TÉCNICO LISBOA

Development of the ISTnanosat-1 Ground Segment

Alexandre Simões Silva

Thesis to obtain the Master of Science Degree in

Bologna Master Degree in Telecommunications and Informatics Engineering

Supervisor: Prof. Rui Manuel Rodrigues Rocha

Examination Committee

Chairperson: Prof. Paulo Jorge Pires Ferreira
Supervisor: Prof. Rui Manuel Rodrigues Rocha
Member of the Committee: Prof. Fernando Henrique Côrte-Real Mira da Silva

May, 2017

Resumo

Neste relatório, é proposto uma solução para o segmento terrestre do ISTNanosat-1.

O ISTNanosat-1 é um CubeSat, um tipo de satélite artificial, actualmente em desenvolvimento por alunos e professores do IST juntamente com especialistas rádio amadores da AMRAD e AMSAT-CT. Tal como qualquer satélite, este precisa de ser monitorizado, controlado e os resultados obtidos em experiências científicas tem que ser recuperados. O segmento terrestre é então a infra-estrutura necessária na Terra que permitirá essas tarefas.

O segmento de solo ISTNanosat-1 é um sistema que permite a recuperação de telemetria e emitir instruções ao satélite. Toda a informação recolhida pode ser visualizada pelos utilizadores através de uma aplicação web. Os utilizadores devidamente autorizados têm ainda a capacidade de emitir as instruções para que o satélite as execute. O segmento terrestre desenvolvido é composto por várias estações terrestres controladas por um servidor principal (denominado por "Core server"). Nesta arquitectura centralizada, o servidor central comunica com o satélite através das estações terrestres utilizando o protocolo desenvolvido para o efeito (o chamado ISTnanosat Control Protocol (INCP)). Há ainda vários serviços que permitem a recolha de dados e erros, execução de comandos e diagnósticos, para além de assegurar a segurança da ligação entre os segmentos terra-espaço.

Palavras-chave: satélite, CubeSat, ISTNanosat, segmento terrestre, estação terrestre.

Abstract

In this report we propose a solution for the ground segment of the ISTNanosat-1.

The ISTNanosat-1 is a CubeSat, a type of artificial satellite, currently under development by students and faculty from IST along with amateur radio experts from AMRAD and AMSAT-CT. As with any spacecraft, it will need to be monitored, controlled and results from scientific experiments have to be retrieved. The ground segment is the necessary infrastructure on Earth that will permit these tasks.

The ISTNanosat-1 Ground Segment presents a system capable of telemetry retrieval and issuing instructions to the satellite. All of the relevant information is then shown to the users using a web application. Furthermore, properly authorized users have the capability to issue the aforementioned instructions. The developed ground segment is composed by multiple ground stations controlled by Core server. In this centralized design, the Core communicates with the satellite trough the ground stations, which serve as gateways, using the designed and implemented ISTnanosat Control Protocol (INCP). In the Core, several services permit for the for data and error retrieval, command and diagnostics execution and security of the ground-space segment link.

Keywords: satellite, CubeSat, ISTNanosat, ground segment, ground station.

Contents

List of Tables

List of Figures

Acronyms

CSP Cubesat Space Protocol.

DGS Designated Ground Station.

GENSO Global Educational Network for Satellite Operations.

GS Ground Station.

GSC Ground Station Control.

INCP ISTnanosat Control Protocol.

LEO Low Earth Orbit.

LOS Line of Sight.

MCC Mission-Control Crew.

PAL Protocol Abstraction Layer.

PMA Persistent Message Authentication.

SC Session Counter.

SN Sequence Number.

SS Secure Session.

TLS Transport Layer Security.

TS Timestamp.

UI User Interface.

Chapter 1

Introduction

Ever since the launch of the first man-made satellite, the Soviet Union's Sputnik1 on 4 October 1957, marking the start of the Space Race, satellites have been instrumental in Human achievements. These are used in scientific, commercial, military and communications applications. The Hubble satellite and the Global Positioning System satellites being important examples. However, it has always been a considerable challenge to successfully develop and launch a satellite. The harsh conditions of space and the difficulty of reaching it, resulted in high costs in terms of money, development time and expertise. Even after a satellite has been successfully launched, a certain element of risk is always present, since a problem or error that reveals itself at this stage is hard to correct, if at all possible. It is because of these factors that only projects which were expected to have a high return on investment, either in monetary or scientific value, were viable. It was this high entry barrier that spurred the creation of the CubeSat.

A CubeSat is a low-cost, small scale satellite, built using a standardized format originally proposed by professors Jordi Puig-Suari of California Polytechnic State University and Bob Twiggs of Stanford University in 1999[1]. Each CubeSat is composed by multiple $10 \times 10 \times 10cm$ modules with up to 1.33kg of mass. Typical designs range between 1U and 3U, corresponding to one and three modules of $10cm^3$, respectively.

Due to their size, CubeSats have certain limitations, these are designed to be deployed in Low Earth Orbit (LEO), 160Km to 2000Km of altitude, since higher orbits would pose enormous challenges in terms of protection against cosmic radiation and communications. Nevertheless, these satellites are extremely useful in cases where the intended use does not justify a fully fledged satellite.

Since the initial launch of the two first CubeSats in 2003, the number of projects involving this type of satellites has exponentially increased, with 102 CubeSats already launched in 2017 and another 463 scheduled to be launched this same year [2]. The key to this success is their (relative) low cost and development time. Their small dimensions allow CubeSats to be deployed using the excess space of larger launch vehicles, this is commonly referred to as *piggy-backing*. In addition, the standard format greatly reduces the cost to adapt the satellites to such spaces, since the same mechanisms can be used for multiple CubeSats.

The ISTNanosat-1 is a CubeSat, which is being developed by students and faculty from IST/ULisbon

of various different engineering programmes, along with amateur radio experts from AMRAD (AMSAT-CT). The nano-satellite itself is composed by different subsystems that fit together inside the cubic case. These modules can be divided in two groups, the so called flight subsystems and the payload. The former are subsystems responsible for the control, power and communication functions of the satellite, while the latter are responsible for the satellite's mission. Additionally, this first iteration of the ISTNanosat project will serve to gain valuable experience for future more ambitious missions and CubeSats.

The mission involves the Automatic Dependent Surveillance – Broadcast (ADS–B), which is a surveillance system for tracking aircraft using periodically broadcast beacons containing their positions [3]. While today this system is optional, it will become mandatory in 2020. Currently, only ground stations track aircraft, and therefore remote areas such oceanic routes remain unmonitored. The tracking of aircraft from space became particularly interesting, as airline industry safety in such regions became the topic of debate after the incident regarding flight 370 from Malaysia Airlines.

The ISTNanosat-1 will thus build upon previous spacecraft missions regarding ADS-B technology [4, 5, 6, 7, 8, 9] and test a wide Field-of-View (FOV), small form factor (patch) ADS-B antenna, installed on the nadir pointing face of the satellite. Once operational, the ISTNanosat-1 will collect aircraft tracking data (specially from remote areas), process and store the information, and finnaly transfer it to the ground whenever possible. This data will then be further analyzed in order to characterize the "Cone of Silence" when a aircraft is at the nadir of satellite and several performance metrics including the probabilities of target acquisition, detection and identification [10].

An artificial satellite system has three major operational components, the Space Segment and the Ground Segment. Together, the ground and space segments are what enables a satellite to be communicable from Earth, which then enables its control, data collection and overall usage by the operations to accomplish its missions, see figure 1.1.

The Space Segment comprises the satellite constellation or, like in the ISTNanosat-1's case, a singular satellite and the uplink and downlink satellite links. The satellite itself is composed of several subsystems. The Command and Data Handling Unit (C&DH) is the module in the satellite responsible for monitoring and coordinating all other subsystems. The Electrical Power Subsystem (EPS) handles the batteries, solar panels and its function is to provide power to the satellite and its subsystems. The Attitude Determination and Control System is responsible for determining the satellite's position and to make orientation corrections if needed. Finally the Communication Module (COM), modem, transceiver and RF front-end assure the satellite's communications.

