TÉCNICO LISBOA

Ground Station Distribution and Automation for Low-Earth Orbit Cubesats

Rafael Paixão Branco

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Alberto Manuel Ramos da Cunha
Prof. Rui Manuel Rodrigues Rocha

Examination Committee

Chairperson: Prof. Rui Filipe Fernandes Prada
Supervisor: Prof. Alberto Manuel Ramos da Cunha
Member of the Committee: Prof. Rui António Policarpo Duarte

November 2022

Acknowledgments

I would like to thank both my supervisors, Professor Alberto Cunha and Professor Rui Rocha for their guidance and for giving me the opportunity to work on this project.

I would also like to thank the ISTSAT team for welcoming me and making me feel at home since day one, and to the Nanostar project that made this work possible.

Abstract

The high orbital velocity of low Earth orbit satellites precludes their persistent connection with a ground station. For this reason, different techniques are employed to increase the connection availability, and data throughput. In this work, the ground segment for the ISTSAT-1 is studied, its architecture revised and improved, to ease its distribution, automation, and operation. While the suggested improvements and solutions are directly applicable to the ground segment of ISTSAT-1, they should be general enough to be employed on other ground segments.

Keywords

Distribution; Automation; Graphical User Interface; Satellite;

Resumo

A elevada velocidade orbital de satélites em órbitas baixas impede a sua conexão persistente com uma estação terrestre. Desta forma, diferentes técnicas são utilizadas para aumentar a disponibilidade de conexão e a taxa de transferência de dados. Neste trabalho, o segmento terrestre do ISTSAT-1 é estudado, a sua arquitectura é revista e melhorada de forma a facilitar a sua distribuição, automação e operação. Embora as sugestões e soluções propostas sejam directamente aplicáveis ao segmento terrestre do ISTSAT-1, devem ser gerais o suficiente para poderem ser aplicadas a outros projectos espaciais.

Palavras Chave

Distribuição; Automação; Interface Gráfica; Satélite;

Contents

1 Introduction 1
1.1 Motivation . . . 3
1.2 Objectives and Contributions . . . 4
1.3 Overview . . . 4

2 Related Work 7
2.1 Ground Station Distribution . . . 9
2.1.1 CONTEC . . . 11
2.1.2 Leaf Space - Leaf Line . . . 11
2.1.3 RBC Signals . . . 12
2.1.4 SatNOGS . . . 13
2.2 Ground Segment Automation and Interface . . . 14
2.2.1 Automation Level . . . 14
2.2.2 Mission Operation software . . . 15
2.2.2.A COSMOS . . . 15
2.2.2.B Major Tom . . . 16

3 Ground Segment of ISTSAT-1 19
3.1 Initial Architecture . . . 21
3.1.1 Protocols . . . 21
3.1.2 Ground Segment Control (GSCTL) . . . 22
3.1.2.A Commands . . . 23
3.1.2.B Operations . . . 24
3.1.3 Core Server . . . 24
3.1.4 Electrical Ground Support Equipment (EGSE) . . . 24
3.1.5 Ground Station . . . 24
3.1.5.A Traditional Ground Station . . . 25
3.1.5.B Software Defined Radio (SDR) Ground Station . . . 26
3.1.5.C Ground Station Interface . . . 27

3.1.6 Gateway Interface . . . 27
3.1.7 Automation . . . 28
3.1.8 Data Visualiser . . . 28
3.1.9 Security . . . 29
3.2 Proposed Architecture . . . 29
3.2.1 Communication Protocols . . . 29
3.2.2 Stand-alone Electrical Ground Support Equipment (SEGSE) . . . 31
3.2.3 Operation Queue . . . 32
3.2.4 Graphical User Interface . . . 32
4 Implementation **33**
4.1 Core Server Interfaces and Gateway Behaviour . . . 35
4.1.1 Trio-RPC . . . 36
4.1.1.A Messages . . . 36
4.1.1.B Concurrency . . . 37
4.1.1.C Syntax . . . 39
A – Server . . . 39
B – Client . . . 40
4.1.1.D Security . . . 41
4.1.1.E Backwards Compatibility . . . 41
4.1.2 Core Server . . . 41
4.1.3 Ground Station . . . 42
4.1.4 EGSE . . . 43
4.2 Graphical User Interface . . . 43
4.2.1 Overview . . . 44
4.2.2 Command Definition . . . 45
4.2.2.A Format . . . 45
4.2.2.B ISTSAT-1's commands . . . 47
4.2.3 Major Tom Translation Layer . . . 49
4.2.4 Telemetry . . . 50
4.2.5 Impediment . . . 51
4.3 Alternative Graphical User Interface . . . 51
4.3.1 Configuration . . . 52
4.3.2 Interface . . . 53
4.3.3 Security . . . 54
4.4 Operation Queue . . . 54

5 Validation **57**

5.1 Mission Test 59

5.2 Vibration Test Campaign 60

5.2.1 Challenges 60

5.3 Space Studies Program (SSP) Class 61

5.4 Results 61

6 Conclusion **65**

6.1 Future Work 67

Bibliography **69**

List of Figures

2.1 Ground Station acting as a Gateway . . . 9
2.2 Ground Station Network . . . 9
2.3 Space Relay Network . . . 10
2.4 CONTEC's Ground Station Network (GSN)[1] . . . 11
2.5 Leaf Line GSN[2] . . . 12
2.6 RBC Signals' GSN[3] . . . 12
2.7 Satellite Networked Open Ground Station (SatNOGS) high level architecture . . . 13
2.8 Ground Operator - Satellite interaction . . . 14
2.9 COSMOS Architecture[4] . . . 16
2.10 Major Tom Architecture[5] . . . 17

3.1 Ground Segment Architecture . . . 21
3.2 Protocol Stack . . . 22
3.3 Protocol Stack . . . 22
3.4 EGSE Communication Protocol Stack . . . 25
3.5 Traditional Radio Ground Station . . . 25
3.6 Traditional Ground Station Communication Protocols . . . 26
3.7 Software Defined Radio Ground Station . . . 26
3.8 SDR Ground Station Communication Protocols . . . 27
3.9 Lack of standardisation of gateway interfaces . . . 28
3.10 Ground Segment Architecture . . . 30
3.11 Ground Station Communication Protocol Stack . . . 30
3.12 EGSE Communication Protocol Stack . . . 31
3.13 Stand-alone EGSE . . . 31
3.14 Web Server . . . 32

4.1 Remote Function Call Sequence Diagram . . . 37

4.2 Two Remote Function Calls with One Executer Sequence Diagram 38
4.3 Two Remote Function Calls with Two Executers Sequence Diagram 38
4.4 ”gateway” Capability 41
4.5 Ground Station Components 42
4.6 Major Tom 43
4.7 Major Tom Interface - 3D View 44
4.8 Smooth Operator’s Data Request Interface 54
4.9 Operation Queue Sequence Diagram 55

5.1 Mission Test Setup 60
5.2 Vibration Test Campaign Reduced Functional Test (RFT) 61
5.3 GSCTL Data Request 61
5.4 Smooth Operator Data Request ”flight_pass” 63

Listings

3.1 Command example . . . 23
3.2 Data Request example . . . 23
3.3 Config Get example . . . 23
3.4 Config Set example . . . 24
3.5 Operation example . . . 24
4.1 Sample request message . . . 36
4.2 Sample response message . . . 37
4.3 RPCServer instantiation . . . 39
4.4 Register Function . . . 39
4.5 Start Server . . . 40
4.6 Instantiate Client and Register Function . . . 40
4.7 Connect and Execute Remote Function . . . 40
4.8 Command Definition . . . 46
4.9 Core Server Basic Functionality Client Application Programming Interface (API) . . . 47
4.10 Data Request EPS' ”battery_charge” . . . 48
4.11 ISTSAT-1 Command Definition First Approach . . . 48
4.12 ISTSAT-1 Command Definition Approach . . . 49
4.13 Major Tom Request Format Example . . . 50
4.14 Major Tom Response Format Example . . . 50
4.15 Major Tom Measurement Format Example . . . 51
4.16 Smooth Opertor's ”data” Definition Example . . . 52
4.17 Smooth Opertor's ”config” Definition Example . . . 53
4.18 Smooth Opertor's ”command” Definition Example . . . 53

Acronyms

ADS-B Automatic Dependent Surveillance-Broadcast

AX.25 Amateur X.25

AFSK Audio Frequency-Shift Keying

API Application Programming Interface

CLI Command Line Interface

CSF CubeSat Support Facility

CSP Cubesat Space Protocol

CSV Comma-Separated Values

EGSE Electrical Ground Support Equipment

ESA European Space Agency

FYS Fly Your Satellite!

FPGA Field-Programmable Gate Array

LEO low Earth orbit

LEOP Launch and Early Orbit Phase

LOS line of sight

MT Mission Test

GSaaS Ground Station as a Service

GUI Graphical User Interface

GSCTL Ground Segment Control

GSN Ground Station Network

HDLC High-Level Data Link Control

INCP ISTNanosat-1 Command Protocol

INESC-MN Instituto de Engenharia de Sistemas e Computadores - Microsistemas e Nanotecnologias

IP Internet Protocol

IST Instituto Superior Técnico

JSON javascript object notation

OSI Open Systems Interconnection

RF Radio Frequency

RFT Reduced Functional Test

RPC Remote Procedure Call

RDP Reliable Data Protocol

SatNOGS Satellite Networked Open Ground Station

SDR Software Defined Radio

SEGSE Stand-alone Electrical Ground Support Equipment

SRN Space Relay Network

SSP Space Studies Program

UI User Interface

TCP Transmission Control Protocol

TLE two-line element set

TLS Transport Layer Security

TNC Terminal Node Controller

TTC Telemetry, Tracking, and Command

UHF Ultra High Frequency

USB Universal Serial Bus

VHF Very High Frequency

VPS Virtual Private Server

VT Vibration Test

YAML YAML Ain’t Markup Language

1

Introduction

Contents

1.1 Motivation 3
1.2 Objectives and Contributions 4
1.3 Overview 4

Nowadays many satellites orbit our planet and while their missions diverge from one another, most of them need to broadcast information and receive commands from Earth in order to fulfil their missions. This information can, typically, be subdivided into mission related data, telecomands, and telemetry which allows the detection of possible anomalies with the spacecraft.

To allow this communication between the two parties a ground segment is needed. A ground segment is the part of the satellite's communication system that resides on the ground [1], allowing the monitoring and control of the spacecraft from Earth. The structure of this segment varies according to the satellites they are supporting. However, every ground segment needs at least one ground station which provides a physical layer interface to communicate with the satellites. In order for ground stations to be able to transmit and receive data, they need a line of sight (LOS) to the target spacecraft. Therefore, the connection availability is limited by the satellite's orbit and the ground station's location.

Traditionally, when a satellite is in LOS, satellite operators are entrusted with the task of sending the appropriate commands to achieve some previously stipulated goal. However, recently, there has been an increase in the automation of repetitive tasks, not only to free staff for more important endeavours but also to reduce the costs associated with the mission.

1.1 Motivation

Today a great number of satellites are launched into low Earth orbits (LEOs), orbits normally at an altitude of less than 1000 km above Earth. The low altitude of these orbits allows high data bandwidth and low latency communications. However, satellites in LEOs travel at a speed of around 7.8 km per hour, taking approximately 90 minutes to circle Earth [2], nearly 16 times the rotational period of Earth. Therefore, resorting to one LEO satellite and a ground station, it is not possible to persistently maintain a LOS and, consequently, a connection between them.

When operating a LEO satellite, one has to be aware of when the spacecraft is above the horizon and thus accessible for communication. This is known as a pass. The number of passes, as well as their temporal distribution, varies according to many factors such as the orbit's altitude and inclination, and the ground station's location.

Communication is central to a satellite's mission success. Therefore it is important to employ techniques to increase the number of available passes and their throughput. This problem needs to be addressed by most LEO satellites and the ISTSAT-1 is not an exception.

ISTSAT-1 is a cubesat (1U) that will orbit Earth at 550km of altitude, in a sun-synchronous LEO. Its mission is to validate that a small satellite, with a planar antenna, and a general microprocessor (instead of a Field-Programmable Gate Array (FPGA) board) is able to collect Automatic Dependent Surveillance-Broadcast (ADS-B) messages sent by planes [3]. In order to accomplish its mission, the

spacecraft needs to be supported by a ground segment.

The ground segment of ISTSAT-1 was developed alongside the spacecraft, while its main task was to support the satellite's needs and test it. Now, that the spacecraft is near launch, it is necessary to audit the state of its ground segment, assessing flaws, suggesting improvements, and implementing solutions. While the suggested improvements and solutions are directly applicable to the ground segment of ISTSAT-1, they should be general enough to be employed on another mission's ground segments.

1.2 Objectives and Contributions

This work results in the redesign of the ISTSAT-1's ground segment, improving its network performance, interface, and automation capabilities. These contributions were motivated by the objectives extracted from the analysis of the initial ISTSAT-1's ground segment.

Since the ground segment was developed at the same time as the spacecraft, its structure grew organically to support the needs of the satellite. This resulted in design decisions that poorly influenced latency and goodput. Thus, one of the objectives was to improve both metrics through the modularization of the ground segment.

During the evolution of the ground segment, operators resorted to a Command Line Interface (CLI) to interact with the spacecraft. While this decision suffices for the development process, it is important to create an User Interface (UI) easier to use. Therefore, another objective was to improve the ground segment so that its operation becomes more user friendly.

Throughout the development of the satellite, it endured long lasting tests. However, these tests did not require any scheduling, they were simply left running. This motivated the objective of employing a system capable of scheduling operations.

1.3 Overview

The remainder of this document is divided as follows:

- Chapter 2: references the work done in the area of ground station distribution, ground segment automation, and command and control systems.
- Chapter 3: details the current status ISTSAT-1's ground segment, exposing the found flaws and providing an alternative architecture.
- Chapter 4: thoroughly describes the steps taken to implemented the proposed ground segment architecture.
- Chapter 5: discusses the steps taken to validate the design implementation.

- Chapter 6: concludes and summarises the document.

2

Related Work

Contents

2.1 Ground Station Distribution 9
2.2 Ground Segment Automation and Interface 14

In order to properly design a ground segment for a LEO satellite, it is important to research about how ground station distribution, ground segment automation, and interfaces are typically handled. This investigation aims to provide insight about solutions that can be applicable to problems the ISTSAT-1 may endure.

2.1 Ground Station Distribution

Ground stations are responsible for acting as a gateway between the ground and space segments (Figure: 2.1). A connection between them is established through the transmission and reception of electromagnetic waves which need to be modulated and demodulated in order to allow the exchange of data between both parties.

