance and responsiveness compared to web-based applications.

The answer to which option is the best for the software interface is that there is no definitive answer. Both options offer different advantages, making them suitable for certain parts of the project and not as ideal for others. The only difference when programming the application for either option lies in how API calls to the server are made. In the case of WASM, JavaScript is used for API calls, while in the desktop application, calls are made directly from the application.

This supposes the need to adapt the way that the calls are made while the rest of the logic and layout of the application keep the same for both options. By adding conditional compilation tags `#[cfg(not(target_arch = "wasm32"))]` or `#[cfg(target_arch = "wasm32")]` to the code, it can be specially tailored to any target architecture. This flexibility in development is why the decision was made to create a graphical interface for both options.

In this way, the web application allows the Scheduling interface to be accessed by multiple users without the need for a standalone executable. However, for the TT&C software, it is recommended to use the desktop application as it can provide better performance due to the complexity of its operations.

The GUI has been designed in a way that all the information displayed in different views is stored in the application's state, making it accessible from any component. All responses received from the server as a result of operator requests are centrally handled to update the information stored in the app's state and, consequently, update the content of the views reactively. By centralizing response handling, the application ensures efficient and synchronized updates to the app's state, thereby reflecting real-time changes in the displayed views.

When launching the application, the initial view presented is the login page, where the operator can enter their credentials. In the event of incomplete or incorrect information, an error message is displayed, ensuring operators are promptly informed. Once successful validation of the credentials, a splash screen is displayed while the necessary information is fetched from the server.

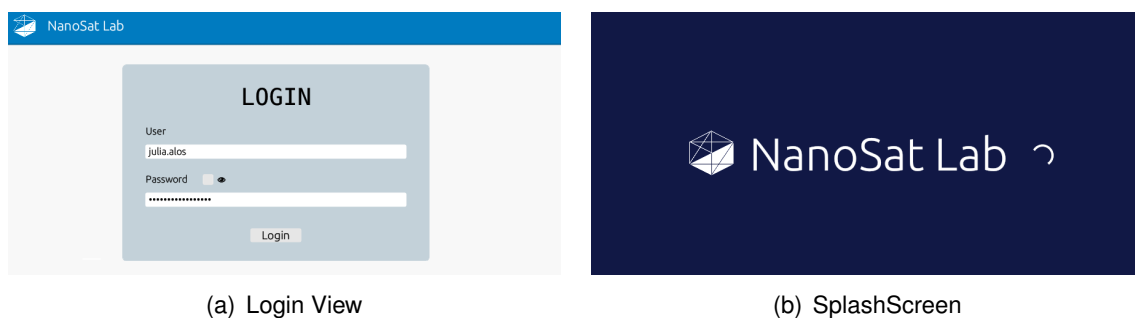
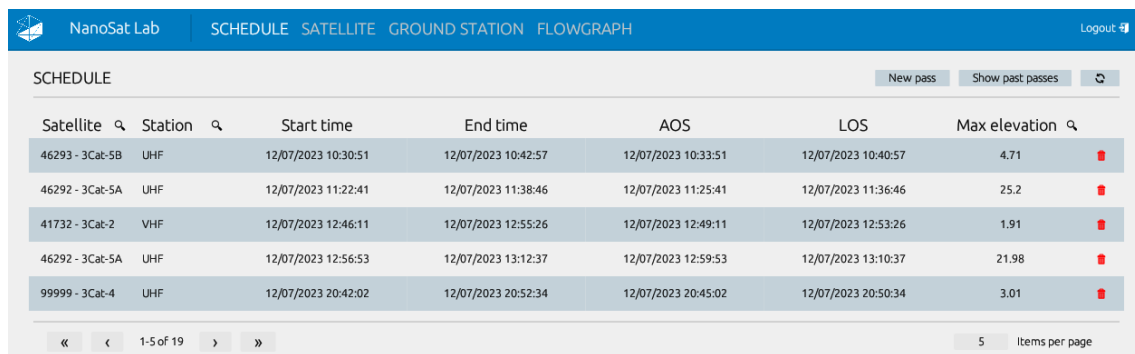


Figure 3.16: Login and SplashScreen Views

After the loading process, a screen is displayed with a navigation bar at the top, providing access to different components of the Scheduling module, such as the scheduler, satellites, and GS, as well as the TT&C module.

The initial screen presents a table, as shown in Figure 3.17, that displays all the scheduled passes. Operators can utilize the interface to program new passes and delete any passes that are not required. Additionally, the table provides filtering options, enabling operators to filter the passes based on specific criteria such as satellite name or NORAD ID, ground station name, or maximum elevation by specifying a threshold.

At the bottom of the table, operators can adjust the number of items to display per page and navigate through different pages of the table.




The screenshot shows the 'SCHEDULE' tab in the NanoSat Lab interface. It features a table with columns: Satellite, Station, Start time, End time, AOS, LOS, and Max elevation. There are search icons for Satellite and Station, and a search icon for Max elevation. The table lists five scheduled passes for satellites 46293, 46292, 41732, 46292, and 99999. At the bottom, there are navigation controls showing '1-5 of 19' items and a '5 Items per page' setting.

Satellite	Station	Start time	End time	AOS	LOS	Max elevation
46293 - 3Cat-5B	UHF	12/07/2023 10:30:51	12/07/2023 10:42:57	12/07/2023 10:33:51	12/07/2023 10:40:57	4.71
46292 - 3Cat-5A	UHF	12/07/2023 11:22:41	12/07/2023 11:38:46	12/07/2023 11:25:41	12/07/2023 11:36:46	25.2
41732 - 3Cat-2	VHF	12/07/2023 12:46:11	12/07/2023 12:55:26	12/07/2023 12:49:11	12/07/2023 12:53:26	1.91
46292 - 3Cat-5A	UHF	12/07/2023 12:56:53	12/07/2023 13:12:37	12/07/2023 12:59:53	12/07/2023 13:10:37	21.98
99999 - 3Cat-4	UHF	12/07/2023 20:42:02	12/07/2023 20:52:34	12/07/2023 20:45:02	12/07/2023 20:50:34	3.01

Figure 3.17: Programmed passes View


From the same screen, operators can program new passes by clicking on the "New Pass" button. This action opens a window, as shown in Figure 3.18, where operators can schedule a pass with the multiple options explained in Section 2.1.3.1..

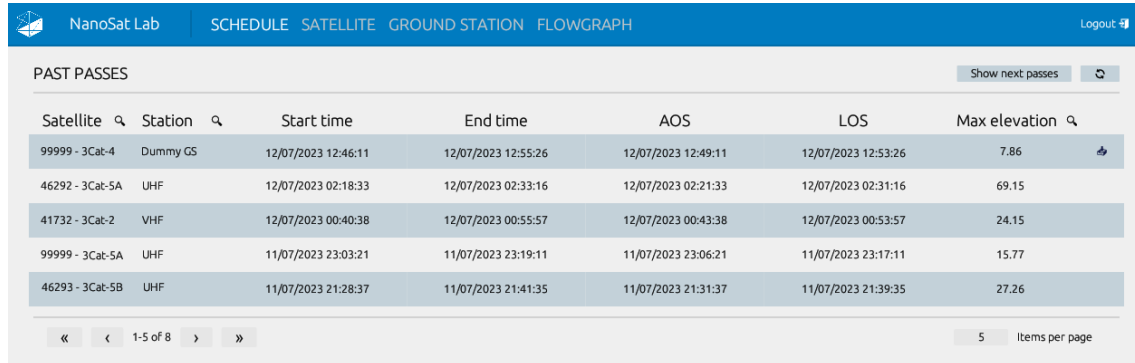


The figure shows six different configurations of the 'New pass 3Cat-4' scheduling window:

- (a) One Pass: Satellite: 99999 - 3Cat-4, Type: One pass, Default max time [48 hours], Number of passes: 3, Initial date: 2023-07-12.
- (b) Multiple Passes: Satellite: 99999 - 3Cat-4, Type: Multiple passes, Number of passes: 3, Initial date: 2023-07-12.
- (c) All Passes Within Time Interval: Satellite: 99999 - 3Cat-4, Type: All passes time interval, Start date: 2023-07-12, End date: 2023-07-12.
- (d) Multiple Passes from Date: Satellite: 99999 - 3Cat-4, Type: Multiple passes from, Number of passes: 3, Initial date: 2023-07-12.
- (e) All Passes Until Date: Satellite: 99999 - 3Cat-4, Type: All passes until date, Final date: 2023-07-12.
- (f) Date Picker: A calendar view for July 2023, with the 12th selected.




Figure 3.18: Scheduling modes

At the same time, if the operators want to view the completed passes, they can click on the "Show past passes" button, and the table will display the completed passes, as shown in Figure 3.19. In the case that any of these passes generated files during communication with the satellite, it will be indicated by the  icon. By clicking on this icon, operators can download a zip file containing all the associated files.



Satellite	Station	Start time	End time	AOS	LOS	Max elevation
99999 - 3Cat-4	Dummy GS	12/07/2023 12:46:11	12/07/2023 12:55:26	12/07/2023 12:49:11	12/07/2023 12:53:26	7.86
46292 - 3Cat-5A	UHF	12/07/2023 02:18:33	12/07/2023 02:33:16	12/07/2023 02:21:33	12/07/2023 02:31:16	69.15
41732 - 3Cat-2	VHF	12/07/2023 00:40:38	12/07/2023 00:55:57	12/07/2023 00:43:38	12/07/2023 00:53:57	24.15
99999 - 3Cat-5A	UHF	11/07/2023 23:03:21	11/07/2023 23:19:11	11/07/2023 23:06:21	11/07/2023 23:17:11	15.77
46293 - 3Cat-5B	UHF	11/07/2023 21:28:37	11/07/2023 21:41:35	11/07/2023 21:31:37	11/07/2023 21:39:35	27.26

Figure 3.19: Past passes View

Through the navigation bar, operators can access the Satellite Management page. On this page, all the satellites configured by the operator are displayed, as presented in Figure 3.20. The operator can perform various actions on this page, including adding a new satellite by clicking on the green card and entering its details, modifying the NORAD ID, or uploading a configuration file for a specific satellite using the  icon, and editing or deleting a satellite using the  and  icons respectively.