The ground segment is the necessary infrastructure on Earth that supports the communications, monitorization and control of the satellite. Ground-Space communications are support by what is called Ground Stations (GSs), a computer connected to a radio capable of communicating with the satellite using a predefined set of protocols. Using one or more of these GSs, the Ground segment can then receive telemetry, data produced by the satellite's subsystems, and issue of commands as to instruct the satellite to perform various tasks. Once, the Ground Segment can monitor and command the satellite it must provide a interface such that the operators can view it and issue commands. Operations invariably include the Mission-Control Crew (MCC), which will take care of the housekeeping related tasks, and

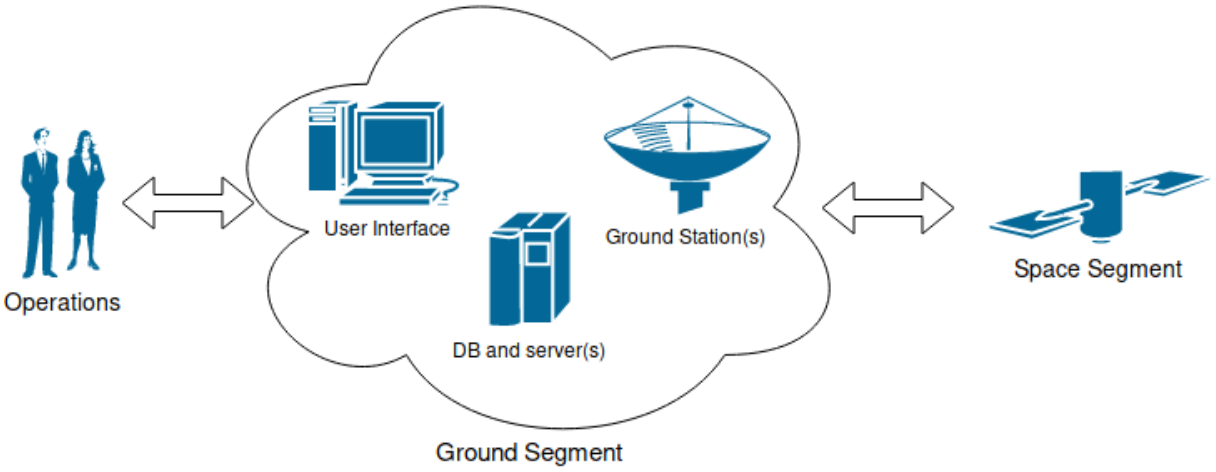


Figure 1.1: Ground and Space segments.

those interested in the missions. Note however, that this two groups are not necessarily composed by different individuals.

1.1 Motivation

As with other satellites, the ISTnanosat-1 will need constant monitoring and adjustments to keep it in good working conditions. Not only to command the satellite flight functions but also to retrieve important data concerning the scientific experiments and applications performed on-board.

Although other ground segment solutions already exist, due to the specific nature of our mission and the emphases on in-house developed subsystems, none would fit our needs. In particular, wich telemetry the satellite produces, how it can be retrieved and how to present it. Therefore, a in house solution must be designed and developed which also respects the educational aspects of the ISTNanosat-1.

First and foremost, the Ground Segment must assure the communications with the ISTNanosat-1 as to support telemetry retrieval and instruction transmission. Due to the characteristics of a LEO the Ground Segment must be able to support and use multiple distinct ground stations. This way, the ground and space segments will have improved comunication capabilites which better handle the limited bandwith and Line of Sight (LOS) windows of any GS.

1.2 Objectives and Challenges

The first and main objective, is to create and provide a system that allows to communicate with the satellite. This system will then be used by the MCC to retrieve data and issue commands. It will also serve as a base for performing the housekeeping of the satellite and to retrieve scientific data.

In order for the MCC to perform its duties, they must be able to issue commands that change the state of the satellite. These commands, which are transmitted over radio in insecure links, can be used

with malicious intentions, rendering the satellite inoperable. Therefore, this system must be designed in such a way that only the actual MCC's commands are executed.

Once communications are assured, a user interface must be produced, which will be used to interact with the satellite. This interface must enable two distinct uses. First, it must enable the ISTNanosat-1 project members to view all collect telemetry and to issue commands to the satellite. In other words, allow for the overall management of the satellite and it's flight subsystems. Secondly, due to the academic nature of this project, all of the data should be publicly available to anyone who wishes to see it. This is the intended use for anyone who is not part of the MCC but is interested in the data produced as a result of the project's missions. Due to the two different uses for this interface, we must strike a balance between creating an interface that is as functionally rich as possible for the MCC and user friendly for everyone else.

Since problems can occur at any time, an automated way of verifying the correct functioning of the satellite is required. Therefore, one must be able to review, schedule and execute automatic diagnostics. This will work by having a mechanism in place that first reviews the data received from the satellite. Then, if certain conditions verify on the analyzed data, instructs the CubeSat to perform diagnostics.

1.3 Document Outline

This report is divided in six chapters. In chapter 2, we present the related work concerning the ISTNanosat-1's ground segment. The addressed concepts and designs will serve as the base on which our solution will be built. In chapter 3, we present this solution and in chapter 4, its implementation. In chapter 5, we detail how testing and evaluating of the proposed solution was performed. Finally, we finish with chapter 6, were we give an overview of the work performed and addressed in this report.

Chapter 2

Related Work

In this chapter we address three topics which directly affected how the ISTNanosat-1's Ground Segment was designed. Firstly, in section 2.1, we detail the types of ground segments' architectures which other CubeSat projects used. Secondly, in section 2.2, we briefly detail some types of user interfaces employed in this kind of project. Thirdly, in section 2.3, we briefly describe the communication protocols used in ground-space segment communications and the satellite's architecture to the extent relevant to the communications.

2.1 Ground Segment Architectures

While there many different CubeSat satellite projects, each with its own set of objectives to accomplish, there are three main architectures when it comes to the ground segment. The designs are as follows:

One Ground Station this design uses only a single GS to establish communication with the satellite variations.

Single Satellite Ground Station Network in this type of architecture a single satellite's ground segment uses multiple ground stations.

Multiple Satellites Ground Station Network this architecture builds upon the previous one, to provide the same network of GSs for multiple different satellites.

2.1.1 One Ground Station

Since only one GS is used, communication with a satellite in LEO is restricted to time windows of approximately 15 to 20 minutes in ideal conditions. This is because a GS can only establish a link with the satellite if they are in LOS of each other. This leads to several problems both for the maintenance of the satellite and the execution of its missions.

Due to the low transmission speeds, if a satellite produces more data than the amount that it can transmit per connection window, some data will be lost. Error handling is also affected, as errors can only be reported when there is connection to the satellite.

2.1.2 Single Satellite Ground Station Network

Since using only one ground station can greatly limit communication with the satellite, a system who uses multiple GSs is the next logical step. The delfi-C3 [12], is a CubeSat which posses a ground segment representative of this architecture. It's this project that will be discussed in order to illustrate this type of architecture.

The ground segment of the Delfi-C3 is composed by the Delft Ground Station at Delft University of Technology, the Eindhoven ground station located at Eindhoven University of Technology, a worldwide network of radio amateurs and a central server to connect them. The hierarchy of functionalities of this architecture can be seen in Figure 2.1. The Delft GS serves as the main point for communication, both for telemetry and commands. The Eindhoven GS serves as a backup in the eventuality that the Delft GS can not issue commands to the CubeSat. Normally it's used for the reception of telemetry. Furthermore, additional telemetry is received trough the radio amateur network to supplement these two main ground stations. All telemetry is sent, trough the internet, to a central server in Delft were it is stored.