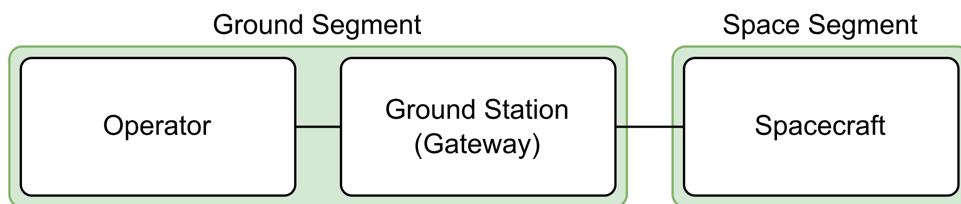

Figure 2.1: Ground Station acting as a Gateway

For a ground station to be able to communicate with a satellite, it must be in LOS. However, for LEO satellites, this only accounts for a relatively short period of time. Therefore, controlling just one ground station might not fulfil the satellite's mission data budget. There are two main approaches to increase communication availability.

One technique is to increase the number of ground stations accessible to the ground segment (Figure 2.2). These are referred to as Ground Station Networks (GSNs).

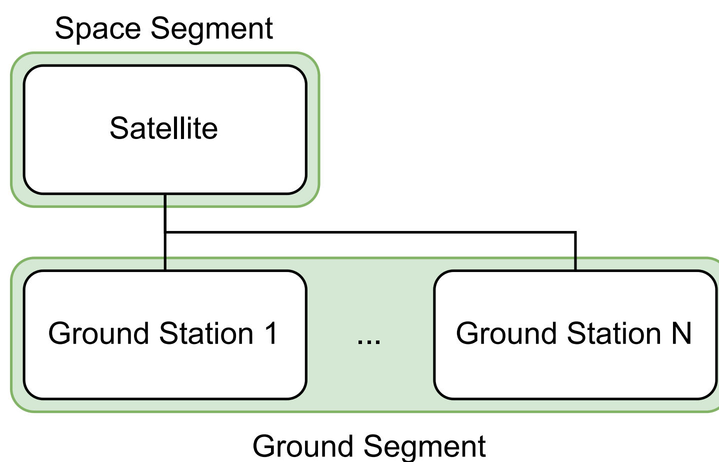

Figure 2.2: Ground Station Network

Another way to solve this issue is by relaying data through a pre-existing network formed by satellites in a constellation (Figure 2.3). Typically referred to as Space Relay Networks (SRNs). However SRNs require the installation of specific hardware and/or software prior to launch in order access these networks.

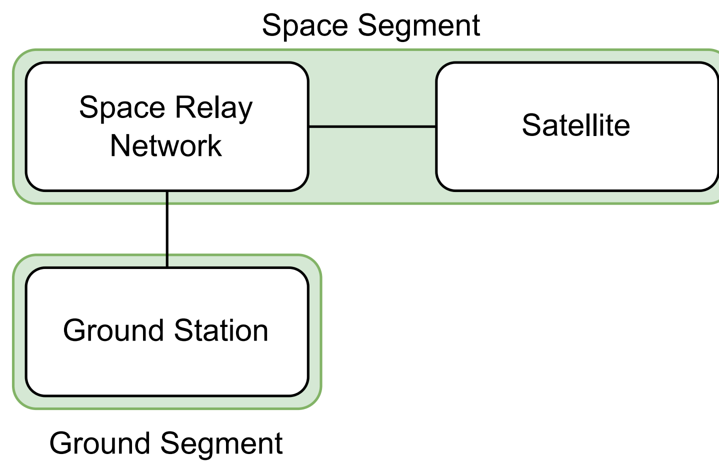

Figure 2.3: Space Relay Network

ISTSAT-1 does not contain the components required to take advantage of SRNs, therefore it is only possible to increase communication availability through ground segment solutions.

GSNs can be characterised as dedicated or shared according to whether they are used by a single mission or shared by multiple [4].

Dedicated GSNs guarantee that their support of a given mission can resort to any ground station at any time. While this configuration provides more control over the GSNs to their owner, it also results in high implementation, ownership, and maintainability costs as they have to be supported by the one operator alone.

Shared GSNs, on the other hand, require each mission to schedule every ground station usage beforehand, preventing the on-demand usage of ground stations, but splitting the cost amongst every participant.

In recent years, with the reduction of LEO satellite's launch prices, there was an increase in the demand for readily available ground stations and companies started providing Ground Stations as a Service (GSaaSs). These services allow a client to rent ground stations from a GSN for some period and only pay for the scheduled time. While GSaaSs are provided by many companies, only a handful are compatible with ISTSAT-1 since its downlink and uplink operate in the Very High Frequency (VHF) and Ultra High Frequency (UHF) bands, respectively. The compatible services will be further described.

2.1.1 CONTEC

CONTEC is a South Korean company that provides GSaaSs, the ability to process the received data and assist with satellite's Launch and Early Orbit Phase (LEOP). Their GSN is composed of 12 ground stations (Figure 2.4) supporting VHF, UHF and S-Band frequencies for Telemetry, Tracking, and Command (TTC). However, it is not mentioned how many support VHF-UHF communications[1].

Figure 2.4: CONTEC's GSN[1]

2.1.2 Leaf Space - Leaf Line

Leaf Line is the GSaaS solution provided by Leaf Space. Even though their GSN is composed of 16 ground stations scattered across 14 locations (Figure 2.5), only 2 ground stations support VHF-UHF communications. The ground stations that support the Radio Frequency (RF) bands used by the ISTSAT-1 for communication are located in Ireland and Spain[2].

[1] http://contec.kr/SpaceGroundService
[2] https://leaf.space/leaf-line/

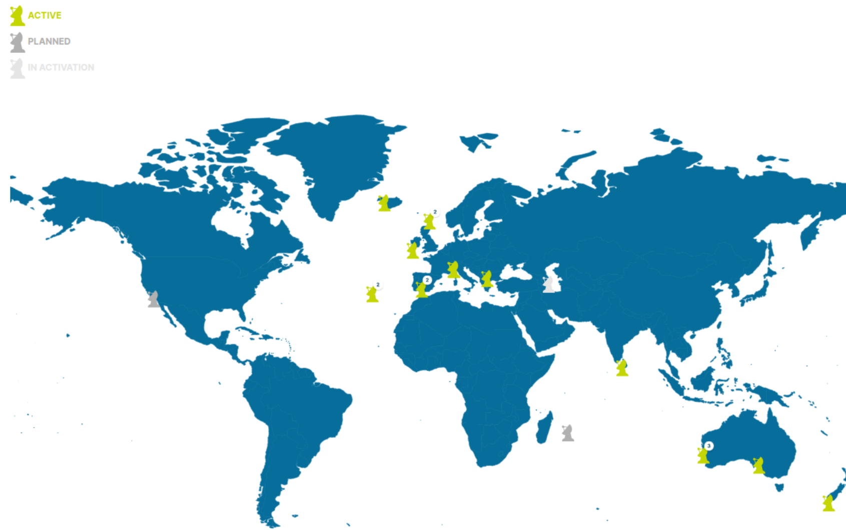

Figure 2.5: Leaf Line GSN[2]

2.1.3 RBC Signals

The GSaaS solution provided by RBC Signals takes advantage of GSN distributed across 55 locations. However, this service, like the one provided by CONTEC (Section 2.1.1), does not specify which of their ground stations supports VHF-UHF bands[3].

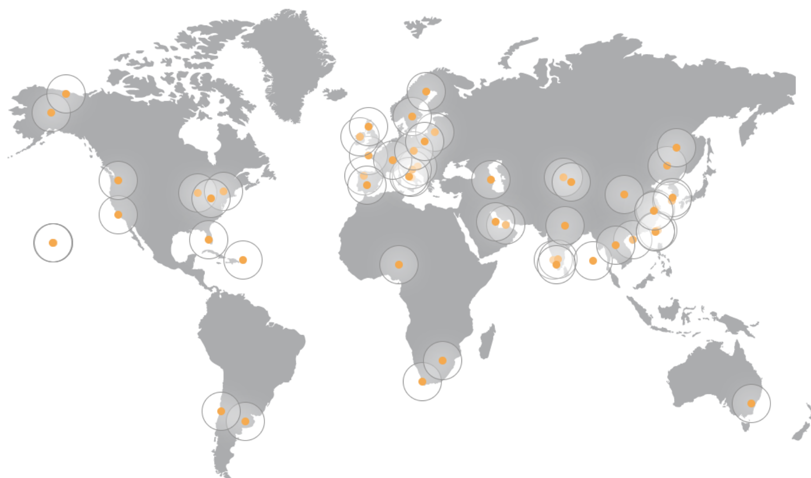

Figure 2.6: RBC Signals' GSN[3]

[3] https://rbcsignals.com/ground-station-as-a-service/

2.1.4 SatNOGS

Satellite Networked Open Ground Station (SatNOGS) is an open satellite GSN platform. The project's objective is to build a full stack of open technologies based on open standards, as well as a full ground station to display the stack developed [5]. To achieve this goals, the project was subdivided into:

- SatNOGS Network: an interface to manage the operation of the multiple ground stations that compose this network.
- SatNOGS DB: a crowd-sourced database with information about the satellites and spacecrafts orbiting Earth.
- SatNOGS Client: a system that processes the scheduled passes from the SatNOGS Network, records the observation and sends it back to the SatNOGS Network.
- SatNOGS Ground Station: a system that receives signals from spacecrafts.

Figure 2.7 provides an example of the architecture a user could follow to incorporate their ground station in the SatNOGS GSN. Since the project is open-source, it is also possible to host the SatNOGS Network and DB modules.

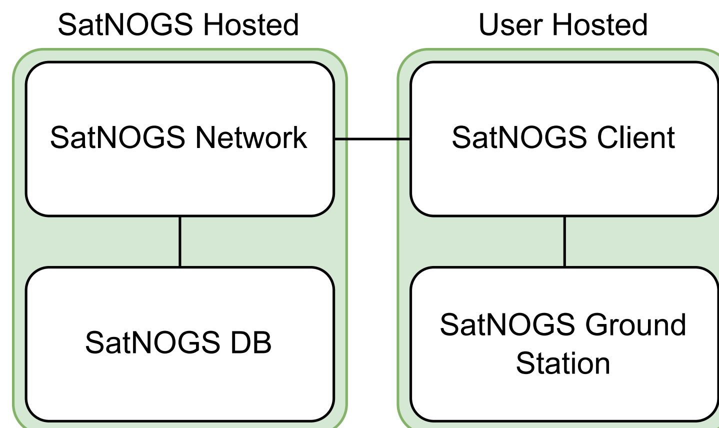

Figure 2.7: SatNOGS high level architecture

Unlike the previously presented GSaaSs, the data received by the SatNOGS Network is publicly available. However, to be able to schedule passes in the system, the user has to contribute at least one ground station to the network. The goal of this measure is to foster the growth of the network.

From the networks discussed in this document, this GSN encompasses the largest amount of ground stations. Even though it does not allow the uplink of data to the satellite, joining this network would enable the reception of telemetry from ISTSAT-1 from all over the globe.

2.2 Ground Segment Automation and Interface

Traditionally the ground segment is operated by humans, known as ground operators. While the specific tasks carried out by these operators differs depending on the satellite being operated, their responsibilities can be reduced to monitoring and controlling the spacecraft. The monitoring process is achieved through the analysis of telemetry and mission data, which in turn are used to decide how the spacecraft should be controlled. Figure 2.8 depicts the feedback loop interaction between the ground operators and the satellite.

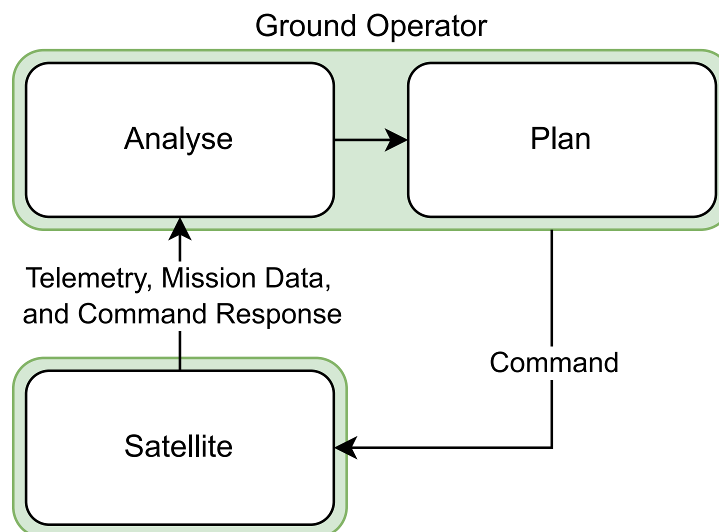

Figure 2.8: Ground Operator - Satellite interaction

Since it is only possible to communicate during a pass, and that passes are typically short for LEO satellites, the time it takes a ground operator between the analysis, planing and commanding phase is critical to the amount of work it is possible to produce during each pass.

This limitation can be mitigated by automating reactions based on received satellite data, and by providing ground operators appropriate interfaces. The first measure ensures ground operators do not have to execute repetitive tasks, which reduces the probability of operation errors and increases data processing and reaction time. While the second, guarantees that when a ground operator is required to interact with the satellite, they can do so efficiently.

2.2.1 Automation Level

There are many ways of quantifying the automation level of a ground segment. One common approach is to rate them according to the light level of the operations room [6]:

- Lights on: refers to a manual system.

- Lights dim: refers to a semi-automated system where some of the operations are automated, like the collection of telemetry.
- Lights off: defines a system where almost all functions are automated.

2.2.2 Mission Operation software

The software that enables the ground segment has a crucial role in the success of the mission it supports. It is typically composed of multiple systems that interact with each other, requiring a big effort and investment to develop. In order to mitigate these costs, it is usual to resort to generic solutions, which are developed taking into account common needs across multiple satellite missions. Some of them will be further discussed.

2.2.2.A COSMOS

COSMOS is the open-source command and control solution provided by Ball Aerospace. It equips users with the required tools to control and monitor embedded systems. COSMOS is comprised of 11 applications[4]:

- **Command and Telemetry Server**: provides the status information of the system.
- **Limits Monitor**: displays telemetry values that are, or were, out of the expected limits.
- **Command Sender**: provides a Graphical User Interface (GUI) to send single commands to the target system.
- **Script Runner**: a GUI that allows editing, executing scripts, shows which line is being executed and alerts the user when a failure occurs.
- **Packet Viewer**: a real-time viewer for telemetry, showing the last received telemetry values.
- **Telemetry Viewer**: a customizable dashboard that allows the user to open, arrange and close windows showing different telemetry visualizations.
- **Telemetry Grapher**: graphs telemetry values over time.
- **Data Extractor**: processes logged data and extracts data into Comma-Separated Values (CSV) files.
- **Data Viewer**: a text based visualization for displaying telemetries like logs, that cannot be efficiently viewed through the other methods.

[4] https://ballaerospace.github.io/cosmos-website/docs/v5/requirements

- **Timeline**: an interface to schedule actions.
- **Admin**: provides administrative functions to manage the system's behaviour.

Figure 2.9 depicts how the different components of COSMOS interact in order to provide the target functionality.