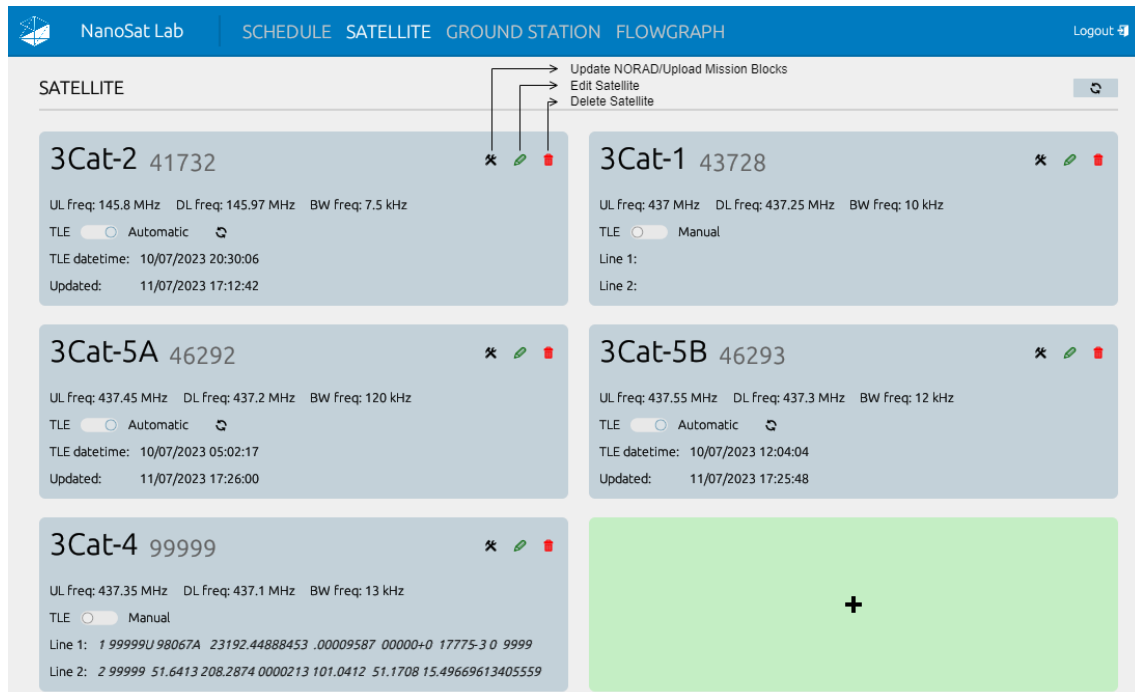


Figure 3.20: Satellites View

As before, operators can access the GS Management page through the navigation bar. On this page, all the registered GS are displayed, including their basic information such as location and operational frequency ranges. Additionally, operators have the option to view and/or modify the status of the GS using a toggle switch, as shown in Figure 3.21.

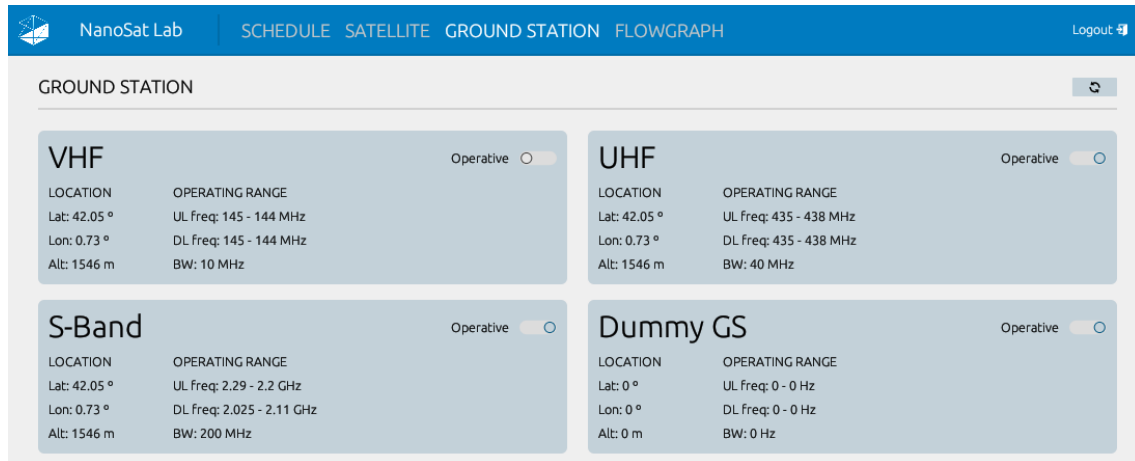




Figure 3.21: Ground Stations View

Lastly, operators can access the page dedicated to the TT&C module, where a list of all satellites with configured flowgraphs is displayed. From this page, operators can view upcoming passes for a specific satellite and the selected flowgraph for execution, by clicking in the  icon, as shown in Figure 3.22. When a flowgraph is about to start or is currently running, a  icon is displayed, providing access to the execution visualization.

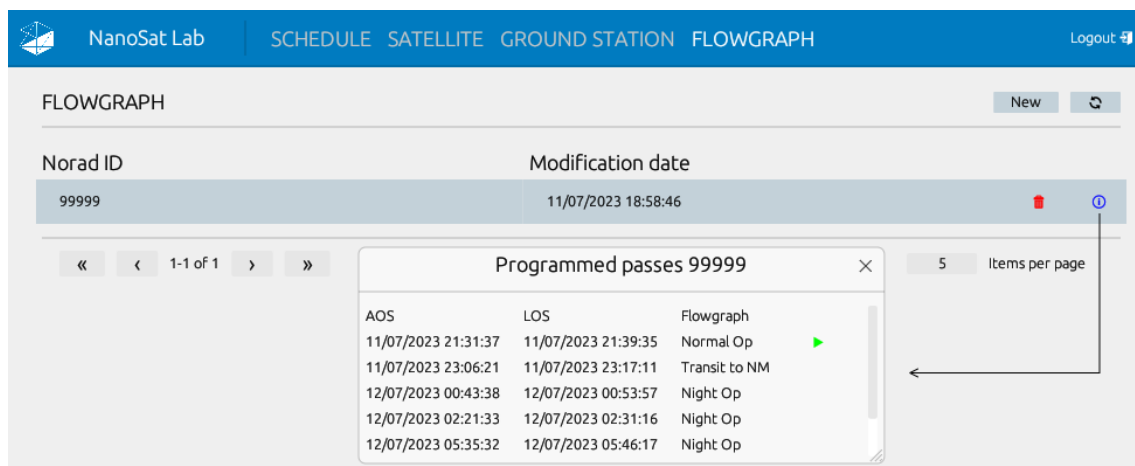


Figure 3.22: Flowgraph menu View

By selecting one of the satellites displayed on the list, the flowgraph editor page opens, as shown in Figure 3.24. The top left of the screen displays the NORAD ID of the edited satellite, along with the date and time of the last update and the remaining time until the satellite's execution begins. This window also provides information about the currently configured flowgraph, and provides access to a menu that lists all the flowgraphs associated with the satellite, from where can be modified their configurations, or create new ones, as illustrated in Figure 3.23.



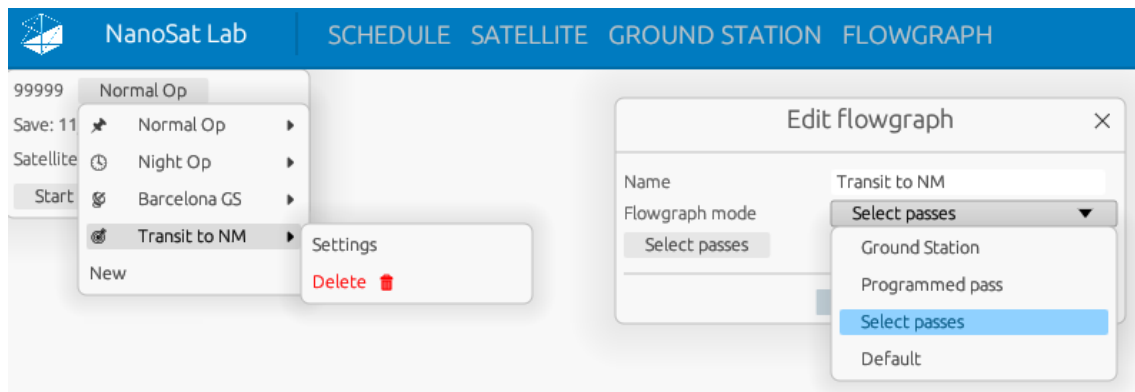


Figure 3.23: Flowgraph Modes Configuration

On the other hand, the top right of the screen displays the scheduled passes for the selected satellite and provides information about the time window during which the satellite will be in line-of-sight, as well as the associated flowgraph that will be executed during that pass.

The flowgraph editor page provides operators with the necessary tools to configure the sequence of blocks that will be executed during the mission. Operators can easily add new blocks, customize commands, upload payload files for the commands, interconnect the blocks, and perform various other actions.

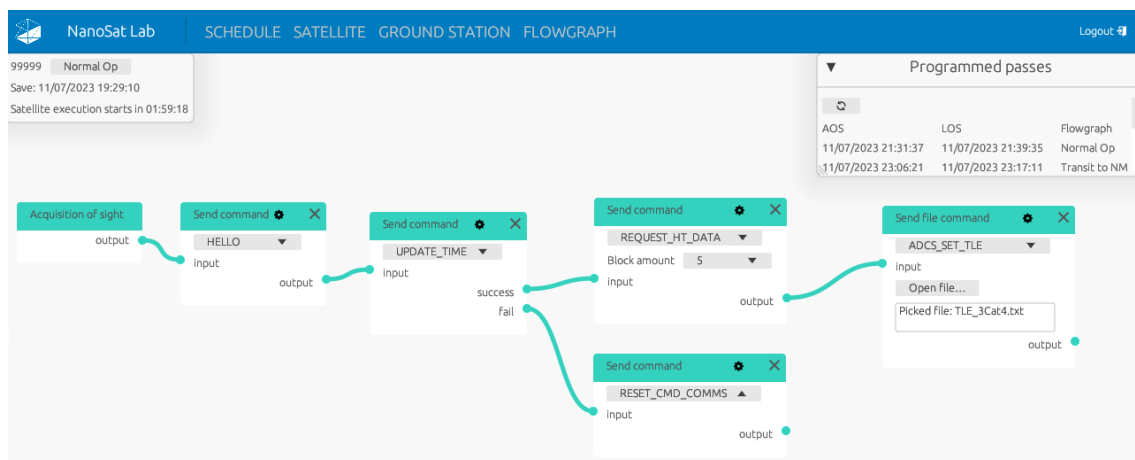


Figure 3.24: Flowgraph configuration View

During an ongoing communication session with the selected satellite, operators will have access to the execution control screen, depicted in Figure 3.25. This screen offers a real-time view of the running flowgraph, allowing operators to monitor and track the performance of the flowgraph. Executed blocks are visually distinguished by a green color, while blocks currently in progress are highlighted in yellow and unexecuted blocks are presented in gray, providing a clear visual representation of the flowgraph's execution progress.