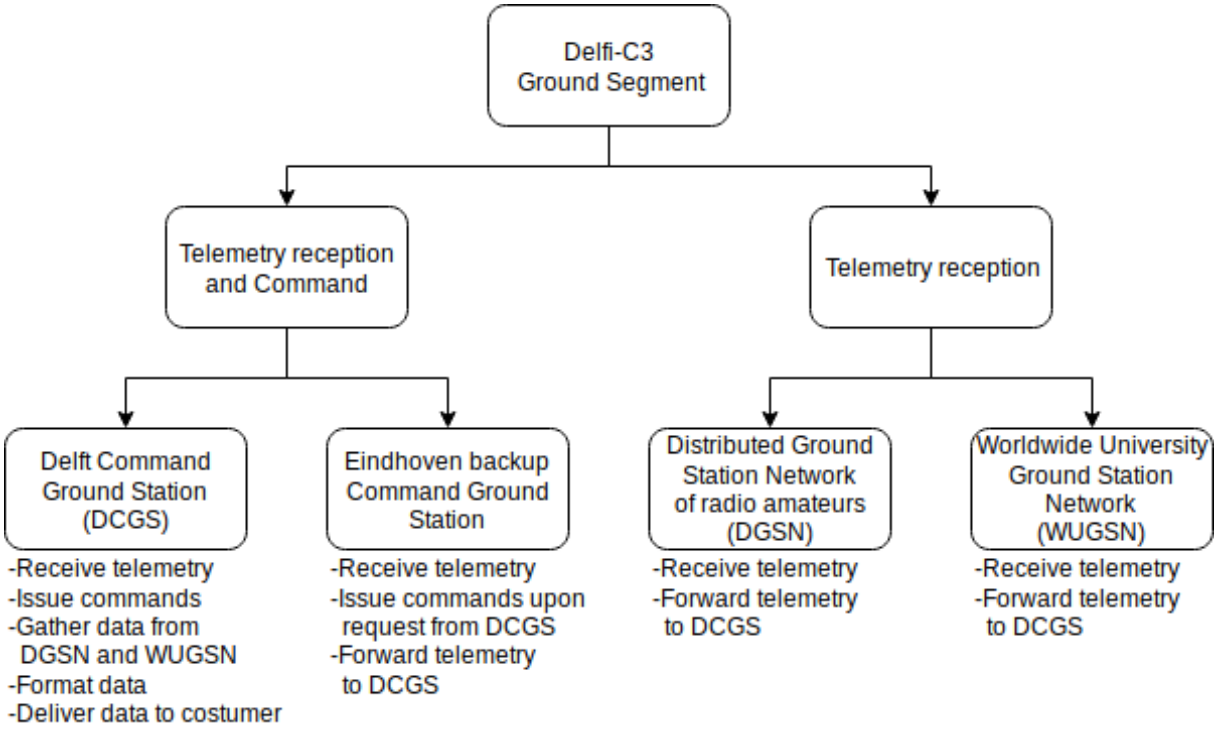


Figure 2.1: Delfi-C3 ground segment entities. (recreated from [12])

Once communication with the delfi-C3 CubeSat is established by any ground station or radio amateur, data is extracted from received frames, fragmented and sent to the central server, which is supposed to collect, store and process this data in order to properly provide the necessary information when requested.

The main part of the ground segment is the Delft Command Ground Station (DCGS). It's used to transmit commands, to receive payload and other housekeeping data. For security reasons up-links are possible only from Delft GS, but in the eventuality of a failure this function can be performed by the ground station in Eindhoven.

The distributed ground station network is formed by amateur satellite operators around the world. They used software produced by the Delfi-C3 team, and enables to decode, view and send all collect data to the central server. This way, telemetry collision in greatly increased, with the possibly that in the future, these amateur operators serving as a ground station, could also send commands from Delfi-C3 team to the satellite. To do this up-link however, some security scheme had to be implemented. The Delfi-C3 team, mentioned that the sent commands would have to be protected with some mechanism that assured the authenticity, integrity and freshness of the commands.

2.1.3 Multi Satellite Ground Station Network

During development, the Delfi-C3 team raised the question of why not use a well established network of ground stations? Since many universities have a CubeSat project of their own and therefore possess a working GS. Additionally, the CubeSat's low orbit translates into only a few passes per GS per day, if the GSs only task were to communicate with their University's CubeSat, their equipment would be severely underused. Therefore, if all of those GS were to be interconnected and made available for other CubeSat projects everyone would benefit. Specifically, it would enable more communications possibilities for each satellite and its MCC.

It was this type of questioning that sparked the creation of Mercury[13], GSN[14], MC3[15], GENSO and SatNet. These systems all perform the same task, they support communications with multiple different CubeSats using the same infrastructure, i.e. multiple GSs. All these systems have different implementations and designs. However, since they all perform the same tasks, they have many similarities. In addition, there are some aspects that are common to all of them, as they are fundamental for this type of system to function.

In order to enable multiple different satellites to use the same GSs, these must not need knowledge about how the data (received and sent) is encoded. These serve as gateways that merely allow accessibility trough them, between the MCC and its respective satellite. Although the GSs do not interpret the data that flows trough them, they must have some knowledge about the satellites whose communications they are supporting. This information must be previously configured using some service of the ground segment, were the satellite's orbit, radio frequency and other such details are specified.

Once a satellite establishes a connection with a GS of such a network, it must, trough some mechanism, know how to establish a second connection to the MCC using the Internet. Once established, the MCC can use this connection to communicate with the satellite using it's software solution.

The main benefit of this approach is the sharing of the same network of GSs for different projects and satellites. This way, new CubeSat projects can benefit from an already existing network of ground stations, which reduce costs in software development and most likely increases the number of ground stations available to any one project, when compared to the number that each project would achieve on its own. However, since communications between the MCC and the satellite are performed trough a GS serving as a gateway, there is an additional amount of delay as the MCC and the GS are geographically separated.

To further detail this architecture we will approach two projects as examples of this type of architecture, the GENSO and the SATNet projects. The first is a perfect example of this type of architecture even tough it has since became deprecated. The latter, is still under development, but due to its unique approach it can be useful to understand what these types of systems might become, or at least to explore different approaches to this type of system.

GENSO

The Global Educational Network for Satellite Operations (GENSO)[16] aimed to increase the overall communication windows with cubesats via multiple coordinated ground stations.

GENSO is composed by three elements, which can be see in figure 2.2, the Authentication Server (AUS), the Ground Station Server (GSS) and the Mission Control Client.

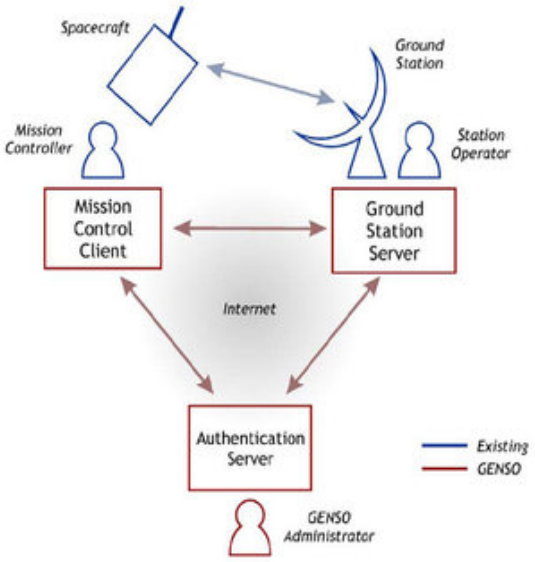


Figure 2.2: GENSO's communication framework.

The AUS is the central core of the network. These are a control tool used by the network's administrators. Multiple servers are used both to avoid a single point of failure and to synchronize them between themselves as to avoid an incoherent state.

Together, the authentication servers are known as the Authentication Servers List (ASL). The ASL serves as the project's infrastructure backbone and validates the identity of all clients using the network. GENSO must also keep lists containing the relevant information about all the MCCs, satellites and ground stations. Then, it must provide these lists on request to those GSSs and mission control clients who need it. The GSS application, enables participating GENSO ground stations to track and establish down-link sessions with all compatible participating CubeSats. In addition, it can also be used to upload commands to satellites, should the MCC need it. Optional automated scheduling and booking negotiations with the MCC, will enable to assign dedicated passes for particular missions. The application is fully configurable, transparent to the user and although received can be stored locally, it must be transferred to the satellite's MCC whenever possible.

SATNet

The Satellite Network (SatNET) project[17, 18, 19] was created with the intention of providing a set of cloud-computing based servers, which allows to agglomerate multiple Ground Stations through the Internet, and by doing so, provide a infrastructure for multiple CubeSats ground segments. By providing these functionalities through a set of well defined API's, accessible trough the internet, it is intended to produce a distributed system composed by multiple connected ground stations.