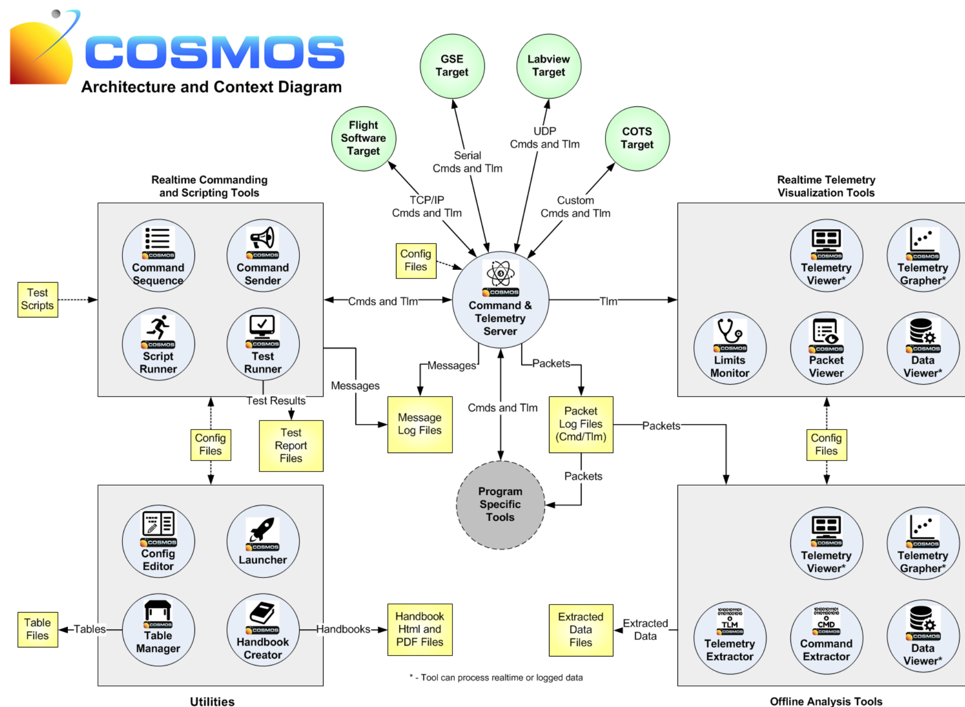

Figure 2.9: COSMOS Architecture[4]

2.2.2.B Major Tom

Major Tom is a cloud based command and control solution provided by Kubos[5]. In addition to providing an interface to send commands to the spacecraft and display the received data, it also affords the possibility of resorting to the GSNs pre-integrated into their system for communication. Figure 2.10 illustrates Major Tom's architecture.

The user is responsible for building the modules represented in orange:

- **Gateway**: in its simplest form, this module translates data between satellite and Major Tom's notation and vice-versa.
- **Data Processing Pipeline**: is any service capable of receiving data produced by the gateway in order to further process it.

[5] https://kubos.notion.site/Major-Tom-Integration-Architecture-d577cc8712f74842b1e540b192558c5b

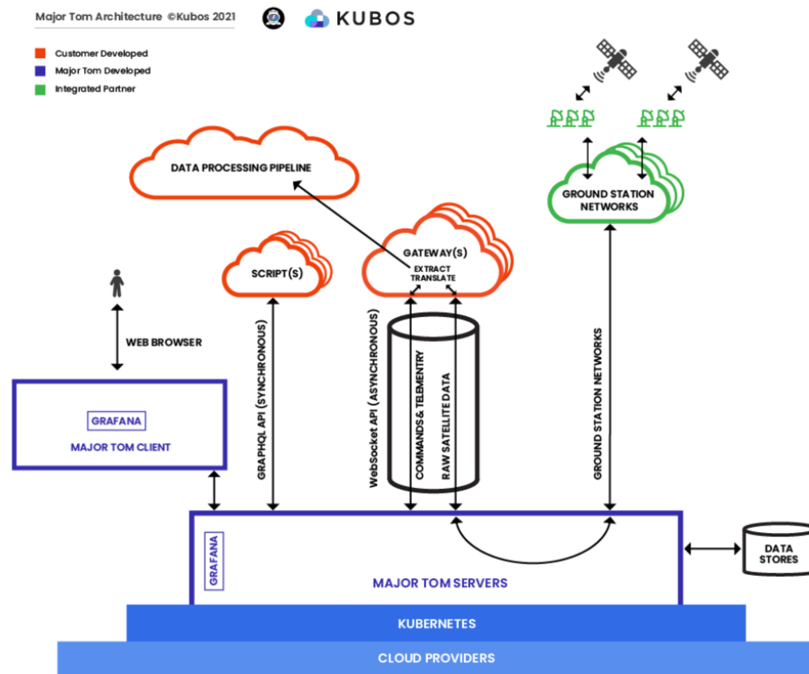

Figure 2.10: Major Tom Architecture[5]

- **Script**: is any program that uses Major Tom's GraphQL[6] Application Programming Interface (API) in order to programmatically issue commands on Major Tom.

[6]https://graphql.org/

3

Ground Segment of ISTSAT-1

Contents

3.1 Initial Architecture 21

3.2 Proposed Architecture 29

During the development phase of ISTSAT-1, the primary focus was on the satellite. Therefore, typically, the feature implementations in the ground segment were motivated as a way of testing new satellite functionalities.

The development of the ground segment around the spacecraft's needs led to an unstructured design, whose architecture and flaws are exposed. After evidencing the shortcommings of some of the approaches taken, a solution is proposed.

3.1 Initial Architecture

The ground segment is divided into the modules shown in Figure 3.1. The coloured blocks represent hardware systems while the white blocks represent relevant processes running inside them. The solid links between blocks represent Transmission Control Protocol (TCP) connections while dashed links represent Universal Serial Bus (USB) communications.

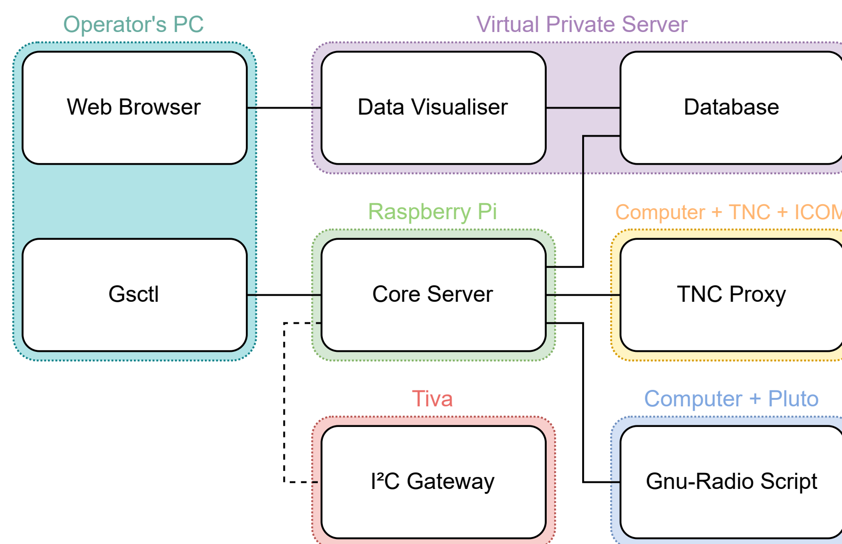

Figure 3.1: Ground Segment Architecture

3.1.1 Protocols

Typically the ground segment communicates with the satellite via radio, but, during the development and testing phases, it is also possible to communicate with each ISTSAT-1 subsystem through a serial bus.

Figure 3.2 is a diagram of the protocols used for radio communication between the ground and space segments. On the left is the protocol name, while on the right the layer of the Open Systems Interconnection (OSI) model it belongs to.

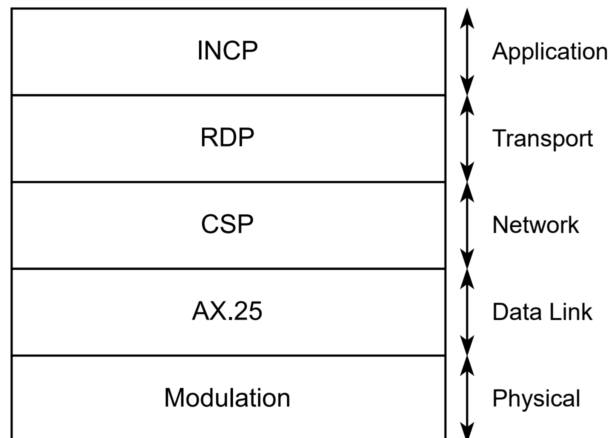

Figure 3.2: Protocol Stack

ISTNanosat-1 Command Protocol (INCP) is used to issue commands to the spacecraft. Reliable Data Protocol (RDP) guarantees the reception of messages, avoiding duplicates while being delay and sequence tolerant [7]. Cubesat Space Protocol (CSP) allows the addressing of the cubsat and of each subsystem [8]. Amateur X.25 (AX.25) is responsible for transferring data encapsulated in frames between nodes, and detecting errors introduced by the communication channels [9]. Modulation defines how the information is is encoded into radio waves.

Figure 3.3 illustrates the protocols used for serial communication with the spacecraft. The INCP messages are directly sent in via the I^{2}C bus to the target subsystem, easing the development and testing of the spacecraft.

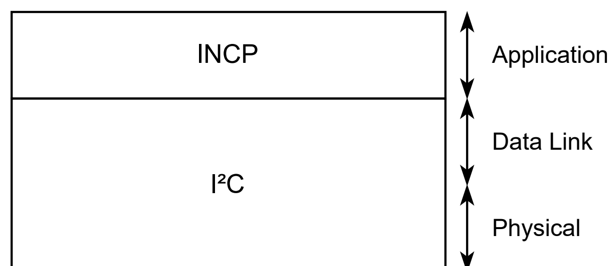

Figure 3.3: Protocol Stack

3.1.2 Ground Segment Control (GSCTL)

The GSCTL is a client program that equips the ground operator with a CLI to access the functionalities exposed by the Core Server. Information exchanged between both nodes happens through the javascript object notation (JSON) format.

Using this UI it is possible, not only, to command the satellite, but also to run built-in operations. Operations is the designation used to define a dynamic flow of commands whose behaviour depends on the satellite responses. The available operations cover the most used scenarios. Currently these operations are mostly used to automate functional tests on the spacecraft.

This interface, while effective for proficient operators due to its flexibility, can intimidate those trying to learn to operate the spacecraft, as it requires the user to constantly recall the names of commands, steepening the learning curve. On the other hand, GUIs organise information in a way that facilitates its mastering. They also account for the majority of UIs, therefore, users are typically more comfortable using them. For these reasons, it is important to create a GUI that enables the ground segment operation.

3.1.2.A Commands

Through GSCTL, it is possible to issue commands, request data, and configure ISTSAT-1. To execute a procedure exposed by this interface, the ground operator has to execute ”gsctl” with the appropriate arguments. The first argument is the name of the procedure, and the following arguments are interpreted by said procedure. After the execution of a procedure, its result is printed to the standard output.

Listing 3.1 is an example of how a ground operator might execute the command ”add” on the satellite. In this example, the procedure is ”cmd”, ”OBC” is the subsystem of the satellite that receives and process this command, and ”1” and ”3” are the arguments for the command ”add”.

```
1 gsctl cmd OBC add 1 3
```

Listing 3.1: Command example

To inspect the state of the spacecraft it is possible to request specific telemetry values. Listing 3.2 depicts the command used to request the uptime from the satellite. The procedure used is ”data-req”, ”OBC” is the target subsystem, and ”time_boot” is the name of the telemetry variable.

```
1 gsctl data-req OBC time_boot
```

Listing 3.2: Data Request example

The satellite’s behaviour can be manipulated through the use of configurations. Listing 3.3 illustrates how the value of a configuration can be requested. The name of the procedure is ”config-get”, ”OBC” is the target subsystem, and ”test_config” is the configuration whose value was requested.

```
1 gsctl config-get OBC test_config
```

Listing 3.3: Config Get example

Listing 3.4 demonstrates how the value of a configuration can be changed. The procedure used is ”config-set”, ”OBC” is the target subsystem, ”test_config” is the configuration whose value is to be

changed, and ”3” is the new value.

```
1 gsctl config-set OBC test_config 3
```

Listing 3.4: Config Set example

3.1.2.B Operations

Operations are procedures whose behaviour depends on there responses of the satellite. There are operations for the most common procedures, avoiding the need for the ground operator to issue multiple repetitive commands. Listing 3.5 exemplifies how to execute the operation ”normalsat”.

```
1 gsctl op run normalsat
```

Listing 3.5: Operation example

3.1.3 Core Server

The Core Server is a module developed by the ISTSAT-1 team that exposes an API to enable client programs to operate the satellite. It also processes mission and telemetry data received from the spacecraft, storing it persistently in a Database. To send information to the satellite, INCP messages need to be generated and formatted according to each gateway specification. When data is received, the inverse process has to take place.

Gateways are nodes that form a passage between two networks operating with different protocols. In the specific case of ISTSAT-1, there are two gateway types, ground stations and the Electrical Ground Support Equipment (EGSE).

3.1.4 EGSE

EGSE is an integrated system of electrical testing solutions to ensure that the satellite is operating according to the specification.

In the specific case of ISTSAT-1, the EGSE can be used to program the satellite and to interface with the spacecraft's I^2C bus. The EGSE Gateway is the module responsible for receiving INCP messages via USB and sending them to the satellite's I^2C bus. This behaviour is evidenced in Figure 3.4.

3.1.5 Ground Station

Each ground station is comprised of at least a radio, an antenna, and a controller that provides an API to exchange data with the satellite. The ground segment has one ground station using a traditional radio transceiver and another using a Software Defined Radio (SDR). Both ground station controllers only

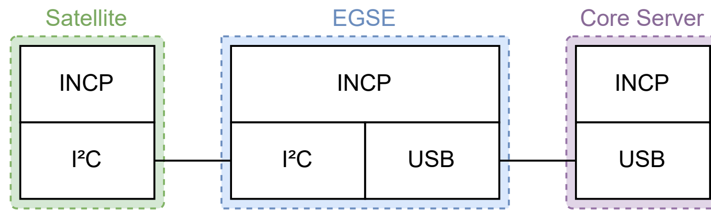

Figure 3.4: EGSE Communication Protocol Stack

allow the sending and receiving of frames. It is not possible to change modulation or the pointing of the antenna through software procedures.

To communicate using any of these ground stations, the operator must request the Core Server to establish a TCP connection with the appropriate controller, typically through GSCTL. This approach is not ideal, as the ground operator needs to know when each ground station is up, and what is its Internet Protocol (IP) address. If the ground station is not hosted on the same network as the Core Server, it might be necessary to configure the ground station's network to allow foreign connections. However, depending on the network, it may not be possible to configure it in such way.

3.1.5.A Traditional Ground Station

Figure 3.5 illustrates the components of the traditional radio transceiver ground station.