Furthermore, the top-right window provides access to the manual mode controls, from which operators are allowed to temporarily halt the execution process and send messages manually, the logs screen, and additionally, operators can also download files generated during the communication with the satellite.

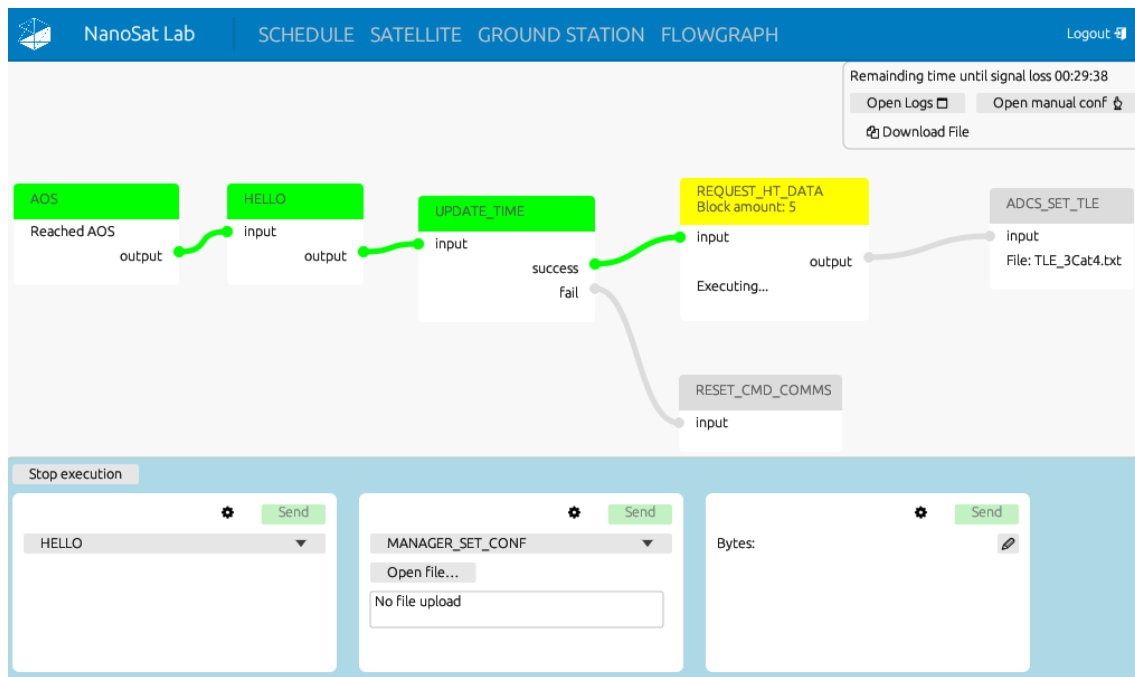


Figure 3.25: Flowgraph execution View

The logs window, shown in Figure 3.26, displays all server-generated messages. These messages include information about the currently executed block, received messages from the satellite, telemetry data, and other relevant events. Operators can filter the messages based on their tags or content, allowing for efficient navigation and analysis of the log information.

```

Output logs
Filter [ ] [ ] DateTime TAG

> [INFO] Acquisition of sight -> Start communication
> [INFO] -----> Telecomand HELLO <-----
> [TRACE] Send app packet: App Header -> Timestamp 1689320721 [64b0fd11], ID 4 [04], Len 16 [10], CRC 3631562925 [d87540ad]
--> Telecomand ID: 0 [00], len: 0 [00]
--> contents: []
> [INFO] App packet received
> [INFO] **** Packet received APP_LAYER_ID_ASYNC ****
> [TELEMETRY] Comms:
{"type":"as","time":1689320723,"comms":{"actual_rssi":-88.0,"app_layer_received":524,"app_layer_sent":26169,"bad_auth_commands_received":45,"boot_count":53,"cmd_received":127,"config_chksm":98,"config_packets_received":0,"ext_temp":35.4375,"freq":437.350016,"int_temp":34.44140625,"last_cmd_id":1,"last_config_packet_id":0,"last_lqi":61.417322834645674,"last_rssi":-58.5,"ll_rx_packets":71,"ll_tx_packets":372,"phy_rx_err":102,"phy_rx_packets":127,"phy_tx_err":0,"phy_tx_packets":2075,"transmitted_power":51,"unknown_cmd_received":2}}
> [INFO] -----> Command processed OK
> [DEBUG] END BLOCK output -> output

```

Figure 3.26: Execution Logs Window

## 3.4. Deployment

For the deployment of the project, the decision has been made to utilize Proxmox VE<sup>19</sup>. Proxmox VE is an open-source server virtualization environment that enables the deployment and management of virtual machines and containers. It leverages Linux Containers (LXC), an OS-level virtualization method that allows for the execution of multiple isolated applications within a shared Linux kernel. Unlike fully virtualized machines, containers utilize the same kernel as the host system, resulting in a lightweight and efficient virtualization solution.

Moreover, deploying the project within an LXC provides additional benefits such as enhanced security and isolation. The container acts as a boundary between the application and the host system, preventing conflicts and dependencies, and ensuring a more secure execution environment. This approach offers a streamlined and efficient deployment process while maintaining the necessary isolation and security for the project's execution.

The steps to follow for the deployment are quite simple. First, a web browser must be opened and navigated to the URL `https://proxmox-IP-address:8006/`. From there, a container image of the desired operating system is downloaded, in this case, Ubuntu. Next, a new container should be created providing the necessary configuration details, such as the container name, container image, disk size, and RAM allocation. Once the container has been created, it just needs to be started.

To run the project, several dependencies need to be installed. The first one, of course, is Rust, which is required for compiling the project. Additionally, SQLite must be installed to store the contents in the database, ZMQ is necessary for establishing the connection with the GSs, and Cap'n Proto is needed for compiling the data schemas used in the API.

Furthermore, to deploy the web application, the WebAssembly compilation target and Trunk, a WASM web application bundler for Rust, must also be installed. Finally, all that's left is to clone the code repository from GitHub and compile the server and web app.

In summary, the commands needed to set up the environment are the following ones:

```
$ sudo apt update
  Install Rust:
$ curl --proto '=https' --tlsv1.3 https://sh.rustup.rs -sSf | sh
  Install SQLite:
$ sudo apt install sqlite3
  Install ZMQ:
$ apt install libzmq3-dev
  Install Cap'n Proto:
$ sudo apt install capnproto

Webassembly compilation target:
$ rustup target add wasm32-unknown-unknown
  Install Trunk:
$ cargo install trunk
  Egui required dependencies:
$ sudo apt-get install libxcb-render0-dev libxcb-shape0-dev
  libxcb-xfixes0-dev libxkbcommon-dev libssl-dev
```

---

<sup>19</sup>Promox Virtualization: <https://www.proxmox.com/en/proxmox-ve>

```
Clone GitHub repository:
$ git clone https://github.com/nanosatlab/operations.git
Build web app:
$ trunk build --release
Build OpCen software:
$ cargo build --release
Launch the server:
$ ./target/release/operations
```

## CHAPTER 4. CASE STUDY: <sup>3</sup>CAT-4

This chapter presents a specific study on the integration of the <sup>3</sup>Cat-4 satellite into the TT&C module. It focuses on the protocols defined for this satellite, the message format used for communication, and the specifications required to successfully integrate and operate the satellite using the designed software.

### 4.1. Application layer

As mentioned in previous chapters, the packets exchanged between the OpCen software and the endpoint of the GSs are transmitted through ZMQ channels, operating at the application layer.

These packets consist of an 11-byte header plus the payload data. The header incorporates a timestamp, which indicates the time at which the packet was generated, an ID that identifies the type of packet, the length field specifies the size of the content in bytes (excluding the header itself), and a CRC32 checksum for ensuring data integrity. All the header fields are encoded using big-endian byte order, in which the most significant bit is stored at the lowest memory address, while the least significant bit is stored at the highest memory address. Figure 4.1 depicts the structure of the packet header with each of its fields.

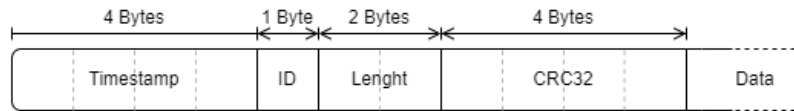


Figure 4.1: App layer packet format

#### 4.1.1. Type Application layer packet

The <sup>3</sup>Cat-4 protocol defines seven different types of messages, each identified by the ID field in the header. These message types serve different purposes, including sending commands, transmitting data, and providing telemetry information.

##### ***IT\_HT\_BEACON*** [ID = 0]

This message type is sent periodically by the satellite and contains Instantaneous telemetry along with a set of Historic telemetry. It provides detailed information about the current and past states of the satellite.

##### ***IT\_BEACON*** [ID = 1]

During the first stage of the satellite mission, this message type is sent periodically. As no Historic telemetry is available during this stage, it only provides Instantaneous telemetry data.

***R.HT*** [*ID = 2*]

This message type is used by the satellite to send Historic telemetry data in response to a telecommand sent by the GS.

***R.PAY*** [*ID = 3*]

This message contains scientific data, such as experimental results or specific measurements. It is sent by the satellite when requested by the GS, after performing the specified experiments.

***UL*** [*ID = 4*]

This message type is used exclusively for sending telecommands in the uplink direction. It allows the GS to send instructions and commands to the satellite for various purposes.