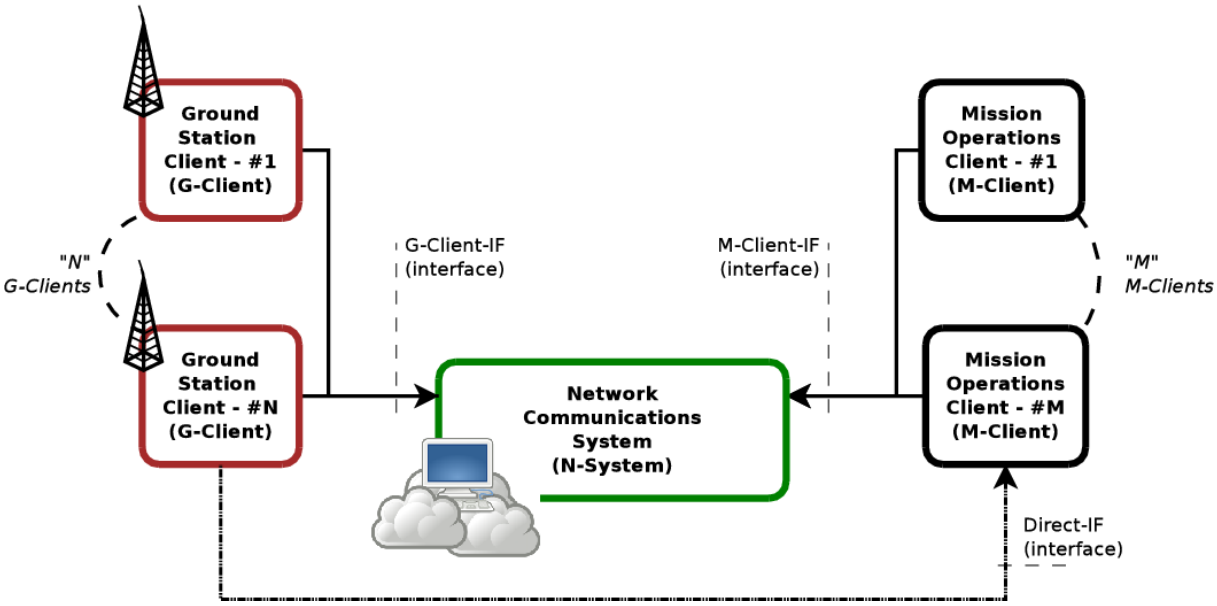


Figure 2.3: SATnet's topology.

In the SATNet system, there are three different types of entities, each with different functions, as can be seen in figure 2.3.

M-Client A set of software clients for spacecraft operators (the MCC) to remotely command the satellites.

G-Client A set of software clients used by the ground stations (which can communicate with the satellites) to interact with the SatNet network.

N-System A cloud system for the coordination of the communications between the two types of clients. It is important to note that the N-System is not a single server but a cloud-computing-based system. This way and depending on further implementation decisions, this cloud system may evolve into a network of interconnected servers that will provide the service required.

The goal the project is to provide only the underlying system, not software, to interact with the GSs communicating with the satellite. It is also not intended to handle the received data. Therefore, all that is provided by the N-System is the N-Server software and a well defined interface between it and the clients.

This way, what is transmitted between the clients is raw binary data which the satellite's corresponding M-Client, and possibly the G-Client, can decode into useful information. This way, new satellites and

clients can be easily added to the system without having to change existing software.

The client registration process in the cloud N-System is the first step for the secure distribution of keys and certificates among the clients of this network. Authentication, security and communication privacy is then assured using Transport Layer Security (TLS), ensuring that a private communication link is established between the M-Client and G-Clients.

The N-System provides management services allowing to add new clients and N-Servers. Then, it also enables to submit configurations for new satellites and make available G-Client information.

The normal scenario, or at least what is expected to be normal, is to schedule time slots in which communication with the satellites is performed trough the G-Clients or more ground stations, or its G-Client to be used once it is in LOS to a satellite. Once contact between the two is achieved, the M-Client establishes a connection with the G-Client trough which, a link to the satellite is established. Connections can either be direct trough the N-System or directly between the clients. In the case of the latter, the N-System can help establish a connection if both clients are behind a NAT or firewall.

If a satellite enters in contact with a ground station without having a scheduled time slot, it's still possible to establish a connection with the M-Client. However, the N-System must have enough information about the satellite and corresponding M-Client to be able to notify it. It's also possible to have indirect communication, as G-Client can store in the N-System data received from the satellite. Once the M-Client is able, it can retrieve the stored data.

At the time of this writing, the system is still in testing, and therefore, not yet operational, so we can only speculate the success of the project. Nevertheless, it has some interesting characteristics. Being decentralized, anyone who wishing so can create a new N-System of their own, depending on the number of N-Servers there is no single point of failure, in fact, there can multiple different networks of N-Servers, should the need arise. The possibility for indirect communication may be useful in situations were Internet connection with the G-Client or GS is less than ideal.

2.2 Commonly Used User Interfaces

In academic contexts in general, and in the ISTNanosat-1 in particular, the availability for the results to be viewed and reviewed is extremely important. Therefore one important aspect to consider is how third-parties can view and interact with the produced results of the project's missions.

In sections 2.2.1, 2.2.2 and 2.2.3, we will discuss some different ways to implement an interface, in the context of a ground segment. Then in sections 2.2.4 and 2.2.5 we address two features, independent of the used architecture, which add great value to the interface.

2.2.1 Telemetry format only

In this approach only documentation of the satellite's telemetry format is made available. Consider the CAMSAT XW-2 project[20] and its telemetry encoding specification[21] as an example. From the perspective of someone not a member of the project's team, it's this specification that can then be used

to decode any received telemetry, from a raw bit-sequence into meaningful information.

Since only the telemetry packet format is made available, if a user wishes to view the telemetry in a meaningful format, he then has to capture and decode any received raw telemetry by his own means. In all of these projects, the capture of data is done using a external toolkit usually GNUradio[22] or soundmodem, which convert sound received from an amateur radio into a bit-stream in the AX25 protocol frame format. Then, the raw binary data can be decoded into meaningful information using the provided specification.

Although we only have been addressing the CAMSAT XW-2, some other examples of projects which fall into this class are the: Serpens[23], BisonSat[24] and AESP14[25].

2.2.2 Decoder application provided

In this type of interface, exemplified in Figure 2.4, an application is provided which serves to decode any received telemetry, and therefore sparing the users of having to manually decode raw data. The relevant data is then shown in the application's GUI.

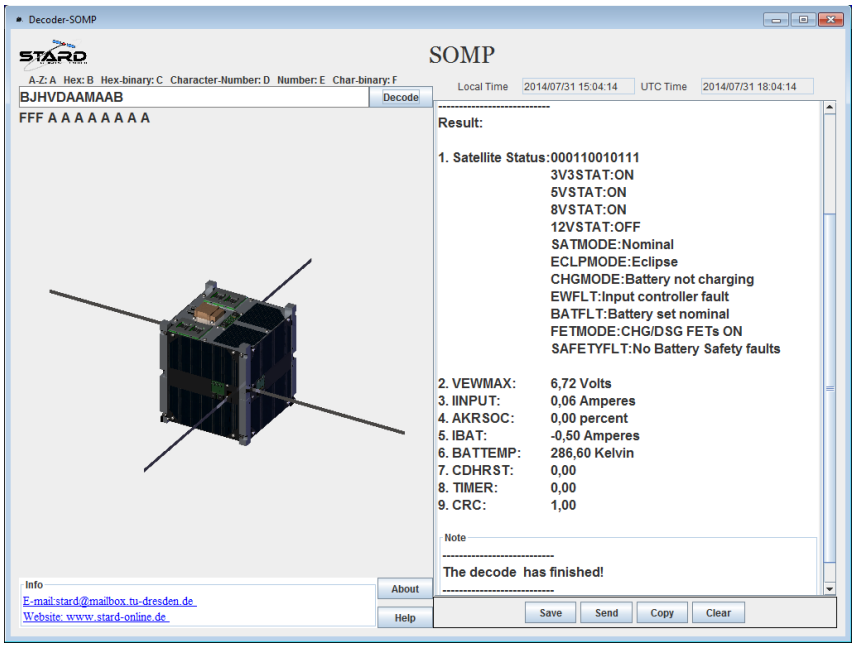


Figure 2.4: SOMP project's telemetry decoder application.

Some examples of this are the: the AMSAT Fox project[26], LilacSat Satellites[27], Deorbit[28] and the PolyITAN-1[29].

Applications are usually implemented in Java, as it enables the same application to function in any system. And due to its maturity, ample support exists to create graphical applications, which reduces the application development time.

The monitoring of the satellite is done in the same way as in sections 2.2.1, however there are certain advantages of providing an application.

Providing an application allows to add certain functionalities useful for both the project members and users. One in particular, is to allow for users to submit the captured telemetry to the project team, see

section 2.2.5. This very significant for the project members as it allows for MCC to access data produced during times which the team ground stations does not have a connection to the satellite.