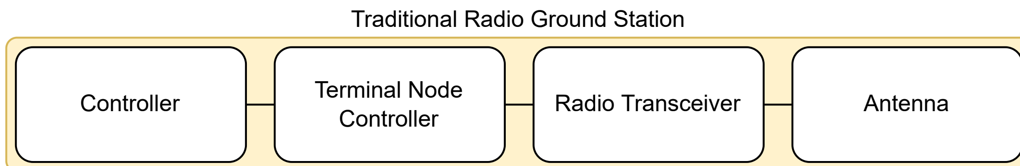

Figure 3.5: Traditional Radio Ground Station

In this ground station the controller receives the address, control, and information fields of the AX.25 frames via the TCP socket. This information is then sent to the Terminal Node Controller (TNC) which frames it according to the High-Level Data Link Control (HDLC) protocol, modulates them using Audio Frequency-Shift Keying (AFSK), and sends the audio to the radio transceiver. The transceiver then frequency modulates the audio signal into the RF band and transmits it using the antenna. When receiving information, the reverse operations are done by the same ground station components.

Figure 3.6 illustrates the functionality provided by this ground station.

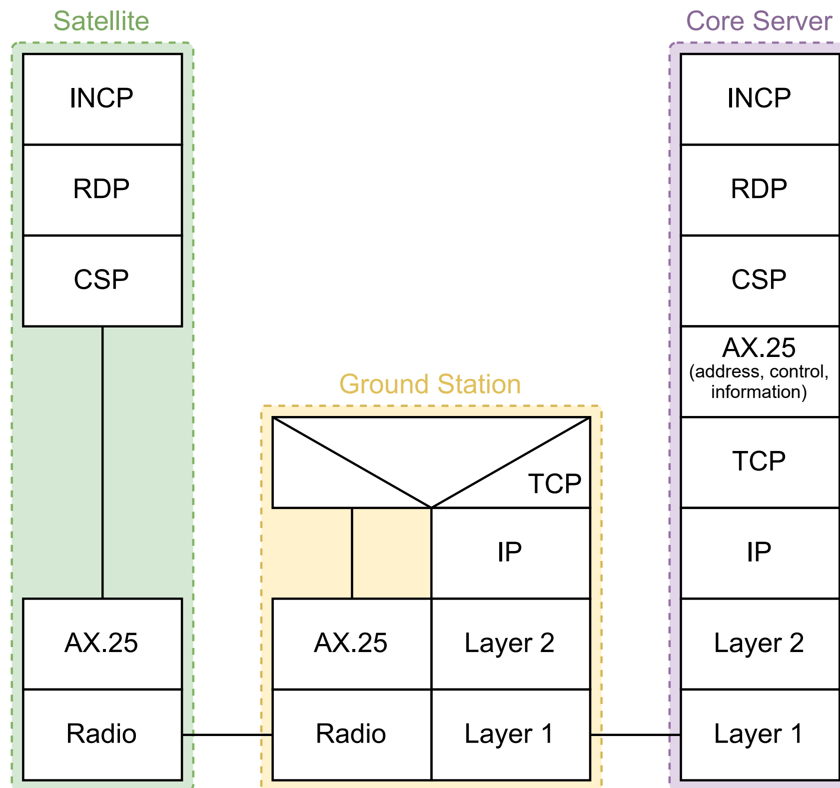

Figure 3.6: Traditional Ground Station Communication Protocols

3.1.5.B SDR Ground Station

Figure 3.7 exposes the components of the SDR ground station.

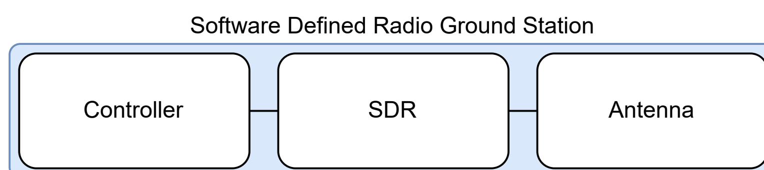

Figure 3.7: Software Defined Radio Ground Station

Differently to how the previously discussed ground station operates, this controller receives the AX.25 frames. It then, processes and forwards them to the SDR. The SDR ensures that they are transmitted using the antenna. As with the traditional ground station, when receiving information, the reverse operations are done by the same components.

Due to the nature of SDRs, it is possible to change the used frequency modulation. This is defined by the GNU-Radio script running in ground station controller.

Figure 3.8 evidences the functionality provided by this ground station.

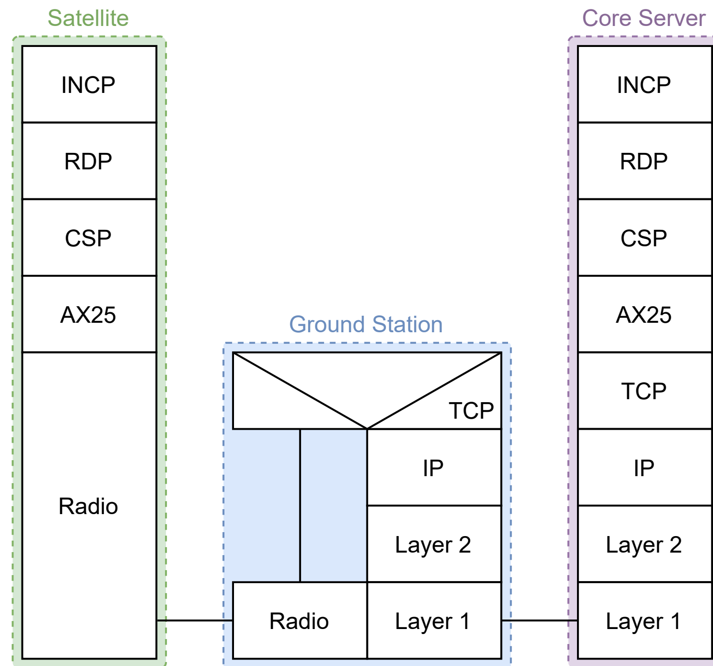

Figure 3.8: SDR Ground Station Communication Protocols

3.1.5.C Ground Station Interface

The approach defined in both ground station descriptions establishes a RDP connection between the Core Server and the satellite. Since the RDP connection is established with the Core Server and not the Ground Station, the RDP segments cannot be acknowledged by the Ground Station and need to be proxied to the Core Server via a TCP tunnel, frequently crossing many Internet nodes. The proxying of data increases the latency in the acknowledgement of RDP segments which may lead to the unnecessary retransmission of segments, thus negatively affecting the goodput of the connection.

3.1.6 Gateway Interface

The communication interfaces of the multiple Gateways are not standardised. This lack of standardisation is notable not only trough the distinct utilized connection types, but also from the used data formats. The Core Server resorts to TCP when establishing a connection with either ground station and to USB when connecting to the EGSE gateway. The data formats used are AX.25 fields, AX.25 frames or INCP messages depending on whether the target gateway is a traditional radio ground station, a SDR based ground station or a EGSE gateway, respectively.

Figure 3.9 illustrates de heterogeneity between gateway interfaces. The dashed connection represents a USB connection, while the remaining links stand for TCP connections.

The utilisation of different connection types and data formats is tied to an increased program orga-

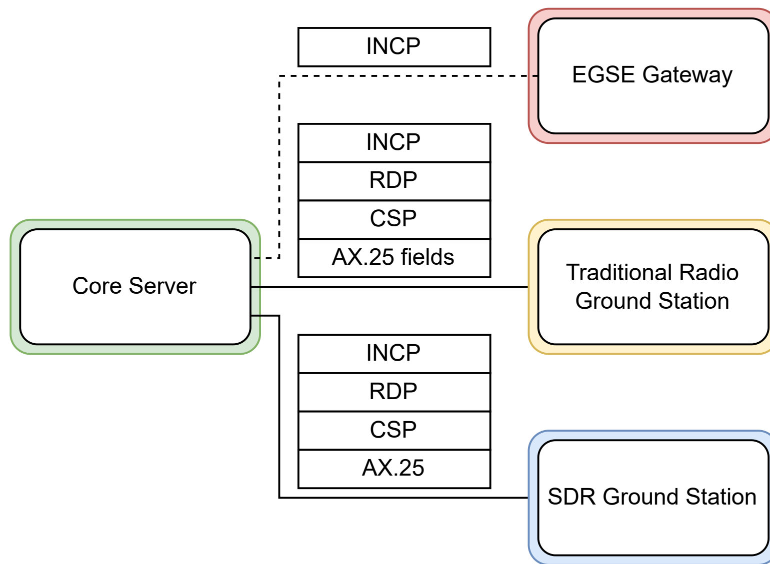

Figure 3.9: Lack of standardisation of gateway interfaces

nization complexity, leading to less maintainability. It also results in the unnecessary proxying of data affecting the goodput between the Core Server and the satellite.

3.1.7 Automation

In order to execute a satellite operation, a connection between a GSCTL instance, the Core Server, a gateway and the satellite must be established. This connection is required since the automation's control flow is decided in the GSCTL.

Even though the capabilities provided by GSCTL can be leveraged to schedule the execution of operations, the process to reach such goal is not practical nor robust. As an example, the operator could schedule the operation to run in their computer during a satellite pass. However, this would require their computer to be on, and connected to the internet at the specified time and during the full pass duration.

It would be preferable to allow the schedule of operations directly in the Core Server, entrusting it with the chore of executing the appropriate operation at the correct time. This solution is more robust as it reduces the number of points of failure.

3.1.8 Data Visualiser

The data visualiser complements GSCTL, allowing the operator to visualy analyse the telemetry and mission data stored in the database. This is achieved through a web based GUI that enables the creation of different panels, according to the data organisation.

This service is running in Caronte, a Virtual Private Server (VPS) instance hosted at Instituto Superior

Técnico (IST).

3.1.9 Security

Connections between the Core Server and ground stations or GSCTL happen through TCP sockets. The data exchanged between these parties is in plain text and there is no authentication mechanism in place. Therefore, a malicious user is able not only to listen to the traffic but also issue arbitrary requests to the Core Server.

3.2 Proposed Architecture

While re-designing the ground segment of ISTSAT-1, the initial architecture was taken into account trying to reuse most of the components in order to ease the development efforts. The proposed solution aims to solve the exposed issues with the previous design.

Figure 3.10 depicts the proposed changes to the ground segment architecture.

The Core Server should be hosted in a dedicated machine, along with the Data Visualiser and database, providing an increase in processing power and availability to the service.

Each gateway should initiate the connection process with the Core Server, ensuring that there is no need for the ground operator to monitor the state of these connections. As soon as the system is available, it's controller will establish a connection with the Core Server. This controller also exposes the system's API.

3.2.1 Communication Protocols

Instead of the previously exposed heterogeneity in the message format accepted by each gateway, every gateway handles INCP messages. For the ground stations, the communication protocol stack is represented in Figure 3.11.

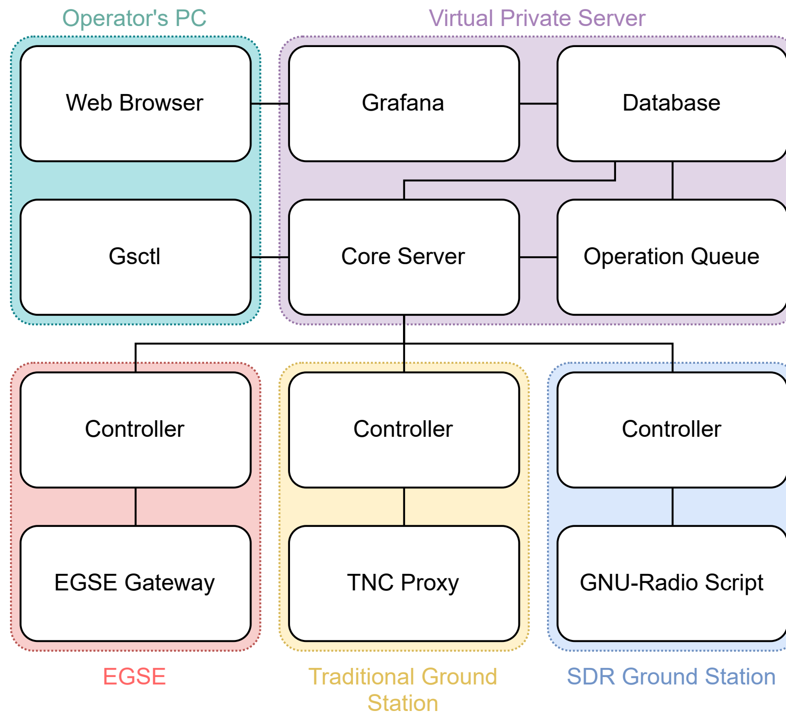

Figure 3.10: Ground Segment Architecture

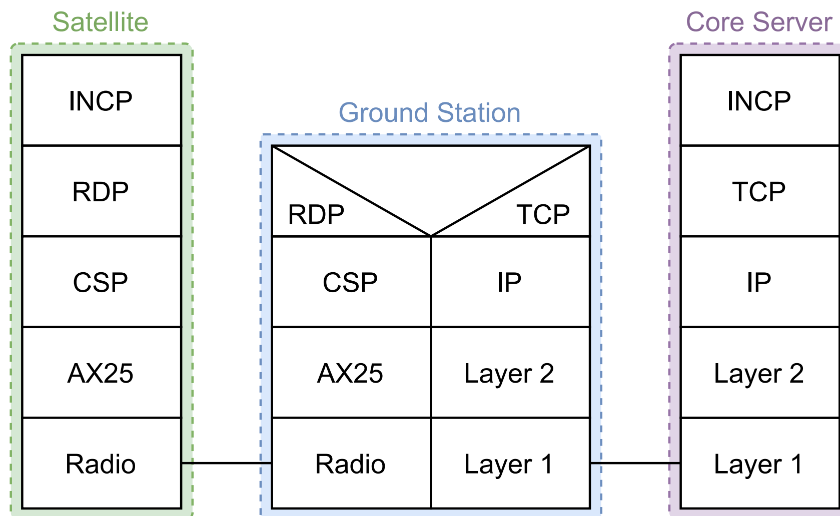

Figure 3.11: Ground Station Communication Protocol Stack

The communication protocol stack for the EGSE is represented in Figure 3.12.