***ASYNC*** [*ID = 5*]

When the satellite responds to an uplink message, it uses this message type to send an asynchronous response back to the GS. The response includes COMMS telemetry, providing information such as system boot count, received signal strength indicators (RSSI), link quality indicators (LQI), transmitted power, packet counts, errors, temperatures, communication frequency, and command and configuration packet details.

***CHEKCKSUMS*** [*ID = 6*]

The checksums message payload contains a struct that holds 16-byte checksum values for various components. These checksums serve as a form of error detection and verification for the corresponding components, including the GNSS (Global Navigation Satellite System), AIS (Automatic Identification System), RAD (Radio), PNS (Positioning and Navigation Service) driver, OBDH (On-Board Data Handling), EPS (Electrical Power System), ADCS (Attitude Determination and Control System), and other on-board systems.

***AOCS.TEST.MODE*** [*ID = 7*]

This message type is specifically for conducting tests related to the Attitude and Orbit Control System (AOCS). It includes telemetry data related to the AOCS subsystem, providing insights into the magnetic field, gyroscope measurements, photodiode readings, quaternion estimations, control values, and others.

#### **4.1.2. CRC32 (Cyclic Redundancy Check)**

The CRC32 algorithm is a checksum algorithm used in communication protocols to ensure reliable data transmission. It calculates a checksum by performing polynomial division on the input data stream using a generator polynomial and bitwise XOR operations. The

sender computes the CRC32 checksum and appends it to the data stream for transmission. Upon receiving the data, the receiver performs the same polynomial division process using the received data and compares the calculated checksum to the received checksum. If they match, it indicates that the data was received without errors or corruption.

CRC32 is a preferred choice for ensuring data integrity because reproducing the same checksum without possessing the exact original data is computationally infeasible. Even a small change in the input data results in a completely different checksum due to bitwise XOR operations and bit propagation. The algorithm exhibits pseudorandomness, distributing bits uniformly and making the checksum highly unpredictable. The use of modulo-2 arithmetic and XOR operations further enhances the uniqueness of the checksum. These characteristics make CRC32 an effective and reliable method for detecting transmission errors and ensuring data integrity [22].

The process for computing CRC32 in <sup>3</sup>Cat-4 follows these steps: first, calculate the CRC32 for the header, excluding the last four bytes corresponding to the CRC32 itself. Then, use the resulting value as the seed to compute the CRC32 for the content. And finally, insert the resulting checksum into the header.

## 4.2. Telecommand

Telecommands are the instructions sent by the GS to control the satellites. As mentioned earlier, these telecommands are sent as the payload of an [UL](#) message. They comprise a 16-byte header followed by the command contents. The header of the telecommand is encrypted using the Advanced Encryption Standard (AES) algorithm to ensure the confidentiality and security of the commands sent from the Ground Station to the satellites.

The telecommand header consists of several important fields: Command Identifier, Command Counter, Padding, Length, and CRC32 checksum. The Command Identifier field identifies the type of command being sent and specifies the desired action or instruction for the satellite. The Command Counter field is used for acknowledgments (ACKs) and keeps track of the sent commands, ensuring that the satellite confirms their receipt and processing. The Padding field, consisting of 8 bytes, ensures that the header length is a multiple of 16, as required for encryption and proper alignment in encryption algorithms. The Length field indicates the size of the command contents, specifying the amount of data that follows the header. Lastly, the CRC32 checksum provides a mechanism to verify the integrity of the transmitted data by detecting errors or corruption in both the header and command contents. It is important to note that all the header fields are encoded using big-endian. Figure 4.2 depicts the structure of the command header with each of its fields.

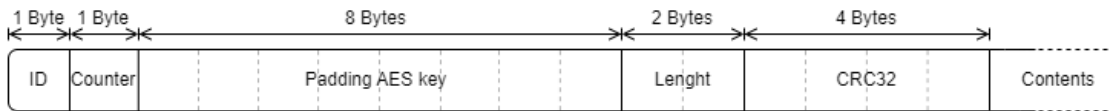


Figure 4.2: Command packet format

When a telecommand is received, the satellite responds with an [ASYNC](#) message, confirming that the telecommand has been successfully received.

### 4.2.1. AES Encryption

AES is a symmetric encryption algorithm. It operates on fixed-size blocks of data, typically 128 bits, and uses a secret key to perform encryption and decryption. AES employs multiple rounds of substitution, permutation, and mixing operations to transform the plaintext into ciphertext. The strength of AES lies in its key size, with options for 128-bit, 192-bit, and 256-bit keys. The encryption process involves applying mathematical operations to the data and key, creating a complex and secure transformation that is computationally difficult to reverse without the correct key. AES provides a high level of security and is recognized as a standard encryption algorithm for sensitive information [23].

In the case of <sup>3</sup>Cat-4, a 128-bit key is used to encrypt the header of the command. Before encryption, it is required to compute the CRC32 with both the header and the content, as it is included in the command header.

## 4.3. Telemetry

Telemetry transmitted by the satellite can be present in two different formats: Instantaneous Telemetry and Historic Telemetry.

**Instantaneous Telemetry** provides a complete snapshot of the satellite's current state. It includes a wide range of detailed information, such as sensor readings, system statuses, power levels, temperature measurements, and more. This real-time telemetry allows ground operators to monitor the satellite's health and performance in great detail. This type of telemetry is transmitted in messages of the **IT HT BEACON** and **IT BEACON** types. The telemetry payload within these messages has a fixed length of 308 bytes.

On the other hand, **Historic Telemetry** offers a condensed and simplified version of the telemetry data collected over a period of time. This format contains a summarized view of the satellite's past performance, and provides valuable insights into the satellite's long-term behavior, enabling operators to analyze its performance over extended periods, even in cases where there was no direct communication with the satellite. This type of telemetry is sent together with the instantaneous telemetry in an **IT HT BEACON** packet or separately upon operator request in an **R HT** packet.

To optimize data transmission and storage, Historic Telemetry is typically sent in blocks that contain a certain number of messages. These blocks are compressed using the POCKET+ algorithm, which reduces the data size without significant loss of critical information. When decompressed, these messages have a length of 111 bytes of information.

### 4.3.1. POCKET+ algorithm

POCKET+ is an efficient compression and decompression algorithm developed by the ESA specifically designed for compressing time series data generated for status monitoring. It is particularly well-suited for compressing fixed-length data structures, such as housekeeping telemetry data from spacecraft, while considering limitations in resources and bandwidth during communication between the spacecraft and the GS.

The compression process employed by the satellite involves several steps to optimize the



transmission and storage of data. Initially, when a new packet is generated, it undergoes an XOR operation with a reference packet. This operation allows for the separation of the bits into predictable and unpredictable categories based on the information provided in the mask packet.

Any predictable bit value that is not the same as the reference data value is immediately classed as nonpredictable (Negative Mask Update), while any non-predictable that has the same value as the reference data value over a tracking period will be classed as predictable in the next tracking period (Positive Mask Update).

The positive and negative mask updates are sent to the receiver in the form of counters, followed by the corresponding non-predictable bit values sent in the clear. Once the compressed packet is created, it can be transmitted or stored as required.

The mask is updated at every iteration and thus the data redundancy is constantly tracked. Unexpected changes are dealt with immediately but it takes at least one tracking period of consistent positive results to declare a bit predictable again. Thus the robustness of the system is increased while the performance and stability are kept.

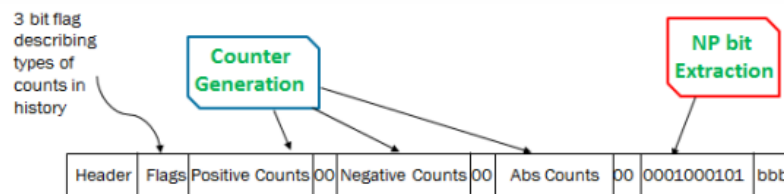


Figure 4.3: Compressed packet [24].

For decompression, the one that must be done in the OpCen software to obtain the telemetry data, a model of the mask is maintained and updated with the counters arriving in every packet. These can be positive updates and/or negative updates and/or absolute values. This mask is then used to indicate which bits in the non-predictable field of the compressed packet should be inserted into a copy of the reference packet. Thanks to the update rules the reference packet can be any packet of that type successfully received in the present or last tracking period [24].



# CHAPTER 5. TESTING AND VERIFICATION

The complexity of this project, coupled with the requirement for scalability, demands thorough testing to ensure the proper functioning of each component. These tests are not only crucial for guaranteeing the project's integrity but also serve to validate the preservation of existing functionality during future developments.

This chapter provides an overview of the testing procedures conducted in the project. It starts with unit tests, which focus on testing small functions. It then progresses to integration tests, which verify the behavior of more complex features. Finally, it includes verifications of proper functionality for all the software modules, including the GUI, and concludes with testing in a real-case scenario.

## 5.1. Unit tests

Unit tests are designed to assess small components or units of software. This approach is well-suited for most of the API operations of the Scheduling module, as they primarily involve CRUD operations (create, read, update, delete) on database instances. This type of testing has been employed to validate the behavior of functions responsible for accessing database resources and to ensure that any future developments do not impact the existing functionality.

During the design of these tests, two challenges were encountered. The first challenge was that all tests are executed in parallel, which impossibilities sharing a single database for all the tests, as using a single database could lead to interference between the tests. To overcome this challenge, a separate database is generated for each test, and it is deleted at the end of the test. To facilitate this approach, the **test-context**<sup>1</sup> library has been used, which provides a convenient way to manage the setup and teardown of resources specific to each test, ensuring isolation and avoiding interference between tests.

During the test setup phase, a database is created and populated with test data. The function under test is then executed, and the resulting operation is validated to ensure it produces the expected outcome. Finally, in the teardown phase, the test database is deleted to clean up resources and prepare for subsequent tests.

The second challenge arises from the fact that Rust does not natively support testing asynchronous code. However, this issue has been successfully resolved by utilizing the **async-trait**<sup>2</sup> library, which has made it possible to guarantee the correct behavior and functionality of asynchronous operations within the project.