2.2.3 Online decoder

The SOMP project[30] gives its operators a choice when it comes to decoding telemetry. They can either use the decoder application (see figure 2.5), or an on-line decoder[31]. The later provides an interesting alternative, as it can n can used as a replacement to the dedicated Java application, without loosing any functionality. When compared with the previous design choice (described in 2.2.2), users are presented with a choice between requiring a local application and Java Virtual Machine installed or an Internet connection.

Figure 2.5: SOMP project's telemetry online decoder.

Although the SOMP's interface is rather simple, it is enough to show a different approach to the interface. In fact the Funcube project's[32] Warehouse, another web application, shows all collected telemetry over the life of span of the project's satellites.

2.2.4 Value over time graph

The Veltox-I[33, 34] telemetry decoder application allows users to choose and view a single telemetry value in a value over time graph (figure 2.6). This is particularly useful when the instantaneous value is not very interesting by itself. But when put into context with several samples over a sufficiently large period of time, it does in fact provide greater insight. For example, to see the variation of the panel temperature over an entire orbit.

Another note worthy characteristic of this GUI is the ability to choose which value to show. When compared with the alternative, hard-coding which values to show, users gain several advantages. First, they are not restricted to what was considered useful values to the graphic, during the application development. And second, not all values are usefully at all times. For instance, a user may only want a single value do be shown at a given time. However, by only allowing one value to be display at any single

instant, some types of tasks may prove troublesome. For example correlating temperature with power consumption.

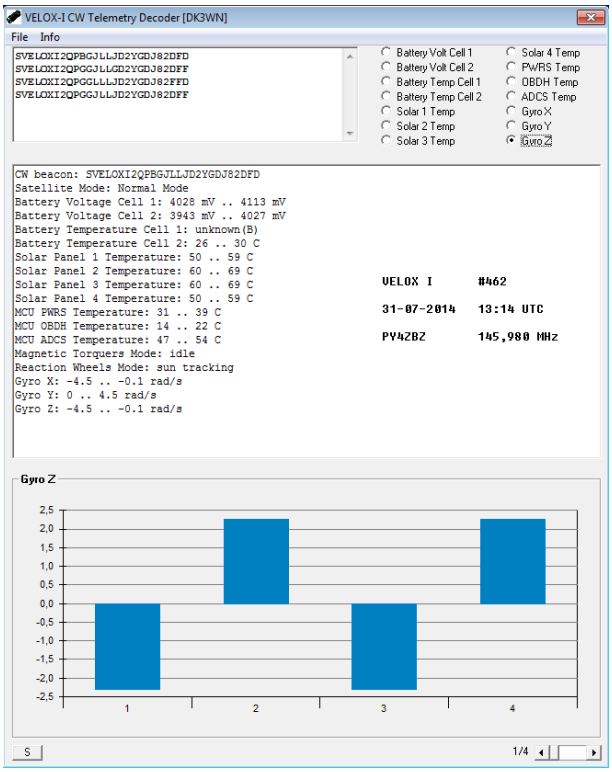


Figure 2.6: Usage example of the VELTOX-I CW telemetry decoder application.

2.2.5 Received data submission

The FUNcube project[35] spans several different missions and is being developed by a team of volunteers from AMSAT-UK.

The ground station, enables to upload all the collected data using the interface of Figure 2.7 to a dedicated server[32] where it's stored. In addition to storage, this server also enables to view the data trough a web interface showing the most recent telemetry from three of the projects satellites: FUNcube Flight Model, a testing prototype; FC1 Engineering model also for testing and UKube FC2 Payload.

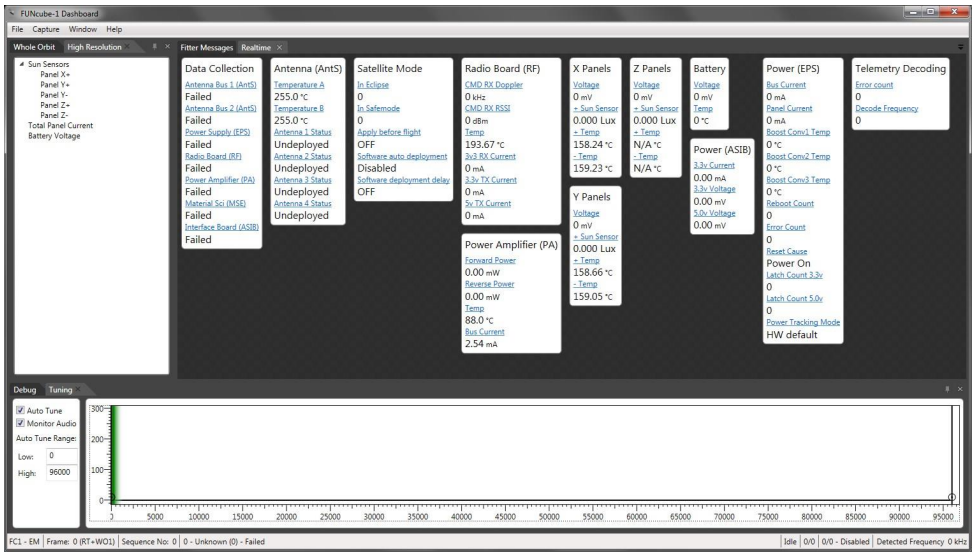


Figure 2.7: FUNcube Dashboard example taken from it's user manual.[36]

2.3 ISTNanosat-1 Architecture and Protocols

As previously mention the ISTNanosat-1 is composed by several computationally independent subsystems. In subsection 2.3.1, we briefly describe how these are inter-connected. Furthermore, in subsection 2.3.2, we describe the protocol stack used in ground-space segment radio communications.

2.3.1 Architecture

In figure 2.8 we can observe the ISTNanosat-1 architecture (at least what is relevant in a communications point of view) spans different types of links and therefore protocols each with its own properties.

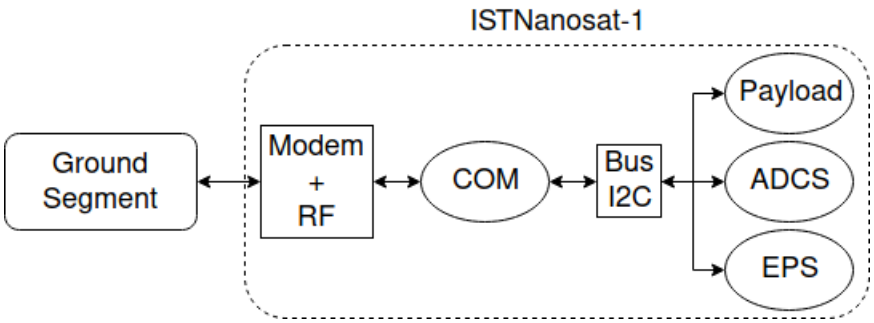


Figure 2.8: ISTNanosat-1 high level architecture

Communication between the Satellite and the Ground Segment is achieved using a radio link. The satellite's communication hardware modulates the data to send or receive from the COM (COMmunication processor board) using a dedicated link. If the satellite is in safe-mode (a more robust operation mode but with limited functionalities) the C&DH takes over the COM's role.

Inter-subsystem communication inside the satellite is achieved using the I2C bus shared by all subsystems. Messages sent/received from the Ground Segment are forwarded by the COM to the target subsystem via this bus.

2.3.2 Underlying protocols

In the ground-space radio link, a Cubesat Space Protocol (CSP) over AX25 protocol stack will be used. These are, respectively, layer 3 and layer 2 protocols in the OSI model. CSP is what allows for the addressing the subsystems from the Ground Segment. The ISTNanosat-1 has two operational modes, specifically normal and safe mode. In normal mode, the satellite will use a CSP over AX25 protocol stack. When operating in safe-mode a only AX25 is used for resource and robustness reasons.

While the underlying packet/frame format has several different fields, see figure 2.9, the ones that are relevant to the Ground Segment are the following:

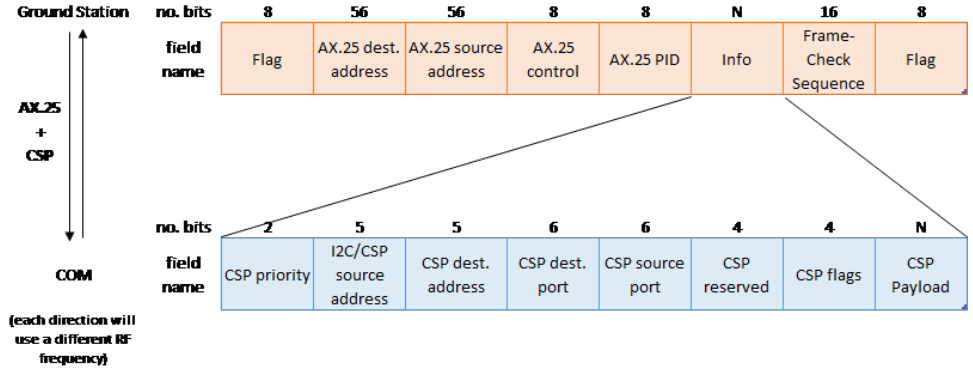


Figure 2.9: Format of a frame when in normal mode.