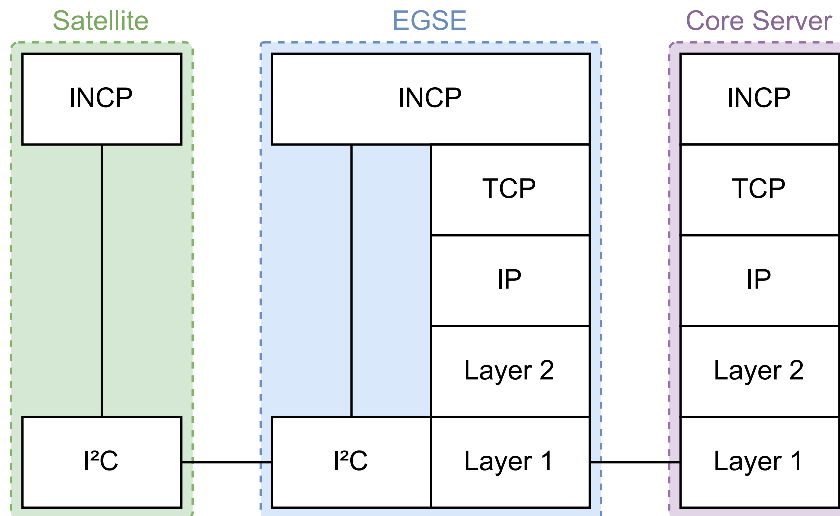

Figure 3.12: EGSE Communication Protocol Stack

3.2.2 Stand-alone Electrical Ground Support Equipment (SEGSE)

The presented architecture assumes that there is always an Internet connection. There are cases where this assumption might not be correct, preventing the operation of the satellite. It is safe to presume that this scenario is only plausible during the debug of the satellite in remote facilities, since, during flight, communication will happen through ground stations that, typically, are in a permanent location with internet access. Notwithstanding, it is necessary to define a system that still allows the debug of the satellite even without an internet connection.

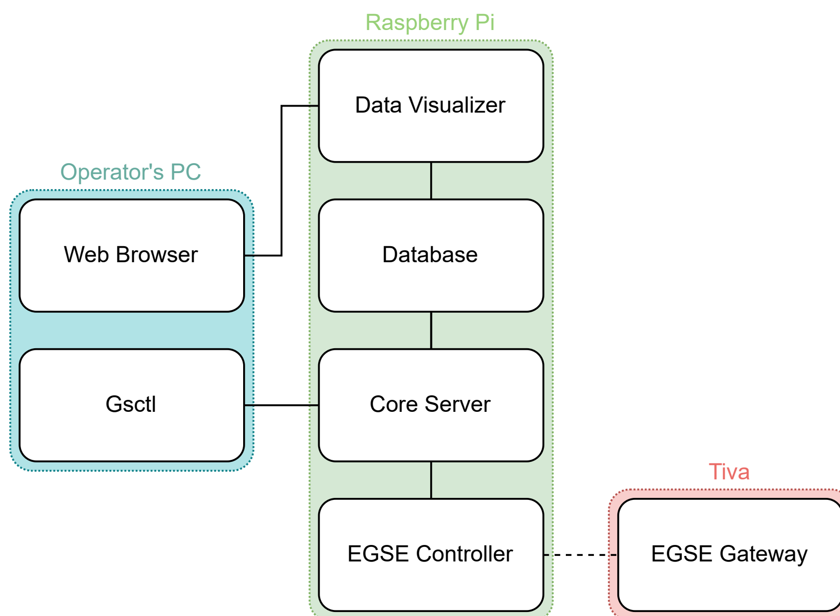

Figure 3.13: Stand-alone EGSE

This system is referred to as SEGSE and is represented in figure 3.13. It is comprised of a general purpose computer and a microcontroller board, connected via USB. The computer is running the Core Server, EGSE Controller, Data Visualiser, and a local database. While the microcontroller is running the EGSE Gateway.

In order to operate the SEGSE, the operator's computer is linked to the computer directly, using an RJ45 cable to connect both devices via Ethernet.

This configuration is possible because all the software modules are connected through TCP. For this reason, their host machine can be abstracted, allowing them to run on any computer without modifications to the source code.

3.2.3 Operation Queue

The proposed design keeps the functionality that allows GSCTL to execute operations, while allowing those same operations to be scheduled in the Core Server. This is achieved by the Operation Queue, a software module, that ideally runs in the same machine as the Core Server and is responsible for establishing the connection between them.

To schedule an operation, the operator, notifies the Core Server about its name and desired timestamp for the execution. Then, if the Operation Queue is connected, the Core Server forwards it the request. The Operation Queue registers the information regarding the execution of this operation, and saves it in the database. As soon as the time for execution arises, the Operation Queue executes the desired operation, behaving like a GSCTL instance.

3.2.4 Graphical User Interface

In order to incorporate a new UI into this ground segment, there are two main approaches. One is to create a program that runs on the operator's computer and directly calls the endpoints exposed by the Core Server's API. This is the technique employed by GSCTL. Another approach would be to host a service, like a web server, that provides a UI, translating user requests into calls to the Core Server API. Figure 3.14 illustrates this interaction using only the relevant blocks.

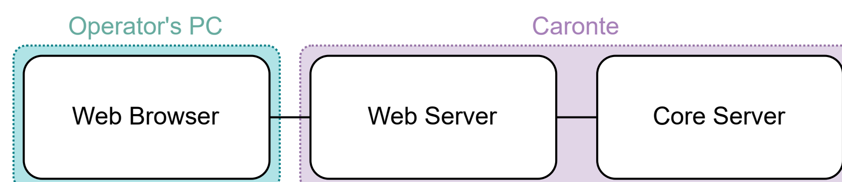

Figure 3.14: Web Server

4 Implementation

Contents

4.1 Core Server Interfaces and Gateway Behaviour 35
4.2 Graphical User Interface 43
4.3 Alternative Graphical User Interface 51
4.4 Operation Queue 54

At the time of implementation, the satellite was at the end of the final test campaigns. As such, modifications to the current design took into account the already written test procedures and, where possible, respected them, avoiding changes to already approved documents. That being said, the proposed solution was implemented in two distinct phases.

During the first phase, the ground segment's architecture was altered to standardise the communication mechanisms to interface with the Core Server. These changes took priority over the implementation of new functionality because they could invalidate already written procedures. By implementing them first, if backwards compatibility was broken, the affected documents could be corrected as soon as possible. It also means that the documents that still needed to be written would already take into account the new architecture.

The second implementation phase, consisted on adding new functionality to the ground segment. A GUI was included and the automation process improved with the development of the Operation Queue.

The implementations process of each augment to the ground segment is discussed throughout this chapter.

4.1 Core Server Interfaces and Gateway Behaviour

Core Server's gateway interface had to be modified to allow the sending and receiving of INCP messages, as well as the commanding of each gateway, depending on its capabilities. Ideally this traffic would be sent over one connection, reducing the number of connections, and thus the load on the server.

The Core Server, as the vast majority of the ground segment, is implemented in the Python[1] programming language, and uses the Trio[2] library to manage asynchronous input and output. This decision was made due to the language's high-level nature, and emphasis on code readability.

The current client API provided by Core Server resorts to Trio-RPC. Trio-RPC is a Remote Procedure Call (RPC) library, developed in-house, that takes advantage of Trio in order to process requests asynchronously. However, this library only allows communication through a traditional client-server model, where the client sends a request and the server answers with a response. Although this approach is acceptable in the case of GSCTL, it is unacceptable in this particular circumstance, as gateways must be able to push and receive INCP messages as soon as possible.

Since Trio-RPC was developed in-house, it was altered to allow servers to call functions registered in the client, thus enabling it to be used for in this scenario. It was also improved to support Transport Layer Security (TLS) and execution of multiple requests concurrently.

[1] https://www.python.org/
[2] https://trio.readthedocs.io/en/stable/

4.1.1 Trio-RPC

This Python library allows the execution of asynchronous remote functions. The most important component of this system is the class RPCNode. It enables the sending and receiving of requests and responses, as well as their handling. Since the interface with this class provides a lower level interface than most applications will need, classes to abstract the intricacies of the system were developed.

These classes are named RPCClient and RPCServer. While both expose methods to register functions that can be remotely called. The RPCServer contains a method to start waiting for new connections, and the RPCClient one for establishing them.

Whenever a connection is established between a RPCServer and a RPCClient, they start a new task to handle it. This task is controlled by an instance of the class RPCNode, lives for the duration of the connection, and starts at least two child tasks, an handler and an executer. The number of executor tasks is the same as the number of concurrent requests the RPCNode is able to process.

Each instance of RPCNode exposes a method that allows the execution of functions in its remote counterpart. This is done by sending a request message to the connected node, which processes it and responds with a response message. Messages follow the JSON format.

4.1.1.A Messages

There are two types of messages, requests and responses.

Request messages should contain the following members:

- id: a number that identifies the request.
- req: a string containing the name of the function to be executed remotely.
- args: an object whose members are either the position or name of the parameters to whom the argument belongs.

Listing 4.1 represents a sample request message for a call to a function named ”fn_name” with a positional argument ”1” and a keyword argument ”2” for the parameter ”a”.

```
1 {
2   "id": 0,
3   "req": "fn_name",
4   "args": {
5     "0": 1,
6     "a": 2,
7   }
8 }
```

Listing 4.1: Sample request message

Response messages should contain the following members:

- id: the number on the ”id” field of the request that trigged this response.
- ret: only present if the request was successful, contains the return value of the call.
- err: only present if the call was not successful, contains the error message produced.

Listing 4.2 represents a sample response for the request made in Listing 4.1, admitting that the call was successful and the return value was ””.

```
1 {
2     "id": 0,
3     "ret": "",
4 }
```

Listing 4.2: Sample response message

4.1.1.B Concurrency

While developing the handling mechanism for concurrent requests some important design decisions had to be made.

Figure 4.1 depicts how a function call between two nodes would be processed. This process started as soon as ”Task 1” invoked the ”call” method of RPCClient’s RPCNode. This method, generated a request message that was then sent to the RPCServer. After sending the request message, the task blocked until a response was delivered or a timeout triggered. When the RPCServer’s handler task received the message, identified that it was a request, and sent it to the executer task. This task, then, called the appropriate method, wrapped the return value in a response message and sent it to the RPCClient. The RPCClient’s handler task, identified it as a successful response message and delivered the return value to the previously blocking task.

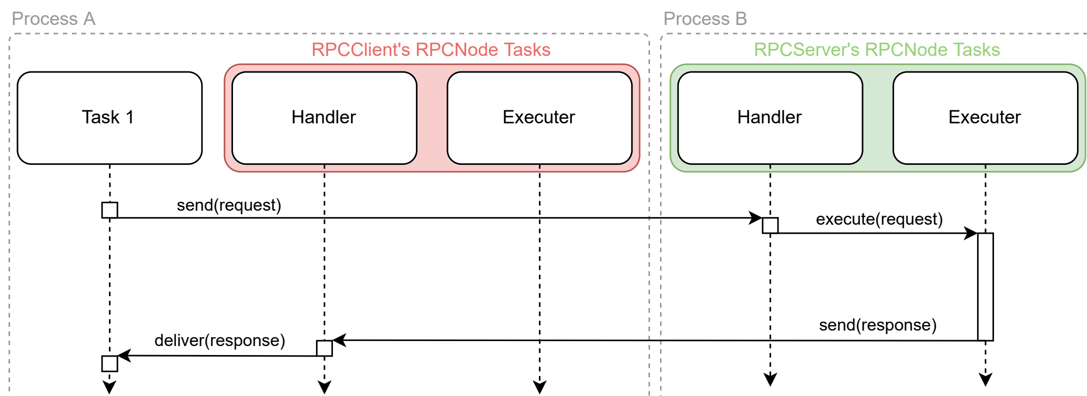

Figure 4.1: Remote Function Call Sequence Diagram

A call from the server to the client would be represented by a similar diagram. A task running in

Process B would invoke the ”call” method of RPCServer’s RPCNode, which would initiate a similar behaviour.

Figure 4.2 illustrates how a RPCNode, with only executor, would handle two concurrent requests from the same process. The scenario evidences that the executer task is only able to process new requests as soon as the one being handled has terminated its execution, even if the function being run had checkpoints allowing the scheduler to change the running task.

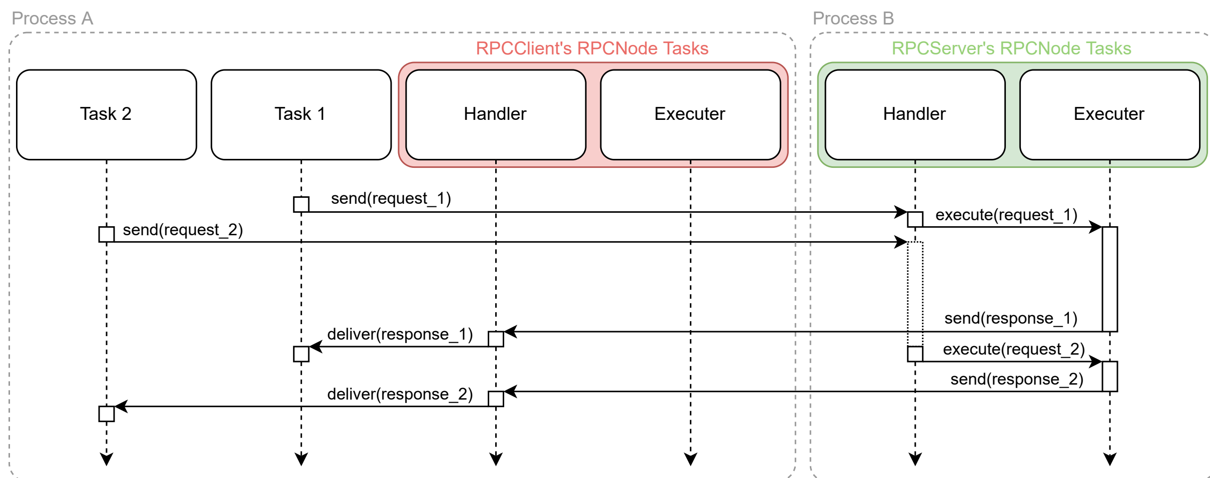

Figure 4.2: Two Remote Function Calls with One Executer Sequence Diagram

Figure 4.3 clarifies how the RPCNode would be able to handle two requests concurrently if the same number of executer tasks were running. This potentiates the earlier return of requests whose functions have a shorter execution time.

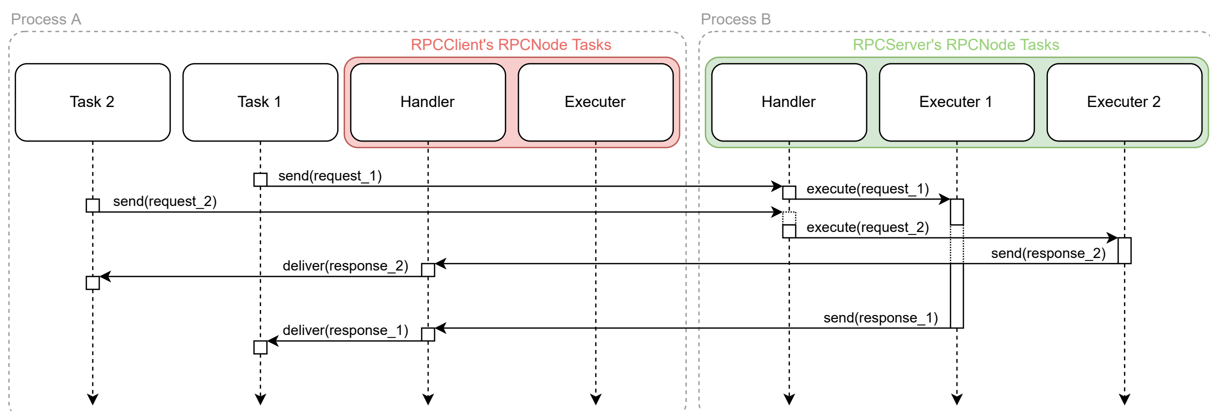

Figure 4.3: Two Remote Function Calls with Two Executers Sequence Diagram

Instead of relying on executer tasks, another solution to allow the execution of concurrent function calls would be to create a new task per request. However, the time it takes to launch a new task would increase the processing time of each request. This solution also enables malicious clients to spawn an

arbitrary number of tasks, eventually reducing the available time to handle other requests. This attack is known as a denial-of-service.