## 5.2. Integration tests

Integration tests are designed to evaluate the interaction and integration between different components. These tests have been used to verify the correctness of the pass scheduling process. This process involves multiple steps, such as interpreting the type of request made by the operator, retrieving the satellite's TLE from the database, selecting a GS,

---

<sup>1</sup>test-context Docs: [https://docs.rs/test-context/latest/test\\_context/](https://docs.rs/test-context/latest/test_context/)

<sup>2</sup>async-trait Docs: [https://docs.rs/async-trait/latest/async\\_trait/](https://docs.rs/async-trait/latest/async_trait/)

```

Running tests/unit_tests.rs (target/debug/deps/unit_tests-3df409d99e66e33b)

running 10 tests
test tests::test_delete_pass ... ok
test tests::test_edit_satellite ... ok
test tests::test_delete_satellite ... ok
test tests::test_get_all_past_passes ... ok
test tests::test_get_all_gs ... ok
test tests::test_get_all_passes ... ok
test tests::test_get_all_satellites ... ok
test tests::test_update_satellite_norad ... ok
test tests::test_add_satellite ... ok
test tests::test_update_state_gs ... ok

test result: ok. 10 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 5.63s

```

Figure 5.1: Unit Tests results.

calculating the predicted passes based on the requested parameters, checking for conflicts with previously scheduled passes, and finally saving the predicted passes.

To verify that the obtained results in the pass prediction are as expected, the communications windows had been calculated using the Gpredict<sup>3</sup> program as a reference. A margin of error has been allowed to account for slight differences between both predictions. Additionally, different scenarios have been tested, including cases without conflicts where the passes should be saved, cases with conflicts, and extreme situations where the satellite will never be visible to the ground station.

```

Running tests/integration_tests.rs (target/debug/deps/integration_tests-dec29c6d8d9244d4)

running 11 tests
test request_pass_tests::test_all_passes_until_date_conflict ... ok
test request_pass_tests::test_all_passes_until_date ... ok
test request_pass_tests::test_all_passes_interval ... ok
test request_pass_tests::test_multiple_passes ... ok
test request_pass_tests::test_multiple_passes_conflict ... ok
test request_pass_tests::test_multiple_passes_date ... ok
test request_pass_tests::test_one_pass_conflict ... ok
test request_pass_tests::test_multiple_passes_date_conflict ... ok
test request_pass_tests::test_one_pass ... ok
test request_pass_tests::test_one_pass_time_over ... ok
test request_pass_tests::test_all_passes_interval_conflict ... ok

test result: ok. 11 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 8.33s

```

Figure 5.2: Integration Tests results.

### 5.3. Verification tests

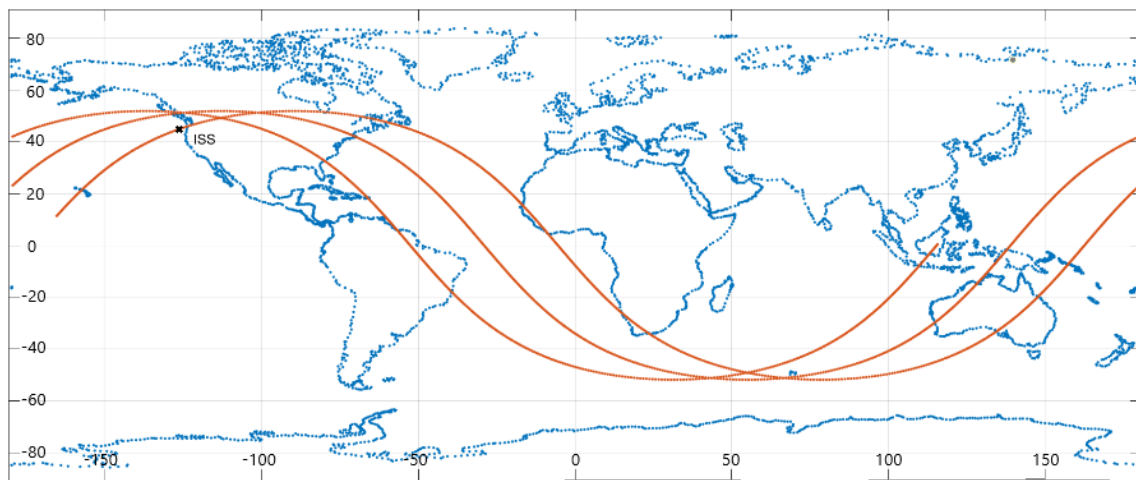
The verification tests have focused on the core functionalities of each module in the OpCen software. For the scheduling and TT&C modules, the tests have aimed to validate the accuracy of the tracking and orbital prediction functions. In addition, for the TT&C module, the tests have also focused on ensuring that the editing and execution of flowgraphs work properly through the GUI. Lastly, for the Data Downlink and Storage module, the tests have verified that the telemetry and experiment data sent by the satellite are decoded correctly and accessible to the operators.

<sup>3</sup>Gpredict: <http://gpredict.oz9aec.net/>

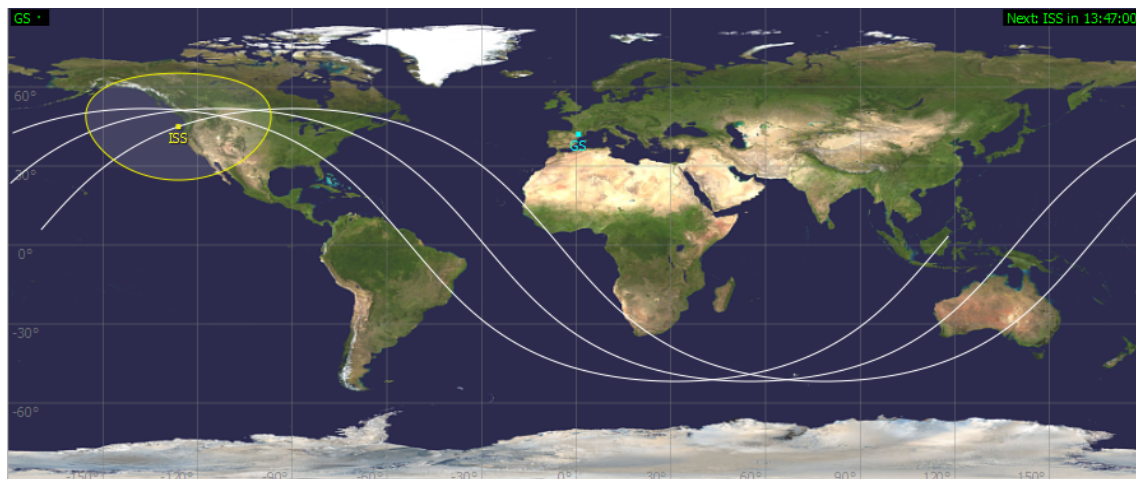
### 5.3.1. Orbit prediction and Tracking

To validate the results obtained from the calculations performed by the software, the Gpredict application has been used. Gpredict is a powerful tool that provides fast and accurate real-time satellite tracking using the NORAD SGP4/SDP4 algorithms and also tracks satellites relative to different observer locations [21].

When verifying the accuracy of orbit predictions, it is important to consider that the SGP4 model, which has been the one implemented in the OpCen software, is specifically optimized for computing orbits in Low Earth Orbit (LEO), which typically range from a few hundred kilometers up to around 2,000 kilometers above the Earth's surface. One well-known satellite operating in this orbit is the International Space Station (ISS). Figure 5.3 shows how the ground track of three orbit predictions performed by the OpCen software matches the prediction obtained with Gpredict.



(a) Orbit prediction OpCen.



(b) Orbit prediction Gpredict.

Figure 5.3: Ground Track verification ISS.

However, conducting a more detailed analysis reveals valuable insights. Table 5.1 presents the communication window predictions for the passes scheduled on 10/7/2023 at the Montsec GS. The calculations of the times, azimuth angles, and elevation angles exhibit a close alignment, demonstrating the accurate performance of the system.

OpCen					Gpredict				
AOS	LOS	Max El	AOS Az	LOS Az	AOS	LOS	Max El	AOS Az	LOS Az
00:20:22	00:30:14	17.22°	286.04°	56.35°	00:20:21	00:30:13	17.22°	286.04°	56.31°
01:58:01	02:07:59	18.46°	305.28°	78.29°	01:58:00	02:07:58	18.46°	305.22°	78.25°
03:34:52	03:45:43	56.94°	306.12°	115.66°	03:34:51	03:45:41	56.92°	306.12°	115.63°
05:11:54	05:21:39	17.98°	291.66°	162.95°	05:11:53	05:21:38	17.98°	291.67°	162.97°
20:18:46	20:28:05	14.30°	190.71°	71.22°	20:18:44	20:28:04	14.30°	190.78°	71.21°
21:54:23	22:05:15	70.69°	239.57°	54.72°	21:54:22	22:05:14	70.68°	239.55°	54.72°
23:32:00	23:42:04	19.76°	278.22°	53.89°	23:31:59	23:42:03	19.76°	278.17°	53.86°

Table 5.1: Passes prediction ISS 10/07/2023.

Additionally, an analysis of the Look Angles for the first identified pass confirms the effective performance of the antenna tracking. Figure 5.4 provides a visual representation of the Look Angles, showcasing the consistent and reliable tracking capabilities of the OpCen software.

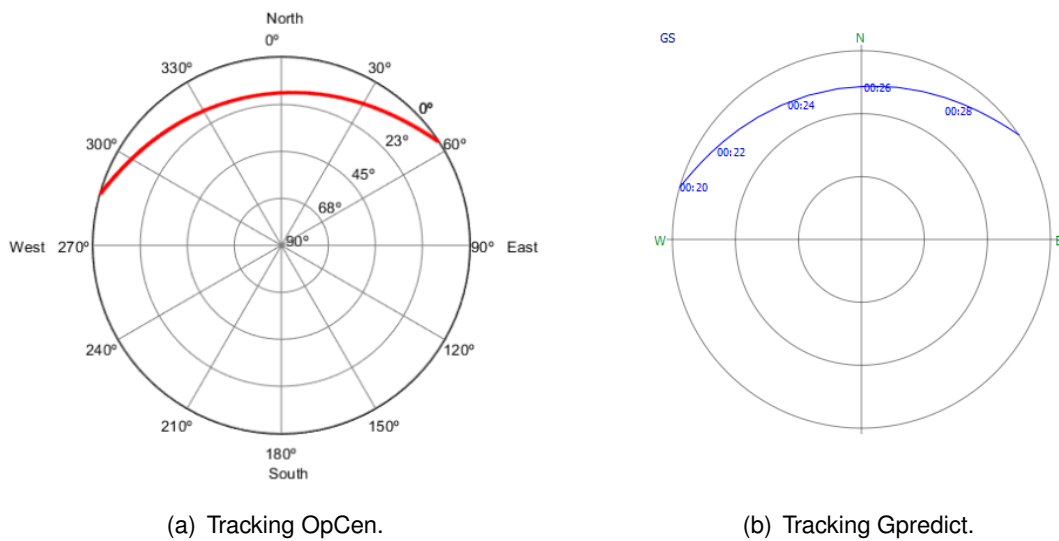


Figure 5.4: Tracking verification ISS.