Source and Destination address (5 bits each) The addresses indicate which of the satellite's subsystems or ground station is the message to/from.

Operation Code (5 bits) Each subsystem has different available operations, and this field indicates which one is the message suppose to invoke.

Info (variable) Each message may have a payload field whose size depends on both the opcode and the underlying protocols. The opcode is relevant because different types of operations need different types of data to be performed. For instance, a request message asking a subsystem for a certain information would need a field to identify the information to request. (assuming that the opCode is insufficient).

In the following sections we will briefly describe each of the protocols.

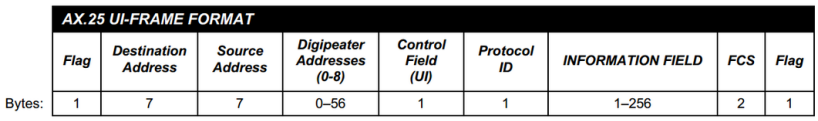
AX25

AX25 is a data-link layer protocol mainly used for radio communications in radio amateur packet networks. It was originally derived from the X25 protocol that was designed to enable packet switched wide area networks (WAN) communications. The major difference between the two is the address field size, that was increased in AX25 in order to accommodate a complete callsign (the common node identification used in radio communications) and the UI frame.

AX25 supports two modes of communications, connection-oriented and connectionless. The connectionless mode is offered by the use of UI frames (as illustrated in figure 2.10). When using the

connectionless mode the frame is sent in a best-effort manner with no checks to verify their arrival, if their content was corrupted or the order of the sent frames no action is carried out. This can be useful when performance is of more importance than correctness. On the other hand the connection-oriented mode is also available and performs all the mentioned checks. This mode is particularly relevant in bidirectional communications with complex protocols that cannot afford to ignore the mentioned problems.

Even though one can, and many do, communicate directly over AX25, using the AX25 callsign as the only address, one can also use a network layer protocol like Internet Protocol (IP) or CSP. This offers the added functionality of having routable networks.



AX.25 UI-FRAME FORMAT									
	Flag	Destination Address	Source Address	Digipeater Addresses (0-8)	Control Field (UI)	Protocol ID	INFORMATION FIELD	FCS	Flag
Bytes:	1	7	7	0–56	1	1	1–256	2	1

Figure 2.10: AX25 frame structure

CubeSat Space Protocol

CSP is a Network layer protocol specially designed to be used in space environments in cubesat related communications. It allows the addressing of not only the cubesat itself, but also any particular subsystem.

The CSP's implementation also includes a layer 4 extension called Reliable Datagram Protocol (RDP) that offers additional features that can be useful such as fragmentation, data integrity check, packet ordering and retransmission requests. In the ISTnanosat-1's case however, when data integrity and ordering is needed we don't use RDP as it would be redundant with the equivalent functionality of the AX.25 protocol we are already using. AX.25 also has the advantage of being present both in safe and normal modes while CSP, and consequently RDP, does not.

Moreover, CSP (a packet is illustrated in Figure 2.11) allows setting different priority for each message, 32 different addresses and all routing information should be preprogrammed in each node before deployment.

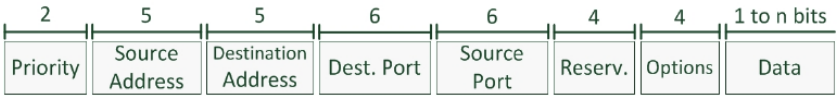


Figure 2.11: CSP packet structure

Chapter 3

Ground Segment Architecture

The Ground Segment of the ISTNanosat-1 exists to fulfill three major goals: i) allows the MCC to perform operations; ii) the retrieval of data from the satellite, both in the context of maintenance and the project's missions; iii) assure the delivery, execution and retrieval result of instructions, i.e. commands and diagnostics, to the satellite. From these goals we distilled the fundamental requirements detailed in section 3.1. From the overall objectives and derived requirements, we designed the architecture detailed throughout this chapter. In particular, we first give a top-level description of the architecture in section 3.2 before detailing each its components in subsequent sections.

3.1 Critical Requirements

1. The Ground Segment must assure the communications with the ISTNanosat-1 as to support telemetry retrieval and instruction transmission (the aforementioned operations). The challenge of this issue is the limited bandwidth and communication window of any ground station. Therefore, the end design must be able to support and use multiple distinct ground stations. As an added bonus, several ground stations also serve to improve the redundancy, and thus reliability, of the Ground Segment. However, the usage of several ground stations leads to the possibility of collisions in the ground-space radio link. Which can occur when multiple ground stations are in LOS of the ISTNanosat-1 at the same time. Therefore, some sort of mechanism must be employed to eliminate, or reduce to the absolute minimum, these collisions.

2. Each subsystem produces telemetry and has a set of operations it can perform. However, some mechanism must exist in order to transfer the telemetry and instruct the execution of these operations. Therefore, a control protocol must be specified and implemented. Particularly, it must enable the retrieval of telemetry (data and error logs) and the issuing of the instructions (diagnostics and commands). Using this protocol and the communications capabilities assured by the above requirements, the Ground Segment will collect and persistently store all telemetry and instruction results.

3. Users (both operators and third parties) must be able to analyze current and past collected telemetry from the satellite. Furthermore, properly authorized users (the operators) shall be able to issue commands and diagnostics for the satellite to execute. All of these interactions are to be perform through a web browser, i.e. the User Interface (UI) must take the form of a web application. This has several advantages, such as being platform agnostic and allows for remote interaction between the user and the internal server or machine which will serve the web interface. Additionally, all collected and stored data shall be made available in a machine friendly format (e.g. csv, json, xml, etc) in order to facilitate its transfer and independent analysis.
4. The Ground Segment must maximize the number of contacts with the satellite. To this end, and has was previously mentioned, multiple grund stations stations should be used. However, the mechanism to control and coordinate these must consider this requirement.
5. In order to maximize the usage of the limited communication window of ground stations contact with the satellite, the delay between issuing control messages and their reception must be minimized. This includes any handshake phase that may or may not exist.
6. The control protocols used to control and monitor the satellite must maximize bandwidth efficiency.
7. Since the ISTNanosat-1 is still under development, the defined telemetry and instructions may change. Therefore, the Ground Segment must have a simple process for extending it. In particular, how it controls the satellite, how the information is stored and presented.

3.2 Architecture Overview

The architecture of the ISTNanosat-1 ground segment involves three different entities, as shown in figure 3.1: The multiple ground stations, the Core server and the users or operators.

The Core server contains several services pertaining to the telemetry and operations of the satellite's subsystems, detailed in section 3.3. For example, the aforementioned services are responsible for retrieving data and the error logs of each subsystem in a controlled automatic fashion, using the in-house developed ISTnanosat Control Protocol (INCP), detailed in section 3.4. These services are also responsible for persistently storing all relevant information in a relational database which only the Core directly accesses. Then, these services expose an internal API which a RESTfull interface consumes to drive the UI of the operators. Using this interface, the project data is exposed simply, in a single point, using a well know method and in a machine friendly way.

Human interaction with the system is performed through the Core which presents the required web application (the UI) detailed in section 3.6. To this end, the core provides both: 1) the static resources that compose the interface (html, css, images, etc); and 2) a RESTfull interface to invoke the services provided by the Core, both in the context of housekeeping and (e.g retrieving stored telemetry and, if authorized, issuing instructions).