4.1.1.C Syntax

The way a user interacts with a library greatly influences its usability and adoption. This library's API tries to accommodate most use cases by providing a generic client and server implementation. While this implementation should fit most use cases, it is expected that some applications may require changes to the default behaviour. For this reason, its inner workings are exposed, allowing the user to implement their own version of either the client or server, to better suit their needs.

A server to add two numbers and the respective client will be implemented. The server will also periodically ping the client. This implementation helps showcase the library's API.

A – Server The first step to develop the server is to instantiate it by providing the address, port, and callback hooks to be notified as soon as a client is connected or disconnected. The hooks allow the server to keep track of the connected clients. Therefore if the server does not intend to invoke remote function calls on clients, they are not needed. Listing 4.3 portrays how the server was instantiated for the example in question. The used hooks are unnamed functions that add or remove the RPCNode corresponding to a specific client connection to the list "nodes", according to whether they connect or disconnect from the server.

```
1 rpc_server = RPCServer(
2     host="localhost",
3     port=8089,
4     hook_on_connect=lambda node: nodes.append(node),
5     hook_on_disconnect=lambda node: nodes.remove(node),
6 )
```

Listing 4.3: RPCServer instantiation

After creating a new server, the functions available to be remotely called must be registered. In Listing 4.4 the function "add" is registered.

```
1 async def add(a, b):
2   return a + b
3
4 rpc_server.register_function(add)
```

Listing 4.4: Register Function

When all functions are registered, the server can be started by running its "start" method. This method receives a nursery, the context used by Trio to spawn child tasks, as argument. Listing 4.5 illustrates this process, as well as the creation of a new task, "ping_clients", that invokes the ping function

in all clients, every second. The ”with” block blocks until all the tasks issued by ”nursery” are terminated.

```
async def ping_clients():
  while True:
    for node in nodes:
      await node.call("ping", args=[], kwargs={})
    await trio.sleep(1)

async with trio.open_nursery() as nursery:
  await rpc_server.start(nursery)
  nursery.start_soon(ping_clients)
```

Listing 4.5: Start Server

B – Client When developing a client, the RPCClient class must be instantiated and similarly to the procedure followed with the server, the functions intended to be remotely called should be registered. In this specific case the function ”ping” is registered. Listing 4.6 exemplifies how this is achieved.

```
def ping():
  return "pong"

rpc_client = RPCClient()
rpc_client.register_function(ping)
```

Listing 4.6: Instantiate Client and Register Function

After registering the functions, a connection should be established with the server. RPCClient's method ”connect” receives a host, port, and nursery as arguments and ensures that this connection is established, returning a ServerProxy object, ”proxy”. The goal of ServerProxy objects is to wrap RPCNodes, providing syntactic sugar to the invocation of remote functions. Listing 4.7 illustrates the connection process, as well as the call of the remote function ”add”, using the sugared syntax (the de-sugared syntax is commented above). Before disconnecting from the server, a sleep of 1 second is added to ensure that the server is able to ping this client.

```
async with trio.open_nursery() as nursery:
  proxy = await rpc_client.connect(
    host="localhost",
    port=8089,
    nursery=nursery,
  )
  # node = proxy._rpcnode
  # print(node.call("add", args=[1,20], kwargs={}))
  print(await proxy.add(1, 20))
  await trio.sleep(1)
  await rpc_client.disconnect()
```

Listing 4.7: Connect and Execute Remote Function

4.1.1.D Security

Both the RPCClient and RPCServer support TLS encryption and peer authentication facilities. To resort to these features, the appropriate SSLContext should be given as an argument to the ”connect” method of RPCClients and to the constructor of RPCServers.

4.1.1.E Backwards Compatibility

The interface to invoke remote calls has been kept unchanged. When porting code using the old version of this library, the only sections requiring intervention were related to the RPCClient’s connection process, as now it requires a nursery.

4.1.2 Core Server

The Core Server exposes two endpoints, the previously defined client API, and a new API to handle incoming connections from gateways.

The client API allows client programs to issue commands to the satellite, while the gateway API enables the handling of gateways. When a gateway establishes a connection with the Core Server, it must provide its identifier and capabilities. The identifier is the label used to uniquely identify each gateway. The capabilities are a list of functionalities available for the connected gateway. Since different gateways might provide different functionalities, it is necessary to inform the Core Server about which functionalities are exposed by which gateway. The available capabilities are ”gateway”, ”ground_station” and ”tiva”. Each gateway can implement any combination of the three.

The ”gateway” capability ensures that the gateway allows remote calls to the function ”send” that sends INCP messages, and that as soon as it receives an INCP message from the satellite, it pushes it to the Core Server by invoking the function ”deliver_incp_message”. Figure 4.4 illustrates this interaction.

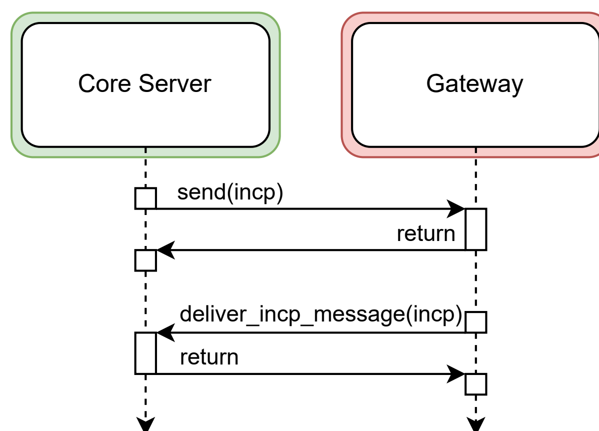

Figure 4.4: ”gateway” Capability

Unlike ”gateway”, the ”ground_station” and ”tiva” capabilities do not notify the Core Server as soon as some event happens. Instead they just handle requests made by the Core Server. The ”ground_station” capability exposes functions related to ground station control, while ”tiva” implements functions to manage the Tiva-C device utilised by the EGSE to debug the spacecraft.

When defining a new function to be invoked through the client API, the developer needs to annotate which capabilities it requires. This allows Core Server to inform the client when they issue an API call using a gateway that does not have all the required capabilities to execute the desired function.

4.1.3 Ground Station

This gateway implements the ”gateway” and ”ground_station” capabilities. It is composed of three main software modules, the controller, GNU-Radio script and modulation changer. Figure 4.5 shows how they are interconnected.

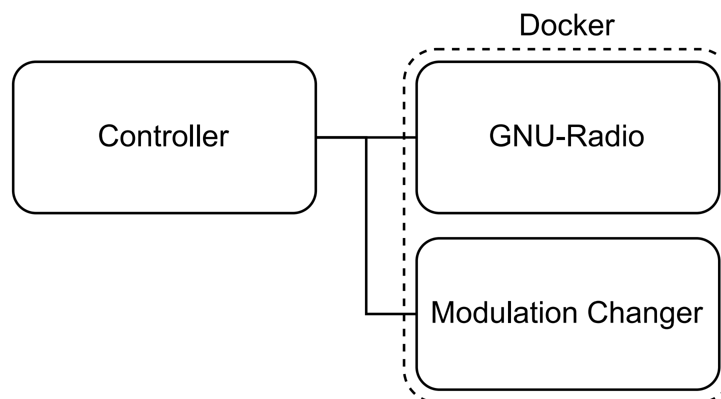

Figure 4.5: Ground Station Components

Due to the heavy dependency list of the GNU-Radio script, it runs inside a docker[3] container, easing the deployment process of the ground station. This module resorts to a SDR to communicate with the satellite via radio. The sending and receiving of messages to this script happens through a TCP connection. The controller exploits this connection to transmit and receive AX.25 frames to and from the spacecraft. In order to change the modulation in use, the running GNU-Radio script must be stopped and a new one started.

One solution to automate the script change would be to stop the docker and start a new one running a different script. Although simple, this solution proves to be inefficient due to the large load time of docker containers. Therefore, the solution adopted was to run a program inside the docker, modulation changer, that exposes a TCP socket allowing an external program to stop the currently running script and start a new one. As a bonus, if the running script crashes, the modulation changer is able to detect and

[3] https://www.docker.com/

restart it. The controller takes advantage of this script to allow the Core Server to change the modulation used by this ground station.

4.1.4 EGSE

The EGSE implements the ”gateway” and ”tiva” capabilities and is composed of a single software module, the controller. This module interacts with the Tiva-C and allows the debug of the spacecraft.

4.2 Graphical User Interface

Instead of implementing an in-house interface, the cloud based command and control solution, Major Tom, was integrated into the ground segment. This decision aimed to reduce the development and maintainability efforts while allowing the possibility of utilising their pre-integrated GSNs. As a bonus, Major Tom, provided an academic program, where it allowed university projects to take advantage of their service free-of-charge.

When integrating Major Tom into a new ground segment, the only required module is a gateway. Based on their notation, a gateway is a software component that receives messages in JSON and translates them into a format capable of being interpreted by the satellite. This gateway is responsible for establishing the connection with Major Tom, and the data it produces is forwarded either to Major Tom, in order to be sent using an integrated GSN, or to a specific gateway directly connected to the Core Server.

Figure 4.6 depicts how Major Tom was integrated into ISTSAT-1’s ground segment.

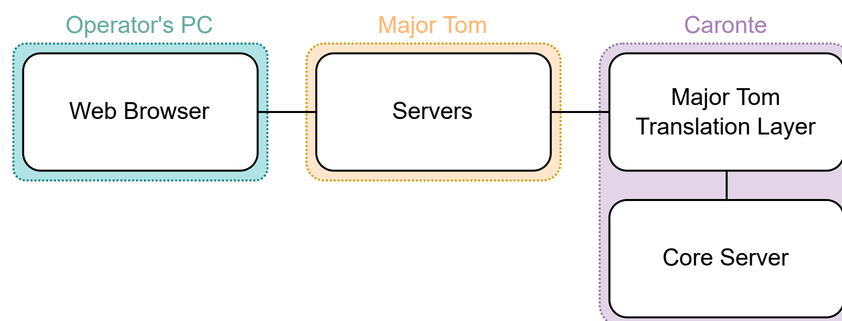

Figure 4.6: Major Tom

Instead of altering the Core Server to establish connections with Major Tom, a new software module, Major Tom Translation Layer was created. This module establishes a connection between Major Tom and the Core Server, and translates the communication between both. This decision allowed the integration of Major Tom, without requiring changes to the already stable Core Server, by leveraging Core Server’s client API (the same used by GSCTL).

The process of integrating Major Tom into the ISTSAT-1's ground segment can be decomposed into two parts, the command definition, and the development of Major Tom Translation Layer. These will be detailed after a brief overview of Major Tom's GUI.

4.2.1 Overview

Major Tom's interface is organized into multiple sections listed on the left side of the GUI. A screenshot of this UI is showcased in Figure 4.7.

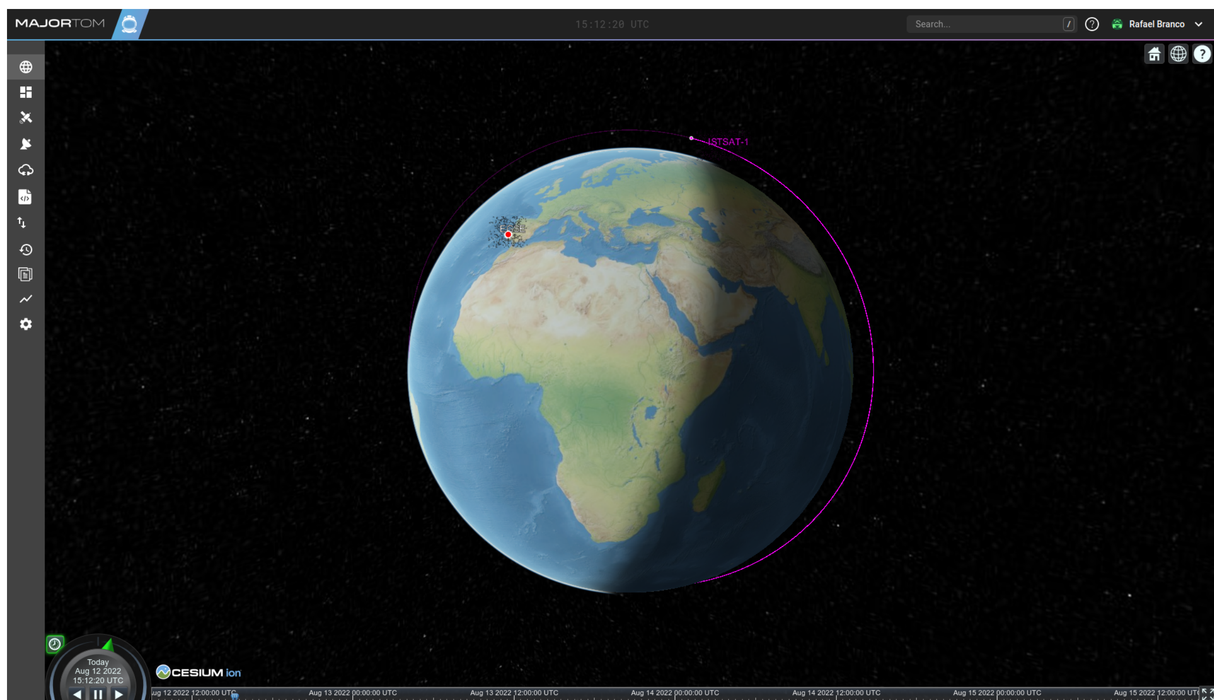

Figure 4.7: Major Tom Interface - 3D View

The different sections that divide the functionality of Major Tom are:

- 3D Viewer: allows the operator to simulate the position of the satellite at a given time (Figure 4.7).
- Satellites: where it is possible to register a new satellite, the commands it supports and its two-line element set (TLE).
- Ground Stations: where it is possible to register a new ground stations, the commands it supports as well as its location.
- Gateways: in this section it is possible to register new Major Tom gateways and to generate their authentication tokens.
- Scripts: enables the register of new scripts as well as the generation of their authentication tokens.