### 5.3.2. GUI Testing: Flowgraph Editing and Execution

To perform this test, two terminals were utilized, one running the desktop application and the other running the web application, allowing the verification of functionality on both platforms. The verification process began by ensuring the correct saving and retrieval of configurations made by the operator, enabling the reproduction and visualization of the flowgraph in the GUI. Additionally, the synchronization between multiple users has been examined to confirm that simultaneous editing of the same flowgraph is possible, with changes being accurately reflected in the GUI for all users.

Following the configuration phase, an execution test has been conducted. The ZMQ endpoints responsible for sending and receiving messages from the satellite were connected to a dummy server that simulated the transmission of real satellite messages previously recorded. This test aimed to evaluate the functionality of the GUI in handling and processing the received messages sent by the OpCen software. Additionally, it allowed for testing the execution process of the blocks within the flowgraph, ensuring that the desired operations and interactions between components were properly executed.

### 5.3.3. Data decoding verification

The main objective was to verify the proper decoding of the files and telemetry sent by the <sup>3</sup>Cat-4 satellite. This involved comparing the data obtained by the OpCen software with the data provided by the satellite's own software. By ensuring that the decoded data matched, it was confirmed that the OpCen software accurately decoded and processed the information transmitted by the satellite.

## 5.4. Real case scenario

The objective of this test is to validate the functionality of the TT&C module in a real environment. The test will be conducted at the UPC NanoSat Lab facilities, utilizing a dummy GS equipped with a PlutoSDR.

PlutoSDR<sup>4</sup> is a versatile Software-defined radio (SDR) device developed by Analog Devices. It offers a wide frequency range and can transmit and receive various signals. In this context, the PlutoSDR will be used to transmit the messages between the GS and the <sup>3</sup>Cat-4 satellite, which is currently in the lab's facilities.

### Scenario

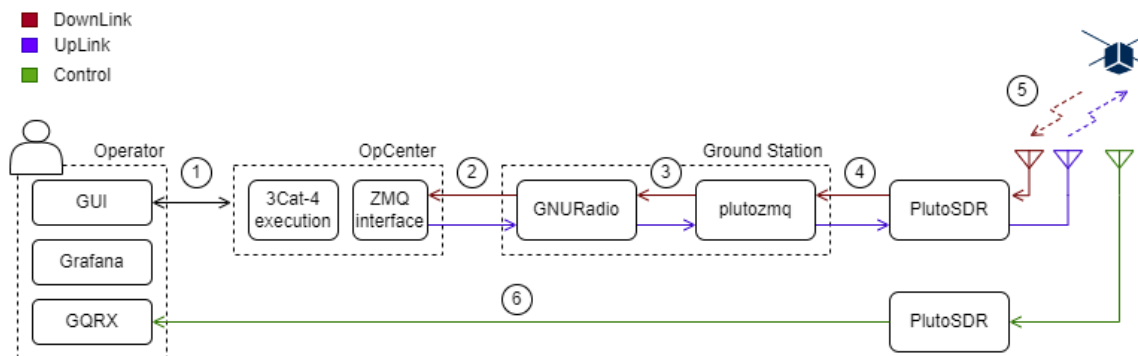


Figure 5.5: Test Scenario.

The operator utilizes the OpCen software GUI to configure the flowgraph and monitor the communication process, while Grafana is used to visualize the received telemetry data. The GUI communicates with the OpCen server via websockets (1), while the OpCen server communicates with the dummy GS using ZMQ sockets (2).

Within the dummy GS, two scripts are running: GNU Radio and plutozmq. GNU Radio<sup>5</sup> is a free and open-source software development toolkit that handles signal processing tasks and provides a variety of blocks for creating radio systems in software. The GNU Radio script converts the bits received from the OpCen (2) into a modulated signal with amplitude, phase, and frequency (3), and vice versa. The plutozmq script manages the PlutoSDR board, redirecting its inputs and outputs to the GNU Radio script.

<sup>4</sup>Pluto SDR: [https://wiki.analog.com/university/tools/pluto/devs/embedded\\_code](https://wiki.analog.com/university/tools/pluto/devs/embedded_code)

<sup>5</sup>GNU Radio Wiki: [https://wiki.gnuradio.org/index.php/Main\\_Page](https://wiki.gnuradio.org/index.php/Main_Page)

The PlutoSDR board transforms the processed bits (4) into a voltage signal, which is then upconverted to the transmitting frequency and amplified (5). The antenna transmits these signals, which are received by the satellite.

Additionally, to ensure the correct transmission between the GS and the satellite, an additional PlutoSDR device is used to listen to the communication frequency channel (437.35 MHz). This device connects via USB (6), and with the help of the open-source software GQRX SDR<sup>6</sup>, the communications can be visualized for monitoring and verification purposes.

## Results

The successful test results demonstrate that the TT&C module operates effectively in a real environment. The communication between the GS and the <sup>3</sup>Cat-4 satellite was successfully established using the dummy GS. All commands were processed correctly by the satellite, and the exchange of messages was completed successfully.

Throughout the test, the operator utilized the OpCen software GUI to configure the flowgraph and monitor the communication process. The operator had the capability to pause the execution and switch to manual mode when required. The telemetry data received from the satellite was accurately received and decoded. The data was then visualized in Grafana.

The test's positive outcome demonstrates the functionality and reliability of the TT&C module in facilitating communication between the GS and the satellite.

---

<sup>6</sup>Gqrx SDR: <https://gqrx.dk/>



## CHAPTER 6. CONCLUSIONS AND FUTURE WORK

This project focused on establishing the foundations for the creation of an OpCen capable of supporting multiple satellite missions and facilitating coordination between different GS. The implementation of the OpCen aimed to centralize the management of both satellite and GS operations, and it needed to be a scalable and flexible platform to adapt to the needs of current and future missions.

The main objective was to reduce the operator's workload through a graphical interface that simplified ground segment tasks. This interface would provide a design tool to configure specific commands and processes during communications, as well as a control function to visualize and monitor ongoing processes. The goal was to reduce the probability of human errors and improve operational efficiency.

This project began by creating a module for orbit calculations to predict satellite communication windows. Through this module, the operator could input satellite data and schedule one or multiple passes with compatible GS.

Simultaneously, a module was developed to support the TT&C area. In this module, the operator was provided with the capability to design message chains to be exchanged with the satellite, allowing them to interconnect blocks and create a customized flowgraph. This functionality enabled efficient and flexible configuration and control of satellite communications, tailored to the specific needs of each mission. The flowgraph design provided an intuitive and visual graphical interface, facilitating the understanding and configuration of interactions between the operator and the satellite. Currently, this feature is only available for communications with the <sup>3</sup>Cat-4 satellite. However, by introducing the necessary code for encoding and decoding messages and protocols from other satellites, it will be able to provide the same support to them.

Furthermore, based on the scheduled passes created by the operator, the software is capable of autonomously establishing a connection with the satellite, allowing for communications at any time of the day. The execution of the flowgraph takes into account the received information to direct the flow of operations. Finally, all processed and generated information during the execution is saved for later visualization and analysis.

Additionally, this project has been one of the first projects carried out by the UPC NanoSat Lab using the Rust programming language. This required an in-depth study of its capabilities and the available libraries that best suited the project's requirements. The choice of Rust as the programming language was based on its focus on safety, performance, and concurrency, making it an ideal option for critical applications such as satellite operations management. It has been confirmed that Rust indeed has a promising future in this field.

## Future work

Although the results of this project have been successful, there is still much work to be done.

Firstly, the scheduling module for programming passes has certain limitations. It considers a time margin before and after the communication to allow for antenna movement and proper satellite pointing, which can result in missed communication opportunities. It would be beneficial to implement a more precise calculation of the time needed for antenna movements, allowing for smoother transitions and support for multiple satellites.

Similarly, the current system only allows scheduling a satellite pass using a single GS. This means that once the satellite is no longer visible to that GS, the communication ends. A valuable upgrade would be to enable the handover of communication between GSs, allowing for longer and uninterrupted communication sessions.

Additionally, the GUI has a good margin for improvement. Implementing new features in the flowgraph editor will allow more flexibility and facilities for the operator when designing the missions. For example, introducing the concept of macro blocks composed of simple blocks (currently implemented) would be useful for reusing patterns in different scenarios. Other potential enhancements will arise as the software is used and user feedback is gathered.

Lastly, the current method of storing all generated files on the OpCen server is not feasible, since they are large-volume files and in large quantities. It is necessary to explore alternative storage solutions that can accommodate the storage demands of the software while maintaining efficient access to the necessary data.