The Ground Segment must be able to use multiple ground stations. All these GSs will be linked and controlled by the Core, as detailed in section 3.5. In order to avoid the collision of radio transmissions,

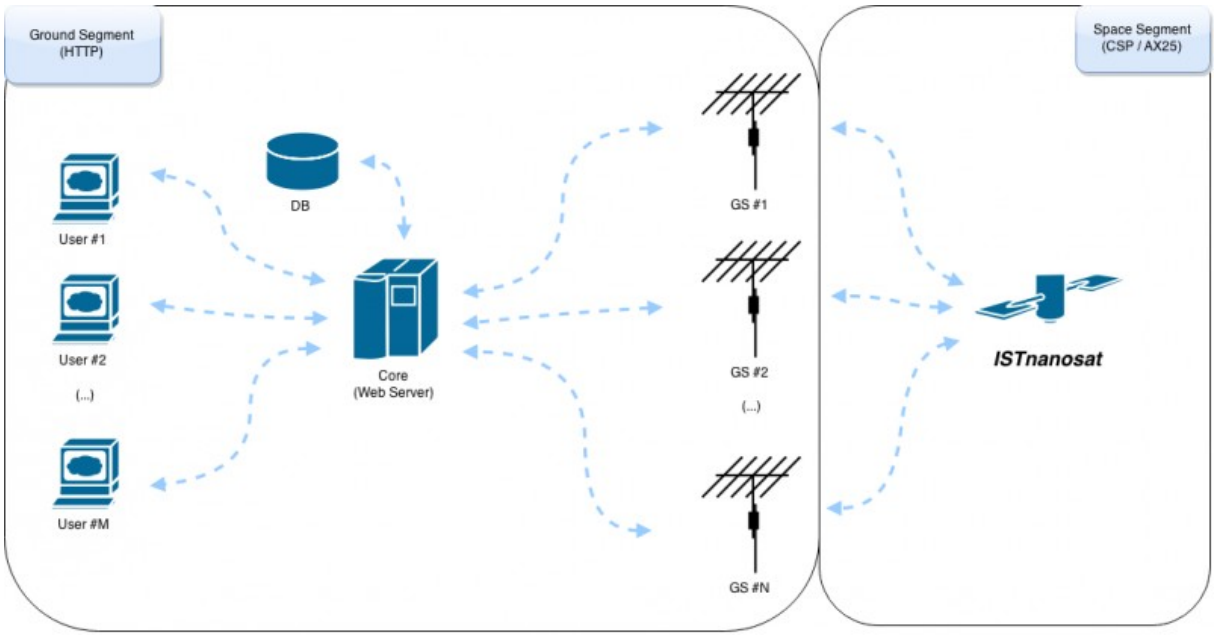


Figure 3.1: ISTNanosat-1 ground segment topology.

when the satellite is within the LOS of a group of GSs, the Core-Server will (in a centralized manner) designate a single GS to perform all the transmissions. The created channel is then used by the Core's services to communicate with the satellite's subsystems. Should the core fail, the ground stations will only communicate with the satellite (of their own initiative) if these were previously instructed to do so.

In order to guarantee that the ISTNanosat-1 only executes instructions from the MCC, and to prevent unauthorized access, a security mechanism was designed. One that takes into consideration that the ISTNanosat-1 is a distributed system, operating over several insecure channels. Our solution, detailed in section 3.7 does not rely on security by obscurity, it is in fact based on the ability to authenticate critical messages.

3.3 Services

Figure 3.2 shows an overview of the Core's architecture. In it, the center block contains the various services which handle the internal logic for data retrieval and instruction execution. The Ground Station Handling block provides a simple API for sending and receiving messages from the Satellite. It Abstracts the internal details of the Core-ground station and ground station-satellite communications. The messages are constructed, serialized and unserialized using the libINCP block. The REST interface uses interacts with to 1) retrieve collected telemetry and executed instructions; 2) issue instructions. The database is used to persistently stored all telemetry, executed instructions, user data and other Ground Segment related information.

The data service is responsible for the retrieval of produced data from the satellite's subsystems. This data is organized in variables were each has an identifier unique per subsystem. Thus, each variable can be uniquely identified by a *(subsystem, ID)* tuple. Additionally, each of these variables has a type

specified by the subsystem's author such that these can be used, stored and displayed.

Consider for example, a variable which describes the charge level of the satellite's battery represented by a percentages and its type is a float. Internally, the battery sensor may be polled several times a second. However, due to memory and throughput restrictions, the subsystem may reduce this raw data into a single *(value, Timestamp (TS))* tuple to be retrieved using the INCP. For example, if produced values are the result of the average of all polled values over a minute, then only this average is transferred to the ground.

The rate at which this values are produced is inconsequential to the Ground Segment whose minimal logical unit is the *(value, TS)* tuple. The Ground Segment only assures that each transferred value is stored and then visualized.

The other kind of telemetry is errors. These are similar to the data but have a few differences. Firstly, these are produced infrequently and typically can have no value at all. Meaning, the Ground Segment only knows that it occur and when.

Each subsystem has a set of commands which can be executed if so instructed by the a authorized user whenever the satellite is in range. Each command is identified by *(value, opCode)* tuple and has input/output values. These arguments and values, are specified by the subsystem and have well defined types.

Like errors and data variables, diagnostics are uniquely identified by a *(value, TS)* tuple. As previously mentioned, the goal of diagnostics is to actively test a component and verify its status. Once issued, the resulting test can take a long to be performed and may even finish once the connection window is closed. In such an occasion, the Core retrieves the results on the next window. These result are similar to command results but also include the start and end TS of the diagnostic execution. In addition to on the fly issuing of diagnostics, these can be scheduled to be issue at the next connection window with the satellite.

Commands and diagnostics need to be send securely. Simply put, whenever the command or diagnostic services are to communicate with the satellite a SS must first be established. Only then, can critical messages, those relating to commands and diagnostics, be sent/received through the Secure Session service.

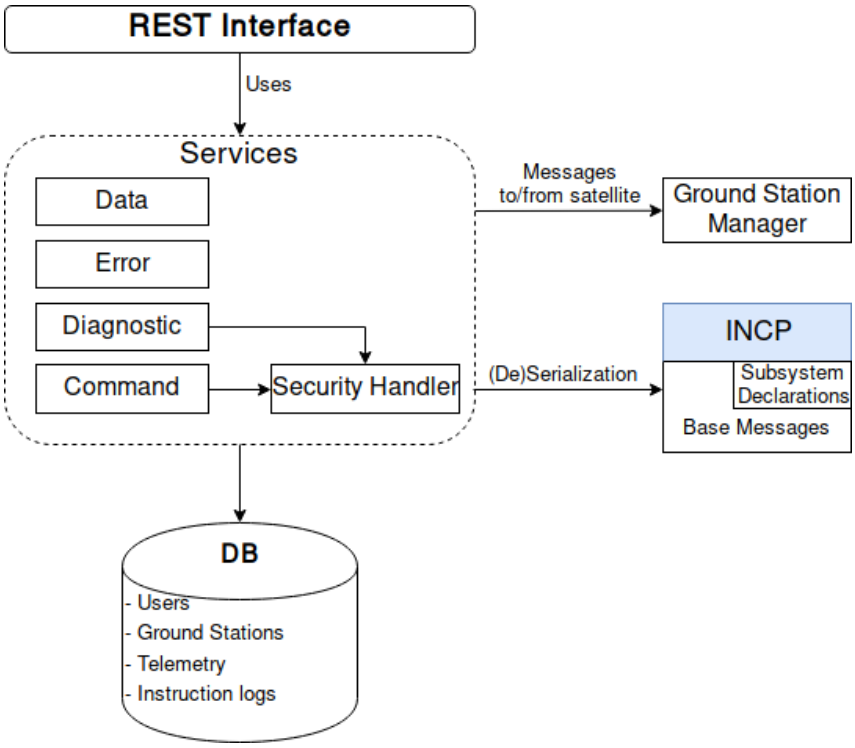


Figure 3.2: Core's Architecture

3.4 ISTNanosat-1 Command Protocol (INCP)

The ISTnanosat Control Protocol defines several messages and how these can be used to support the mentioned services. Furthermore, and because each subsystem is computationally independent, it can also be used for inter-subsystem communication and not just for Space-Ground communications.

Before we can begin detailing the ISTnanosat Control Protocol (INCP) we must first address some issues regarding message passing of messages over an unreliable channel. Specifically, the radio link between the ground and space segments.

3.4.1 Message handling

The ISTNanosat-1 will use a CSP over AX25 protocol stack when operating in normal-mode and AX25 only in safe-mode. The standard way of using AX25 is with UI frames, were there is no network session established and frames are sent without any sort of delivery guarantees (other than a error checking via a checksum). CSP on the other hand, specifically the libCSP, has a builtin RDP implementation which could be used for message control.