- Communications: lists the last issued commands, allows the repetition of said commands and provides a ”Rapid Commanding” functionality, where the user is provided with a CLI to issues commands to the satellite.
- Events: presents a list of events which are actions or occurrences recognized by Major Tom. Examples of events are the execution or completion of commands.
- Files: lists the files downlinked from the spacecraft, and the files containing data manually exported from Major Tom.
- Analysis: redirects to a Grafana UI where the operator is able to add new dashboards and panels to visualise the data gathered by Major Tom.
- Mission Dashboard: operator organized board containing widgets to ease the operation of the ground segment. The available widgets are:
  - Orbit simulator: displays ground station’s locations and the expected satellite position according to its TLE.
  - Live telemetry: a list containing the last values of tracked telemetry metrics.
  - Communications: provides the same functionality as the Communications section.
  - Actions: displays the satellites, ground stations, gateways, and scripts registered in Major Tom. It also allows the sending of commands to satellites, and ground stations.
  - Pass Timeline: visualisation for the next satellite passes, and allows the schedule of commands for each pass.

4.2.2 Command Definition

In order for Major Tom to provide a GUI capable catering to the different needs of multiple missions, it needs to be configurable. The process of configuring Major Tom’s UI is achieved through the upload of JSON objects to Major Tom’s website. These objects, refered to as command schema, define the commands that are available for a specific satellite or ground station.

4.2.2.A Format

Each command schema requires one attribute named ”definitions” whose value is another object that contains the command definitions. Each attribute key is a command name and its value the respective command definition. Command definitions are objects whose attributes are:

- ”display_name”: a string that will be used to display the command name in the GUI.

- ”description”: a string that will be used to display a description of the command in the GUI.
- ”tags”: an array of strings containing the categories it belongs to.
- ”fields”: an array containing field definitions.

The attributes ”display_name”, ”description” and ”fields” are required in every command definition. Field definitions are objects that represent the parameters of a command. Their attributes are:

- ”name”: the name of the field.
- ”type”: which is the type of the field. It can be ”integer”, ”float”, ”enum”, ”string”, ”text”, ”datetime” or ”boolean”.
- ”value”: a constant, unchangeable value.
- ”default”: a placeholder value.
- ”characterLimit”: the maximum allowed number of characters. Only valid for types ”string” and ”text”.
- ”range”: the input varies according to the type of the field. For ”integer” types an array with two values, while for ”string” types an array with all the possible values.
- ”enum”: an object whose attributes are the label of each ”enum” value and the value their value. Only valid for type ”enum”.

Every field definition requires a ”name” and ”type” attributes. If its type is ”enum” then the ”enum” attribute must also be specified.

Using the command definition rules defined, listing 4.8 shows a possible definition of a command to ping subsystems of ISTSAT-1.

```
1  {
2    "definitions": {
3      "ping": {
4        "display_name": "ping",
5        "description": "Ping satellite",
6        "fields": [
7          {
8            "name": "dst",
9            "type": "string",
10           "range": ["OBC", "EPS", "TTC", "COM", "PL"]
11         }
12       ]
13     },
14   }
15 }
```

Listing 4.8: Command Definition

4.2.2.B ISTSAT-1's commands

Communication with the satellite happens through the INCP protocol. The ISTSAT-1 is composed of subsystems and each subsystem has different data variables, reports, configurations, commands, and enumerations. These are described in a YAML Ain't Markup Language (YAML)[4] file that is processed to generate subsystem specific libraries, defining the available instructions.

Data variables represent the value of a metric at some point in time, an example would be the battery charge of the spacecraft. Reports are groups of data variables, they allow the request of multiple data variables in a single call. Configurations are parameters that determine the behaviour of the satellite, and can be altered in runtime. Commands are parametrisable procedures that execute on a subsystem.

The Core Server provides, through its client API, functions to interact with each spacecraft subsystem. These functions are "data_req", that requests the value of some data variable or report, "config_get" and "config_set", that, respectively, get and set the value of some configuration, and "cmd", which issues a command. These procedures are the basic building blocks that enable all satellite operations. Listing 4.9 provides definitions for these basic client API endpoints.

```
1  data_req(
2    subsystem: str,
3    variable_name: str,
4  ) -> List[str, Union[int, str]]
5
6  config_get(
7    dst: str,
8    name: str,
9    record: int,
10 ) -> Tuple[Union[int, str], str]
11
12 config_set(
13   dst: str,
14   name: str,
15   record: int,
16   value: str,
17 ) -> None
18
19 cmd(
20   dst: str,
21   name: str,
22   pargs: List[str],
23   kwargs: Dict[str, str],
24 ) -> Dict[str, Any]
```

Listing 4.9: Core Server Basic Functionality Client API

A sample call to Core Server's endpoint "data_req" to request the value of the data variable "battery_charge" that is present on the subsystem "EPS" is represented on Listing 4.10.

[4] https://yaml.org/

```
rpc_node.call("data_req", ["EPS", "battery_charge"], {})
# or, using the sugared syntax
# proxy.data_req("EPS", "battery_charge")
```

Listing 4.10: Data Request EPS' "battery_charge"

When translating the interface exposed by the Core Server to the Major Tom's command schema format, the first instinct is to directly map the Core Server's endpoints into command definitions. Listing 4.11 exemplifies how this would be done, using "data_req". This approach may mislead users since there is no way of specifying which variables belong to each subsystem. There is also no way of providing a description for each data variable.

```
{
  "definitions": {
    "data_req": {
      "display_name": "data-req",
      "description": "Request data from the satellite",
      "fields": [
        {
          "name": "subsystem",
          "type": "string",
          "range": ["OBC", "EPS", "TTC", "COM", "PL"]
        },
        {
          "name": "variable_name",
          "type": "string",
          "range": ["gyro_x", "battery_charge", ...]
        }
      ]
    },
  }
}
```

Listing 4.11: ISTSAT-1 Command Definition First Approach

One solution to the first issue, would be to not provide a "range" in the field "variable_name". Despite solving the problem, this approach would force the user to memorise all the possible variable names. Another solution would be to create a command definition for each subsystem and Core Server endpoint combination. However, this approach does not solve the problem of not being able to provide a description for each variable.

Since none of the afore mentioned alternatives provided a satisfactory solution, the chosen approach was to create a command definition for each specific data variable. Listing 4.12 illustrates the final, implemented format, resorting once again to the example of the data variable "battery_charge".

```
{
  "definitions": {
    "var_EPS_battery_charge": {
      "display_name": "battery_charge",
      "description": "Average of the last 10 readings ..."
      "fields": [
        {
          "name": "_command",
          "type": "string",
          "value": "data_req"
        },
        {
          "name": "dst",
          "type": "string",
          "value": "EPS"
        },
        {
          "name": "variable",
          "type": "string",
          "value": "battery_charge"
        },
      ]
    },
  }
}
```

Listing 4.12: ISTSAT-1 Command Definition Approach

In order to create the full command definition schema, a program was written to parse the INCP description file, and extract the required information. The command definition schema was then uploaded to Major Tom in order to customize its GUI to the mission requirements.

4.2.3 Major Tom Translation Layer

After successfully configuring Major Tom, a service to establish a connection between their servers and Core Server still needed to be developed. This service is the Major Tom Translation Layer. It establishes a WebSocket [10] connection with the Major Tom's servers, allowing them to send the requests issued by the client via their GUI. When it receives the issued command, translates it, and calls the appropriate function through Core Server's client API. As soon as response is received from the Core Server it is translated to Major Tom's format and sent. Listing 4.13 illustrates the message format received from Major Tom when a request for the data variable "battery_charge" is issued.

```
{
  "type": "command",
  "command": {
    "id": 123,
    "type": "var_EPS_battery_charge",
    "fields": [
      {
        "name": "_command",
        "value": "data_req"
      },
      {
        "name": "dst",
        "value": "EPS"
      },
      {
        "name": "variable",
        "value": "battery_charge"
      },
    ]
  }
}
```

Listing 4.13: Major Tom Request Format Example

Listing 4.14 depicts a possible response, in Major Tom's format, for the request in Listing 4.13.

```
{
  "type": "command_update",
  "command": {
    "id": 123,
    "state": "completed",
    "output": [
      ["battery_charge", 83],
    ]
  }
}
```

Listing 4.14: Major Tom Response Format Example

4.2.4 Telemetry

The described module enables the utilisation of Major Tom's UI to communicate with the spacecraft. However, the Grafana interface integrated into Major Tom is only being populated when responses to commands, issued from this client, are received. As such, telemetry data is also not being used to populate this Grafana's database.

In order to populate Major Tom's Grafana database with all the data received by the ground segment, instead of just the responses to commands issued from Major Tom, a new command was created in the Core Server, enabling a client to request to be notified as soon as new satellite data is received. From

this moment on, whenever the Core Server receives data variables or reports from the spacecraft, it forwards it to all the interested clients. To implement this behaviour, it calls the function ”on_data” on the clients that wish to be notified.

The Major Tom Translation Layer was then changed to request the Core Server to be notified when data is received from the spacecraft. When this data is received, it is translated into the Major Tom format and sent to their servers to be stored in their database. The format used to send data variables to Major Tom is represented in Listing 4.15, through an example of a ”battery_charge” update.

```
1  {
2    "type": "measurements",
3    "measurements": [
4      {
5        "system": "ISTSAT-1",
6        "subsystem": "EPS",
7        "metric": "battery_charge",
8        "value": 83,
9        "timestamp": 1666990090918,
10     }
11   ]
12 }
```

Listing 4.15: Major Tom Measurement Format Example

4.2.5 Impediment

Despite the working condition of the implemented GUI, due to contractual changes, Major Tom stopped providing their software free-of-charge for university projects. As such, this command and control service is no longer a viable solution for the ISTSAT-1’s ground segment.

4.3 Alternative Graphical User Interface

Since the previously integrated UI stopped being viable, a new GUI was developed, in-house. The decision to develop a new interface, instead of integrating another one into the ground segment, was motivated from the experience with Major Tom. Integrating a general command and control mission software into the ground segment does not provide the ideal flexibility nor customization levels, and entails the loss of control over the decisions made by the software provider. By developing the interface internally we guarantee complete control over these factors.

The developed interface was a web server, allowing clients to operate the satellite from anywhere, without the need of installing software on their machines. The architectural design followed is represented in Figure 3.14. The web server runs in the same machine as the Core Server and the operator accesses the interface through its browser.

In order to ease the development process, this module was programmed in Python using the pglet[5] framework. This framework allows the creation of GUIs through the use of pre-built components.

This UI focuses on providing a simple interface to ping, request data, edit configuration values, and issue commands to the satellite. Throughout this document, this interface will be referred to as ”Smooth Operator”.

4.3.1 Configuration

Similar to Major Tom, this interface is configurable through a JSON file containing the information about the different subsystem data variables, reports, configurations and commands. Currently this file is loaded on start-up and there is no update mechanism; however a feature that allows the operator to update the configuration file, by requesting it to the Core Server, is intended to be implemented.

The configuration file is composed of key-value pairs, where the key is the name of the subsystem and the value, an object containing information about it. This information is, itself, comprised of key-value pairs where the key is either ”data”, ”configs”, or ”commands”, and the values are lists of objects. The ”data”, ”configs”, and ”commands” objects contain the keys ”name” and ”doc” that, respectively, accommodate the name of the feature, and its documentation. Listing 4.16 exemplifies the definition of the data variable ”battery_charge”.

```
1  {
2    "EPS": {
3      "data" : [
4        {
5          "name": "battery_charge",
6          "doc": "Average of the last 10 readings for ...",
7        },
8      ],
9    }
10 }
```

Listing 4.16: Smooth Opertor’s ”data” Definition Example

The ”configs” objects also contain the key ”type” that defines the type of the value of this configuration, or ”options” which is an array containing possible values for the this configuration. To illustrate the possible format of a ”configs” configuration, Listing 4.17 depicts the definition of ”heater_enable”. If, like in this example, both ”options” and ”type” are provided, the latter will be ignored.

[5] https://pglet.io/

```
{
  "EPS": {
    "configs": [
      {
        "name": "heater_enable",
        "doc": "Enable heater functionality which uses ...",
        "options": ["false", "true"],
        "type": "integer",
      },
    ],
  }
}
```

Listing 4.17: Smooth Opertor's "config" Definition Example

The "commands" objects, contain a key "arguments" that is a list of "argument" objects. The "argument" objects contain the same keys as the "config" objects.

```
{
  "EPS": {
    "commands": [
      {
        "name": "heater_enable",
        "doc": "Enable heater functionality which uses ...",
        "arguments": [
          {
            "name": "heater_enable",
            "doc": "Enable heater functionality which ...",
            "options": ["false", "true"],
            "type": "integer",
          },
        ],
      },
    ],
  }
}
```

Listing 4.18: Smooth Opertor's "command" Definition Example

This configuration file can be automatically generated using a script that parsed the ISTSAT-1's configuration file.

4.3.2 Interface

The GUI is subdivided into two parts, the navigation bar, on the left side, and the operational interface on the right. Figure 4.8 displays the interface to request data variables and reports from the satellite.

Whenever possible, the interface aims to reduce user mistakes by replacing standard input boxes by drop-down menus. While in the "Data Request" tab, the only documentation available is related to the

selected variable, in ”Commands”, some arguments also contain a description to aid the operator. In these cases, this information is also displayed.

To further reduce operator errors, every time a field is altered, all the fields whose validity depends on such field are checked, and reset if invalid. This means that, if in Figure 4.8, the operator was to change the subsystem selected from ”EPS” to ”OBC”, the variable value would be erased, since the ”OBC” does not contain the variable ”battery_charge”.

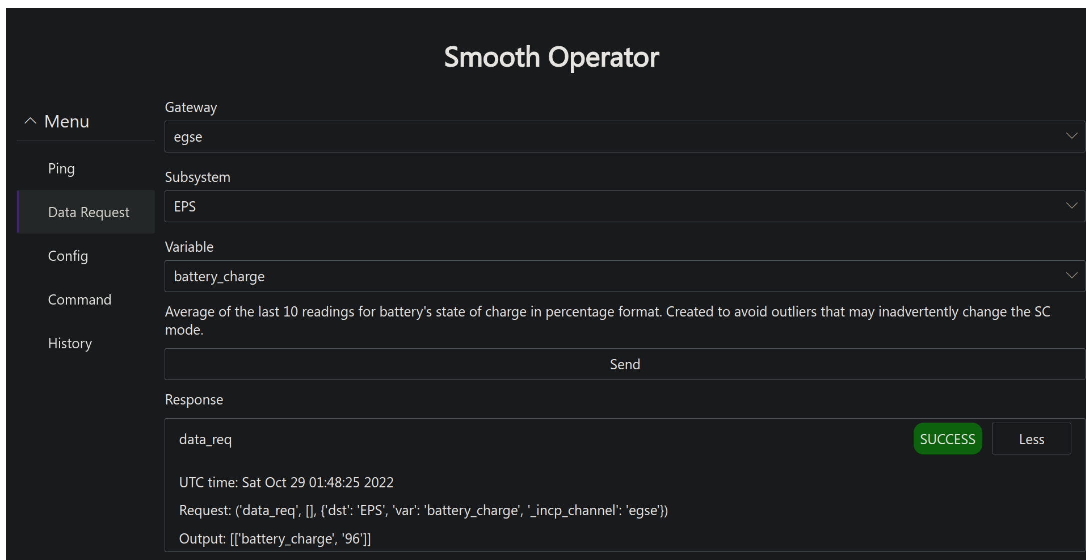

Figure 4.8: Smooth Operator’s Data Request Interface

Each page contains a response section at the bottom of the operational interface where the last issued request is displayed. The ”History” tab contains a list with information about all the requests issued in this session.