# BIBLIOGRAPHY

- [1] Cap'n Proto: Introduction. [Online]. Available: <https://capnproto.org/> (Accessed on May 22, 2023). 50
- [2] Celestrak. "Understanding GPS Coordinates." *Celestrak Columns*, vol. 4, no. 5, pp. 1-7, May 1996. [Online]. Available: <https://celestrak.org/columns/v04n05/> (Accessed on June 9, 2023). 43
- [3] IBM. REST APIs. [Online]. Available: <https://www.ibm.com/topics/rest-apis> (Accessed on May 21, 2023). 48
- [4] IBM. WebSocket Applications. [Online]. Available: <https://www.ibm.com/docs/en/was/9.0.5?topic=applications-websocket> (Accessed on May 22, 2023). 48
- [5] ZeroMQ. ZeroMQ. [Online]. Available: <https://zeromq.org/> (Accessed on May 22, 2023). 49
- [6] GitHub. Egui GitHub Repository. [Online]. Available: [bibliographyhttps://github.com/emilk/egui](https://github.com/emilk/egui) (Accessed on May 22, 2023). 70
- [7] Tokio. Tokio Tutorial. [Online]. Available: <https://tokio.rs/tokio/tutorial> (Accessed on May 23, 2023). 39
- [8] RustLang. The Rust Programming Language. [Online]. Available: <https://www.rust-lang.org/> (Accessed on June 28, 2023). 38
- [9] Aina Garcia Espriu. *Design and Implementation of a software architecture for an extensible network of Satellite Ground Stations*. (UPC. Barcelona. 2020). 19
- [10] Center for Astrophysics Harvard & Smithsonian. Astronomical Times. [Online]. Available: <https://lweb.cfa.harvard.edu/~jzhao/times.html#GMST> (Accessed on July 5, 2023). 12
- [11] ESA. Navipedia. [Online]. Available: [https://gssc.esa.int/navipedia/index.php/Conventional\\_Celestial\\_Reference\\_System](https://gssc.esa.int/navipedia/index.php/Conventional_Celestial_Reference_System) (Accessed on July 5, 2023). xi, 11
- [12] ESA. Navipedia. [Online]. Available: [https://gssc.esa.int/navipedia/index.php/Transformation\\_between\\_Terrestrial\\_Frames](https://gssc.esa.int/navipedia/index.php/Transformation_between_Terrestrial_Frames) (Accessed on July 5, 2023). 12
- [13] ESA. Navipedia. [Online]. Available: [https://gssc.esa.int/navipedia/index.php/Transformations\\_between\\_ECEF\\_and\\_ENU\\_coordinates](https://gssc.esa.int/navipedia/index.php/Transformations_between_ECEF_and_ENU_coordinates) (Accessed on July 5, 2023). xi, 14, 15
- [14] Celestrak. Orbital Coordinate Systems. [Online]. Available: <https://celestrak.org/columns/v02n01/> (Accessed on July 5, 2023). xi, 12, 13
- [15] Zehentner, Norbert. *Kinematic orbit positioning applying the raw observation approach to observe time variable gravity*. (2017). xi, 9

- [16] NASA. Satellite Orbital Elements. [Online]. Available: [https://www.grc.nasa.gov/www/k-12/TRC/laefs/laefs\\_k.html](https://www.grc.nasa.gov/www/k-12/TRC/laefs/laefs_k.html) (Accessed on July 5, 2023). 9
- [17] Saika Aida, Michael Kirschner. *ACCURACY ASSESSMENT OF SGP4 ORBIT INFORMATION CONVERSION INTO OSCULATING ELEMENTS*. 10
- [18] Alen Space. A basic guide to nanosatellites. [Online]. Available: <https://alen.space/basic-guide-nanosatellites/> (Accessed on July 5, 2023). xi, 7
- [19] NASA. *CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers*. (2017). 7
- [20] UPC NanoSat Lab. Missions and Projects. [Online]. Available: <https://nanosatlab.upc.edu/en/missions-and-projects> (Accessed on July 7, 2023). 16
- [21] NASA. *Ground Data Systems Chapter*. (2022). 87
- [22] Patrick Geremia. *Cyclic Redundancy Check Computation*. (1999). 81
- [23] Abdullah, Ako. *Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data*. (2017).
- [24] Evans, David J., Alessandro Donati. *The ESA POCKET+ Housekeeping Telemetry Compression Algorithm: Why Make Spacecraft Operations Harder than It Already Is?* 2018 SpaceOps Conference, American Institute of Aeronautics and Astronautics, 2018. DOI.org (Crossref). 82
- [25] Celestrak. A New Way to Obtain GP Data (aka TLEs). [Online]. Available: <https://celestrak.org/NORAD/documentation/gp-data-formats.php> (Accessed on July 10, 2023). xii, 83

# **APPENDICES**



# APPENDIX A. OPERATION CENTER SOFTWARE API

## A.1. Authentication

**Description:** Validates the credentials provided by the user and generates an authentication token with the permissions and access level of the requester.

**Access Levels (Roles):**

- Ground Station Manager
- Satellite operator
- Telemetry Expert

**Method:** POST

**Path:** */auth*

**Body:** [User](#)

**Answers:**

- 200 OK → [Roles](#)
  - 401 UNAUTHORIZED
- 

## A.2. Scheduling endpoints

### Get all Passes

**Description:** Returns a list of all the passes scheduled in the OpCen that are not yet complete, sorted by date in descending order (from closest to farthest).

**Method:** GET

**Path:** */schedule*

**Auth:** -

**Body:** -

**Answers:**

- 200 OK → [PassList](#)
-

## Request Passes

**Description:** Schedule the passes for the given satellite based on the requested mode, only if the stored configuration of the satellite is supported by any of the ground stations and if there is no conflict with those already programmed.

**Method:** PUT

**Path:** */schedule*

**Auth:** Ground Station Manager

**Body:** [RequestPass](#) (maxSearchDuration is optional, by default 48 hours)

**Answers:**

- 200 OK → [PassList](#)
  - 400 BAD REQUEST
  - 401 UNAUTHORIZED
  - 404 NOT FOUND → (Unable to find the specified satellite or a compatible ground station for the given satellite)
  - 406 NOT ACCEPTABLE → (Error occurred while retrieving the TLE from the database, decoding the elements, or computing the Keplerian elements)
  - 409 CONFLICT → (Scheduling could not be successfully completed because one or more passes have conflicted with the already scheduled ones)
- 

## Delete Pass

**Description:** Deletes the existing pass identified by the pass\_id parameter.

**Method:** DELETE

**Path:** */schedule*

**Auth:** Ground Station Manager

**Body:** -

**Answers:**

- 200 OK
  - 401 UNAUTHORIZED
  - 404 NOT FOUND
-



## Get all past passes

**Description:** Returns a list of 101 finished passes, sorted from most recent to least recent, starting from index\*100.

**Method:** GET

**Path:** */schedule/past/:index*

**Auth:** -

**Body:** -

**Answers:**

- 200 OK → [PassList](#)
  - 400 BAD REQUEST
- 

## Get all satellites

**Description:** Returns a list of all satellites stored in the database.

**Method:** GET

**Path:** */schedule/satellites*

**Auth:** -

**Body:** -

**Answers:**

- 200 OK → [SatelliteList](#)
- 

## Add Satellite

**Description:** Creates a new satellite and stores it in the database using the provided configuration.

**Method:** POST

**Path:** */schedule/satellites*

**Auth:** Ground Station Manager

**Body:** [Satellite](#)

**Answers:**

- 200 OK
  - 400 BAD REQUEST
  - 401 UNAUTHORIZED
  - 404 NOT FOUND → (Failed to download TLE data for the registered satellite)
-

## Edit Satellite

**Description:** Rewrites the information of an existing satellite with the one provided in the request.

**Method:** PUT

**Path:** */schedule/satellites*

**Auth:** Ground Station Manager

**Body:** [Satellite](#)

**Answers:**

- 200 OK
  - 400 BAD REQUEST
  - 401 UNAUTHORIZED
  - 404 NOT FOUND → (Failed to download TLE data for the updated satellite)
- 

## Update TLE Satellite

**Description:** Update the TLE of the satellite identified by the `norad_id` parameter if the satellite has the automatic download TLE flag configured.

**Method:** PUT

**Path:** */schedule/satellites/:norad\_id*

**Auth:** Ground Station Manager

**Body:** -

**Answers:**

- 200 OK
  - 400 BAD REQUEST → (The satellite does not have the automatic flag set)
  - 401 UNAUTHORIZED
  - 404 NOT FOUND → (Failed to download TLE data for the updated satellite)
- 

## Delete Satellite

**Description:** Deletes the satellite identified with the specified `norad_id` parameter.

**Method:** DELETE

**Path:** */schedule/satellites/:norad\_id*

**Auth:** Ground Station Manager

**Body:** -

**Answers:**

- 200 OK
  - 401 UNAUTHORIZED
  - 404 NOT FOUND
- 

## Update Satellite NORAD ID

**Description:** Update the NORAD ID of the satellite identified by the `norad_id` parameter and replace it with the `norad_id_new` parameter if the satellite is not currently performing a pass.

**Method:** POST

**Path:** `/schedule/satellites/:norad_id/:norad_id_new`

**Auth:** Ground Station Manager

**Body:** -

**Answers:**

- 200 OK
  - 401 UNAUTHORIZED
  - 404 NOT FOUND
  - 406 NOT ACCEPTABLE → (Satellite is currently executing a pass)
- 

## Get all Ground Stations

**Description:** Returns a list of all the ground stations stored in the database.

**Method:** GET

**Path:** `/schedule/gs`

**Auth:** -

**Body:** -

**Answers:**

- 200 OK → [GroundStationList](#)
-

## Update operative state Ground Station

**Description:** Update the operational flag of the ground station if it exists.

**Method:** PUT

**Path:** */schedule/gs*

**Auth:** Ground Station Manager

**Body:** [GroundStation](#) (Only the gsId and operative fields are necessary)

**Answers:**

- 200 OK
  - 400 BAD REQUEST
  - 401 UNAUTHORIZED
  - 404 NOT FOUND
- 

## A.3. Flowgraph-based TT&C endpoints

### Get all Satellites (with a flowgraph configured)

**Description:** Returns a list of all satellites that have at least one configured flowgraph, ordered by their last update timestamp.

**Method:** GET

**Path:** */flowgraph*

**Auth:** Satellite operator & Telemetry Expert

**Body:** -

**Answers:**

- 200 OK → [FlowgraphList](#)
  - 401 UNAUTHORIZED
- 

### Create new Flowgraph

**Description:** Create the first flowgraph for the desired satellite if none has been configured yet, with the Acquisition of Sight block already instantiated and the default mode configured.

**Method:** POST

**Path:** */flowgraph*

**Auth:** Satellite operator

**Body:** [Flowgraph](#) (Only the name and noradId fields are necessary)

**Answers:**

- 200 OK → [Flowgraph](#)
  - 400 BAD REQUEST
  - 401 UNAUTHORIZED
  - 406 NOT ACCEPTABLE → (Already exists one flowgraph for the satellite)
- 

## Upload Mission Blocks

**Description:** Upload a file containing the information for the desired mission block execution to be selected during the flowgraph execution, along with a list of all the commands.

**Method:** PUT

**Path:** */flowgraph*

**Auth:** Satellite operator

**Body:** [MissionBlockFile](#)

**Answers:**

- 200 OK → [Edition](#) (operationType: missionBlock)
  - 400 BAD REQUEST
  - 401 UNAUTHORIZED
  - 406 NOT ACCEPTABLE → (Error reading the import file)
- 

## Get all Flowgraphs Satellite

**Description:** Obtain all the flowgraphs associated with the satellite identified by the no-rad.id parameter.

**Method:** GET

**Path:** */flowgraph/:norad\_id*

**Auth:** Satellite operator

**Body:** -

**Answers:**

- 200 OK → [FlowgraphList](#)
  - 401 UNAUTHORIZED
  - 404 NOT FOUND
-

## Delete Flowgraphs Satellite

**Description:** Delete all flowgraphs associated with the satellite.

**Method:** DELETE

**Path:** */flowgraph/:norad\_id*

**Auth:** Satellite operator

**Body:** -

**Answers:**

- 200 OK
  - 401 UNAUTHORIZED
  - 404 NOT FOUND
- 

## Test Satellite execution

**Description:** Starts the execution of the flowgraph using the Ground Station of the test environment.

**Method:** GET

**Path:** */flowgraph/test/:norad\_id/:flowgraph\_id*

**Auth:** Satellite operator

**Body:** -

**Answers:**

- 200 OK
  - 401 UNAUTHORIZED
  - 404 NOT FOUND
- 

## Websocket configuration interface

**Description:** Allows configuring the flowgraph identified by the flowgraph\_id parameter. All operations performed are validated and accepted by the server, which sends updates to all users connected to the socket, enabling collaborative configuration.

**Path:** */ws/configuration/:flowgraph\_id*

**Auth:** Satellite operator

**Message format:** [Edition](#)

---

## Websocket execution interface

**Description:** Allows viewing and controlling the execution of the flowgraph for the satellite identified by the `norad_id` parameter.

**Path:** `/ws/execution/:norad_id`

**Auth:** Satellite operator (all functionalities) & Telemetry Expert (only receive messages and request files)

**Message format:** [Execution](#)

---

## A.4. Cap'n Proto Models

### auth.capnp

```
@0xa53401f35b9c1292;

struct User {
    username @0 : Text;
    password @1 : Text;
}
struct Roles {
    token @0 : Text;
    satelliteOperator @1 : Bool;
    groundStationManager @2 : Bool;
    telemetryExpert @3 : Bool;
}
```

### ground\_station.capnp

```
@0xde755e0b536afcda;

struct GroundStationList {
    list @0 : List(GroundStation);
}
struct GroundStation {
    gsId @0 : Int32;
    gsName @1 : Text;
    gsLocation @2 : Coords;
    ulfreqMax @3 : UInt32;
    ulfreqMin @4 : UInt32;
    dlfreqMax @5 : UInt32;
    dlfreqMin @6 : UInt32;
    bandwidth @7 : UInt32;
    operative @8 : Bool;
}
struct Coords {
    latitude @0 : Float64;
    longitude @1 : Float64;
    altitude @2 : Float64;
}
```

## schedule.capnp

```
@0xf47587b04aae443e;

struct RequestPass {
  noradId @0 : Int32;
  maxSearchDuration @1 : UInt32;
  forcedSave @2 : Bool;
  requestType : union {
    onePass @3 : Void;
    multiplePasses : group {
      numPasses @4 : UInt32;
    }
    allPassesUntil : group {
      end @5 : Text;
    }
    allPassesInterval : group {
      start @6 : Text;
      end @7 : Text;
    }
    multiplePassesFrom : group {
      numPasses @8 : UInt32;
      start @9 : Text;
    }
  }
}

struct TLE {
  objectName @0 : Text;
  line1 @1 : Text;
  line2 @2 : Text;
}

struct PassList {
  list @0 : List(PassData);
}

struct PassData {
  id @0 : Int32;
  noradId @1 : Int32;
  startTime @2 : Text;
  endTime @3 : Text;
  aos @4 : Text;
  los @5 : Text;
  maxElevation @6 : Float64;
  aosAzimuth @7 : Float64;
  losAzimuth @8 : Float64;
  conflict @9 : Bool;
  satName @10 : Text;
  gsName @11 : Text;
  logPath @12 : Text;
}
```

**i** Date time format is YYYY-MM-DDTHH:MM:SSZ (Example: 2023-06-18T16:30:20+00:00)



## satellite.capnp

```
@0xee66025180549fc4;

struct SatelliteList {
    list @0 : List(Satellite);
}
struct Satellite {
    noradId @0 : Int32;
    satelliteName @1 : Text;
    ulfreq @2 : UInt32;
    dlfreq @3 : UInt32;
    bandwidth @4 : UInt32;
    tle : union {
        manual : group {
            line1 @5 : Text;
            line2 @6 : Text;
        }
        automatic : group {
            datetime @7 : Text;
            updateTime @8 : Text;
        }
    }
}
```

## configure\_flowgraph.capnp

```
@0x89bf19e6c1a7021f;

struct Edition {
    lastEdit @0 : Text;
    operationType : union {
        newBlock : group {
            operationId @1 : UInt32;
            blockId @2 : Int32;
            blockDefinition @3 : Text;
            positionX @4 : Float32;
            positionY @5 : Float32;
            command @6 : Text;
            outputType @7 : Text;
        }
        newConnection : group {
            originId @8 : Int32;
            output @9 : Text;
            destinationId @10 : Int32;
            input @11 : Text;
        }
        uploadFile : group {
            blockId @12 : Int32;
            file @13 : Text;
            data @14 : Data;
        }
    }
}
```

```

    }
    deleteBlock @15 : Int32;
    deleteConnection : group {
        originId @16 : Int32;
        output @17 : Text;
        destinationId @18 : Int32;
        input @19 : Text;
    }
    loadFlowgraph : group {
        blockList @20 : List(BlockInstance);
        connectionList @21 : List(Connection);
        paramsList @22 : List(BlockParams);
    }
    missionBlock : group {
        noradId @23 : Int32;
        blockList @24 : List(MissionBlock);
    }
    moveNode : group {
        blockId @25 : Int32;
        positionX @26 : Float32;
        positionY @27 : Float32;
    }
    setCommand : group {
        blockId @28 : Int32;
        command @29 : Text;
    }
    addFlowgraph @30 : Flowgraph;
    updateFlowgraph @31 : Flowgraph;
    deleteFlowgraph @32 : Int32;
    blockEdition @33 : Void;
    setCommandBytes : group {
        blockId @34 : Int32;
        bytes @35 : List(UInt8);
    }
    configTimeout : group {
        blockId @36 : Int32;
        time @37 : UInt8;
        rep @38 : Int16;
    }
    setCommandOption : group {
        blockId @39 : Int32;
        option @40 : UInt8;
    }
}

struct FlowgraphList {
    list @0 : List(Flowgraph);
}

struct PassesList {
    list @0 : List(Pass);
}

```

```

}
struct Pass {
    passId @0 : Int32;
    aos @1 : Text;
    los @2 : Text;
    fgName @3 : Text;
}
struct Flowgraph {
    noradId @0 : Int32;
    flowgraphId @1 : Int32;
    name @2 : Text;
    lastEdit @3 : Text;
    mode : union {
        pass @4 : List(Int32);
        time : group {
            gsId @5 : Int32;
            startTime @6 : Text;
            endTime @7 : Text;
        }
        groundStation @8 : Int32;
        default @9 : Void;
    }
}
struct BlockInstance {
    blockId @0 : Int32;
    blockDefinition @1 : Text;
    positionX @2 : Float32;
    positionY @3 : Float32;
    command @4 : Text;
    timeOutTime @5 : UInt8;
    timeOutRep @6 : Int16;
    outputType @7 : Text;
}
struct MissionBlock {
    name @0 : Text;
    blockType @1 : Text;
}
struct MissionBlockFile {
    noradId @0 : Int32;
    data @1 : Data;
}
struct Connection {
    originId @0 : Int32;
    output @1 : Text;
    destinationId @2 : Int32;
    input @3 : Text;
}
struct BlockParams {
    blockId @0 : Int32;
    params @1 : Data;
}

```

## execute\_flowgraph.capnp

```
@0xadb696cb50df038c;

struct Execution {
  operationType : union {
    loadFlowgraph : group {
      blockList @0 : List(BlockInstance);
      connectionList @1 : List(Connection);
      paramsList @2 : List(BlockParams);
      missionBlockList @3 : List(MissionBlock);
    }
    startBlock @4 : Int32;
    endBlock @5 : Int32;
    sendLog : group {
      datetime @6 : Text;
      tag @7 : Text;
      log @8 : Text;
      color @9 : Text;
    }
    reqManualNextBlock @10 : List(Int32);
    resManualNextBlock @11 : Int32;
    finish @12 : Void;
    executeManualBlock : group {
      blockDefinition @13 : Text;
      command @14 : Text;
      data @15 : Data;
    }
    aosTime : group {
      blockId @16 : Int32;
      aosTime @17 : Text;
      losTime @18 : Text;
    }
    manualExecution @19 : Bool;
    manualBlock : group {
      blockId @20 : Int32;
      blockDefinition @21 : Text;
      command @22 : Text;
      params @23 : Data;
      timeoutTime @24 : UInt8;
      timeoutRep @25 : Int16;
    }
    manualBlockDelete @26 : Int32;
    downloadFile : group {
      fileName @27 : Text;
      data @28 : Data;
    }
  }
}

struct BlockInstance {
```

```
    blockId @0 : Int32;
    blockDefinition @1 : Text;
    positionX @2 : Float32;
    positionY @3 : Float32;
    command @4 : Text;
}
struct Connection {
    originId @0 : Int32;
    output @1 : Text;
    destinationId @2 : Int32;
    input @3 : Text;
}
struct BlockParams {
    blockId @0 : Int32;
    params @1 : Data;
}
struct MissionBlock {
    name @0 : Text;
    blockType @1 : Text;
}
```