4.3.3 Security

Pglet provides built-in support for authentication through the sign in with a GitHub, Google, or Microsoft account. This functionality was used to block unwanted access to the application.

However traffic between the client’s browser and the web server still happens over an unencrypted connection. During the deployment phase, this needs to be taken this into account, proxying the data through a TLS termination proxy.

4.4 Operation Queue

The Operation Queue was developed to enable the scheduling of operations. When this module boots, it automatically establishes a connection with the Core Server’s client API. After the connection is

established, it notifies the Core Server that this client is an Operation Queue.

When another client wants to schedule an operation, it issues a remote call to the Core Server providing the operation name, and the date and time for when this operation needs to take place. The Core Server then proxies this message to the connected Operation Queue that registers it in its queue of operations and waits for the next timestamp. Figure 4.9 represents a sequence diagram showing both the start-up sequence and the schedule of an operation using GSCTL.

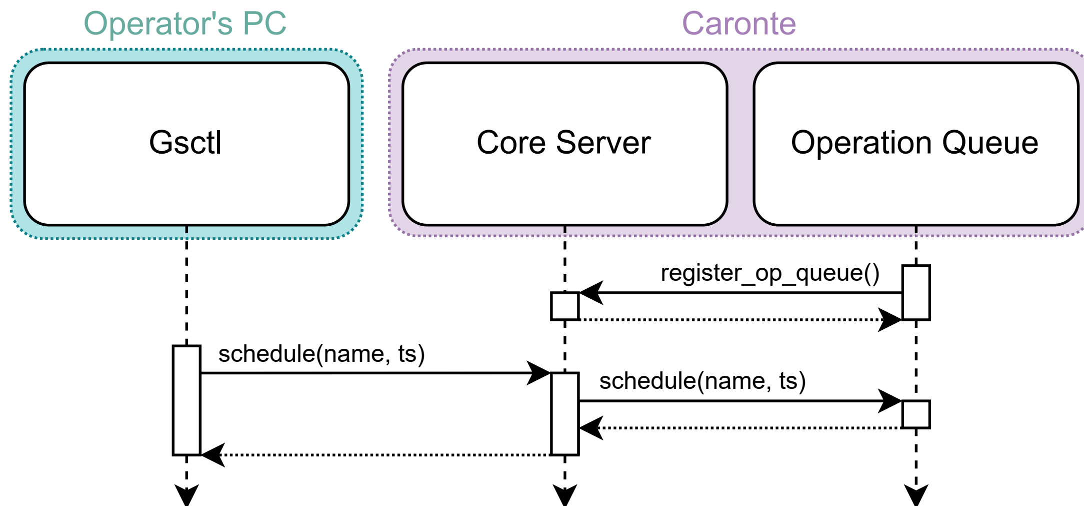

Figure 4.9: Operation Queue Sequence Diagram

When it is time to execute the operation, the Operation Queue, runs it.

5

Validation

Contents

5.1 Mission Test 59
5.2 Vibration Test Campaign 60
5.3 Space Studies Program (SSP) Class 61
5.4 Results 61

ISTSAT-1 is part of European Space Agency (ESA)'s Fly Your Satellite! (FYS) [11] programme. As such, in order to launch the spacecraft, it has to endure a panoply of tests. Some of these tests, while assessing the functionality of the satellite, also test the ground segment, as they require a close interaction between both parties.

At the end of the first development phase, both the satellite and the ground segment, enrolled on a testing campaign (Mission Test (MT)), aimed at verifying that the ISTSAT-1 is able to perform its intended mission and withstand contingency scenarios. Since the satellite needs the ground segment to be able to complete its mission, by extension, it was also tested.

After the MT, the satellite enrolled on another test campaign, the Vibration Test (VT), that intended to verify that the spacecraft can withstand the random vibrations expected for the launch phase. Despite the ground segment not being the main target of this test campaign, it can once again be used to validate its behaviour.

Following the VT, students from the International Space University's SSP[1] visited the infrastructure of the ISTSAT-1's laboratory for a class. During this class, a simulation of a real satellite operation took place, corroborating once again the working condition of the implemented system.

5.1 Mission Test

In order to verify that the satellite is able to fulfil its mission, this test simulated the interactions with spacecraft during its first orbits.

The MT was subdivided into four test phases, launch preparation, in-orbit commissioning, nominal operations, and contingency operations. The launch preparation phase detailed the necessary steps to properly setup both the spacecraft, and the ground segment. During the in-orbit commissioning, the satellite antennas were deployed, and simple radio communications performed. The nominal operations phase, tested the functionality related to the satellite's mission. While the contingency operations, verified that all the systems put in place to recover from a malfunction were properly working.

Both the nominal and contingency operation phases were operated remotely, due to COVID-19 restrictions.

This test campaign took place in Instituto de Engenharia de Sistemas e Computadores - Microsistemas e Nanotecnologias (INESC-MN)'s clean room. To perform this test, all the required equipment to operate the satellite had to be transported to this location. The necessary equipment was the EGSE and the SDR ground station. Figure 5.1 depicts the architecture used in order to execute the MT.

[1] https://ssp22.isunet.edu/

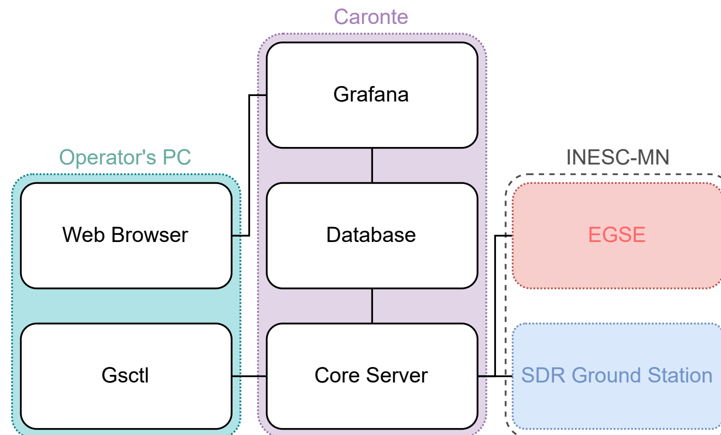

Figure 5.1: Mission Test Setup

5.2 Vibration Test Campaign

During this test, the spacecraft was supposed to be vibrated on its three axis, assuring that it will not be damaged during the launch. Before and after vibrating the satellite, it had to be subject to a Reduced Functional Test (RFT), to ensure that the vibration did not harm the spacecraft.

This test campaign took place in the CubeSat Support Facility (CSF), an assembly integration and testing facility for CubeSats, located at the ESA Education Training Centre, based at the ESEC-Galaxia facility in Transinne, Belgium. As with the MT, the EGSE and the SDR ground station were transported to the site, because they are required to execute the RFT.

The ground segment architecture followed was the same as the one used in the MT. Figure 5.2 is a photograph taken inside the CSF's clean room, during the execution of the first RFT. In the bottom left of the picture, is the spacecraft with connected to the EGSE.

5.2.1 Challenges

After configuring both gateways to connect to the local Wi-Fi network, none was able to establish a connection with the Core Server. Due to the lack of time, there was no chance to properly investigate the root cause of the issue. However, it is possible that the router was configured to block outbound traffic to the port where the Core Server is hosted.

This issue was solved through the use of a mobile hotspot. Another possible solution, would be to setup a SEGSE, avoiding the need for an internet connection.

Figure 5.2: Vibration Test Campaign RFT

5.3 SSP Class

The objective of this class was to introduce the students to the basic concepts of satellite design and operation. To this end, the ground and space segments of ISTSAT-1 were used.

After familiarising the students with the mission of ISTSAT-1, the class culminated with a live operation of a simulated pass, in which the ground operator would issue commands to the spacecraft showcasing some of its functionality. Figure 5.3 is a screenshot of the CLI command used to request the report "flight_pass" from the spacecraft's subsystem "EPS".

```
(istnanosat-T9WOlWn3-py3.8) Software on  development core » gsctl data-req EPS flight_pass
time_boot                      : 915215
hk_scmode                      : SCMode.SAFE
battery_charge                 : 95
battery_voltage                : 7956
battery_current                : -6
battery_temperature1           : 570
battery_temperature2           : -114
battery_temperature3           : 525
battery_temperature4           : 5241
eps_temperature                : 2718
```

Figure 5.3: GSCTL Data Request

5.4 Results

During both test campaigns and the demonstration, the ground segment's functionality was extensively tested.

The MT verified that both gateways were able to connect to the Core Server without any kind of network configuration. As soon as both components had internet access, provided by INESC-MN's Wi-Fi, they established a connection with the Core Server, enabling any client to use them to communicate with the spacecraft. This vast test also ensured that none of the previously implemented features were broken.

The VT, validated once again the working condition of the ground segment. Despite the challenges described, the architecture of the ground segment proved to be robust enough to surpass them. Nonetheless, alternative solutions to the found challenges should be investigated.

The SSP class, while also a success in terms of ground segment functionality, could have been improved had the new GUI, already been implemented. However, since these tests took place before the end of the second development phase, this could not be achieved. Nevertheless, Figure 5.4 exposes how the same command, displayed in the Figure 5.3, would have been issued through Smooth Operator.

Smooth Operator

Menu

Ping

Data Request

Config

Command

History

Gateway

egse

Subsystem

OBC

Variable

flight_pass

Requests the values of: time_boot, hk_scmode, hk_mon_ttc_status, hk_mon_eps_status, hk_mon_com_status, hk_mon_pl_status, adcs_state, adcs_ang_vel_magn, adcs_sat_updates_left, adcs_sat_updates_time_left, adcs_sun_updates_left, adcs_sun_updates_time_left

Send

Response

data_req SUCCESS Less

UTC time: Mon Oct 31 21:27:37 2022

Request: ('data_req', [], {'dst': 'OBC', 'var': 'flight_pass', '_incp_channel': 'egse'})

Output: [['time_boot', '940704'], ['hk_scmode', 'SCMode.SAFE'], ['hk_mon_ttc_status', 'SStatus.ON'], ['hk_mon_eps_status', 'SStatus.ON'], ['hk_mon_com_status', 'SStatus.OFF'], ['hk_mon_pl_status', 'SStatus.OFF'], ['adcs_state', 'ADCSState.IDLE'], ['adcs_ang_vel_magn', '25973628'], ['adcs_sat_updates_left', '0'], ['adcs_sat_updates_time_left', '0'], ['adcs_sun_updates_left', '0'], ['adcs_sun_updates_time_left', '0']]

Figure 5.4: Smooth Operator Data Request "flight_pass"

6

Conclusion

Contents

6.1 Future Work 67

Communication is crucial to a satellite's mission success. Therefore it is important to employ techniques to increase the number of available passes as well as their throughput. This problem needs to be addressed by most LEO satellites, and the ISTSAT-1 is not an exception.

The main goals of this work stemmed from the analysis of the initial ISTSAT-1's ground segment. The assessment revealed issues regarding latency and goodput, the need for a GUI, and the ability to schedule operations.

Throughout this work, changes to the ISTSAT-1's architecture were proposed and implemented, aiming to improve the goodput of the satellite, as well as the usability of the UI leveraged to operate the ground segment and, by extension, the spacecraft.

To this end, the connection mechanisms and interfaces between gateways and the centralised server where standardised with the purpose of reducing the network load, increasing maintainability, and providing a clear commanding interface. The design decisions made to accomplish these goals, motivated the extension of an in-house RPC library, to allow servers to issue remote calls on clients, which greatly simplified the implementation process. These changes were validated during two test campaigns.

The work on the UI for the ground segment resulted in the integration of the cloud based, command and control solution Major Tom. While this service proved to be an adequate solution to interface with the functionality provided by the ground segment, due to contractual problems, the partnership with this company did not proceed; motivating the development of a new GUI. This UI provides a simpler interface, more tightly coupled with the needs of this specific ground segment.

On the automation front, an approach to the scheduling of operations was discussed and implemented. Enabling users to calendar the execution of operations directly through the centralised server.

6.1 Future Work

In order to improve upon this work, the GUI developed should be subject to user trials, evaluating its performance when compared to the CLI, both for new and proficient users.

Another way of expanding on this work would be to allow interested people to contribute to the telemetry data collection process. This can either be achieved by integrating the ground segment with a GSN, or by openly distributing the software for the ground stations. If the latter option is chosen, any person could use their ground station to collect telemetry, and automatically send it to the Core Server. In order to incentivise contributions, the satellite data could be displayed in a public manner, with a leaderboard listing the most active contributors.

Bibliography

[1] B. Elbert, *The Satellite Communication Ground Segment and Earth Station Handbook, Second Edition*, ser. Artech House space technology and applications library. Artech House, 2014. [Online]. Available: https://books.google.pt/books?id=eepZBAAAQBAJ

[2] G. D. Considine and P. H. Kulik, *Van Nostrand's scientific encyclopedia*, 10th ed. Hoboken, N. J: Wiley-Interscience, 2008.

[3] *Instrument procedures handbook*. Aviation Supplies and Academics, Inc., 2017.

[4] T. Pirrone, "Integrating a Global TT&C Network," in *SpaceOps 2012 Conference*. Stockholm, Sweden: American Institute of Aeronautics and Astronautics, Jun. 2012. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-1275262

[5] J. Nicolas, "SatNOGS: Towards a Modern, Crowd Sourced and Open Network of Ground Stations," *Proceedings of the GNU Radio Conference*, vol. 2, no. 1, Jan. 2021, number: 1. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/view/100

[6] J. Wertz and W. Larson, *Space Mission Analysis and Design*, ser. Space Technology Library. Springer Netherlands, 1999. [Online]. Available: https://books.google.pt/books?id=veyGEAKFbiYC

[7] D. Velten, R. Hinden, and J. Sax, "Reliable data protocol," RFC-908, BBN Communications Corporation, Tech. Rep., 1984.

[8] M. Pessans-Goyheneix, J. Bønding, M. B. Tychsen, and K. F. Jensen, "AAUSAT3," publisher: Citeseer.

[9] W. A. Beech, D. E. Nielsen, and J. Taylor, "AX.25 Link Access Protocol for Amateur Packet Radio," p. 143.

[10] A. Melnikov and I. Fette, "The WebSocket Protocol," RFC 6455, Dec. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6455

[11] J. Vanreusel, “Fly Your Satellite! The ESA Academy CubeSats programme,” in *Proceedings of the ITU Symposium & Workshop on Small Satellite Regulation and Communication Systems, Santiago de Chile, Chile*, 2016, pp. 7–9.