However, when the satellite is operating in safe-mode we wont have access to CSP and therefore RDP protocols while still needing to control it. Note that we could use two different control mechanism for when operating in normal or safe mode. However, we saw to no advantage to the added complexity this would bring. Therefore, we must address the problems associated with message transmission over an uncontrolled link. The issues to consider are then:

Lost Messages Messages that were sent but not received.

Repeated Messages A possible problem is the reception of duplicated messages. This can occur for a variety of reasons, but a common cause is when a host re-sends a message, because it mistakenly detected that the first one did not arrive. This is particularly worrisome for messages that trigger sensible procedures and require at-most-once semantics (i.e. are not idempotent). For instance, we do not want a subsystem to reset twice should the mechanism that handles message delivery resent a message unnecessarily.

Detecting Connectivity Connectivity between the ground segment and ISTNanosat-1's subsystems can be gained or lost as a consequence of two factors. Firstly, each subsystems is computationally independent and can fail independently, and thus stop responding to sent messages. Secondly line-of-sight between a ground station and the satellite will inevitably be lost which ceases all radio communications.

Message Ordering Since we may want to perform several operations at the same time we will need to send several messages in sequence. Since there may be assumptions in the order in which the messages are received and processed, their order of delivery must be assured.

Message Integrity Finally the issue of message integrity must be addressed. However, note that AX25 protocol already performs frame integrity checks, meaning it is not an issue but we mentioned it for the sake of completion.

3.4.2 Protocol overview

The ISTnanosat Control Protocol (INCP) can be described according to a client-server type model with asynchronous message-passing. In this approach, the Ground Segment (the client) can send one or more messages (requests) to the satellite (the server), or more specifically its subsystems. When received, these messages will trigger an action or operation, ultimately resulting in a message (response) being sent to the ground in order to inform the client of the success or failure of the operation.

In order to handle the issues mentioned in section 3.4.1 and for purposes of it's normal functioning, the protocol will have certain characteristics:

- Firstly, there are several types of messages, each having a header containing an Operation Code (opcode) fields identifying its type. The opcode has a 5 bit size, such that it can be inserted in the destination port field of the CSP header, which would otherwise be unused.
- Secondly, all messages can be interpreted in isolation. In other words, we must not have to know what messages were previously exchange in order to "understand" a received message. This property is desirable because it allows the flexibility to have the satellite unilatery send messages if need be.
- Thirdly, detecting that the satellite is in LOS can be achieved by detecting the satellite's periodicaly sent beacons.

- Finally, we need a reception control mechanism which handles the issues detailed in subsection 3.4.1.

For this last issue regarding reception control, two options were considered: 1) we pondered designing and implementing a minimalist custom transport protocol; and 2) using AX25 in Connection-Oriented mode. In this mode, and after a connection is established between the two hosts, the AX25 protocol assures the properties discussed in the previous subsection. The comparison of the two alternatives resulted in table 3.1. Ultimately, we opted for using AX25-CO due to its many advantages, which were not clear at first glance. Particularly, AX25 is much more reliable, as it is an already used protocol. Additionally, an implementation is already available. However, using a custom transport protocol would enable some finer grained control but as indicated it didn't outweigh AX25-CO advantages.

	AX25-CO	**Custom Transport Protocol**
Robustness	Proven and tested solution	New and untested solution
Functionality	Connection window: 8 or 128. Frame loss recovery: Implicit reject, selective reject or selective reject-reject.	Whichever functionalities we could have chosen to implement.
Overhead in connection establishment	1 round trip time message (2 AX25 SABM messages). AX25 also has a parameter negotiation phase, but it can happen while data is already being exchange by temporarily using default values.	1 round trip time (2 messages)
Overhead when in steady state	Equal (1 byte of overhead when compared to a simple UI frame.)	
Code reutilization	Already implemented for Linux systems. Some parts can be reused in the Satellite.	Full novel implementation.
Performance	Similar performance in throughput. In terms of CPU and memory usage it's unknown.	

Table 3.1: Table comparing the pros and cons of using a Custom built transport protocol vs using AX25 in Connection-Oriented mode

Due to the bandwidth requirements we avoid using Type-Length-Value type formats such as ASN-1, expecially text based ones such as XML, in order to maximized the bandwidth usage. Thus, we opted for a binary static format were, as mentioned, the type of the message is deduced from it's opcode.

3.4.3 Commands

The command execution service is the simplest one. To remotly request the execution of an operation two messages are required: i) an initial *function* message, Figure 3.3a, requesting the subsystem to perform the operation, and ii) a reply message *functionRet*, Figure 3.3b, carrying the result of the operation. Both messages transport some predefined data specific to each operation, uniquely identified by the opCode, were each subsystem has it's own set of operations. Note that in the case of *function* messages, this opCode is the one in the header. This being the only message which can have multiple different opcodes for the same message type. As opposed to all other messages were each have a

unique opcode.

(a) *function m*essage format (b) *functionRet* message format

Figure 3.3: Command related messages: *function (a*) and *functionRet* (b)

Once a subsystem receives the *function* message it executes the associated operation, as in Figure 3.4. The reply message also carries the command opCode (the same as in the *function* message) thus associating the command with the corresponding reply. Furthermore, the *functionRet* message may also contain data as defined by the operation in question.

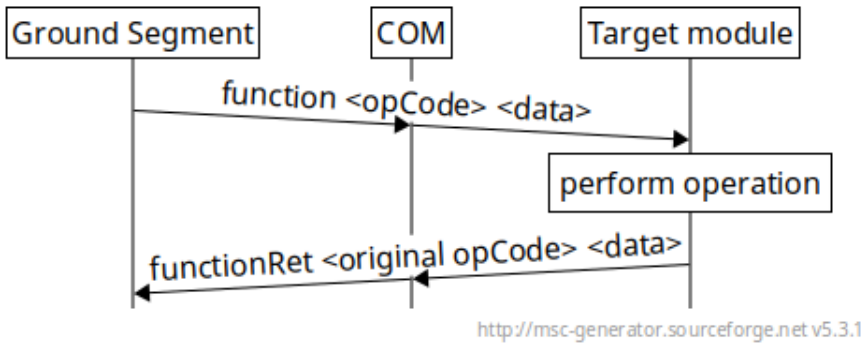
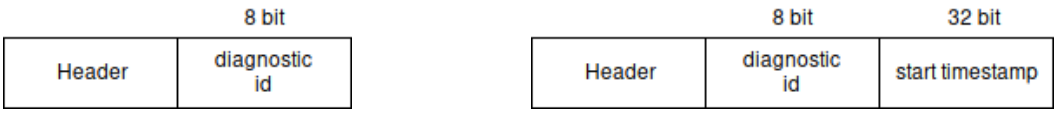


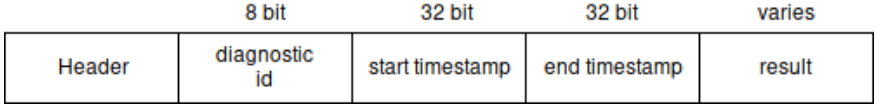
Figure 3.4: Example of a command execution message exchange.

3.4.4 Diagnostics

The exchanged message needed to issue diagnostics can be though of as an expansion of the command execution exchange. In particular, it supports the long execution times that some diagnostics require. First, we send a *Diagnostic* message, the format of which is shown in Figure 3.5a, to instruct the subsystem to start performing the diagnostic identified by the value of the *ID* field, as can be seen in figure 3.6.



(a) *Diagnostic* message format. (b) *DiagnosticStart* message format.



(c) Diagnostic result message format.

Figure 3.5: Diagnostic related messages: *Diagnostic* (a), *DiagnosticStart* (b) and *DiagnosticResult* (c). Note that, *Diagnostic* and *ReqLastResult* messages have the same format.

Since diagnostics in general may take long time to be performed, it is useful to know when said diagnostic was started. This is done by a *StartingStart* message, Figure 3.5b, sent as the response to the

original *Diagnostic* message. It indicates that the diagnostic has started being executed. Furhtermore, it indicates the time it start via the TS field.

Once a diagnostic completes, the subsystem sends a *DiagnosticResult* message, Figure 3.5c. This message includes two timestamps, on for the start and another for the end time instants, and the resulting data. Note, that the data field is individually defined for each diagnostic but opaque to the INCP since all that it's needed is the size it occupies. Additionally, if the connection to the satellite is lost, we can at a later time, use the *ReqLastResult* to request the last result of a certain diagnostic.

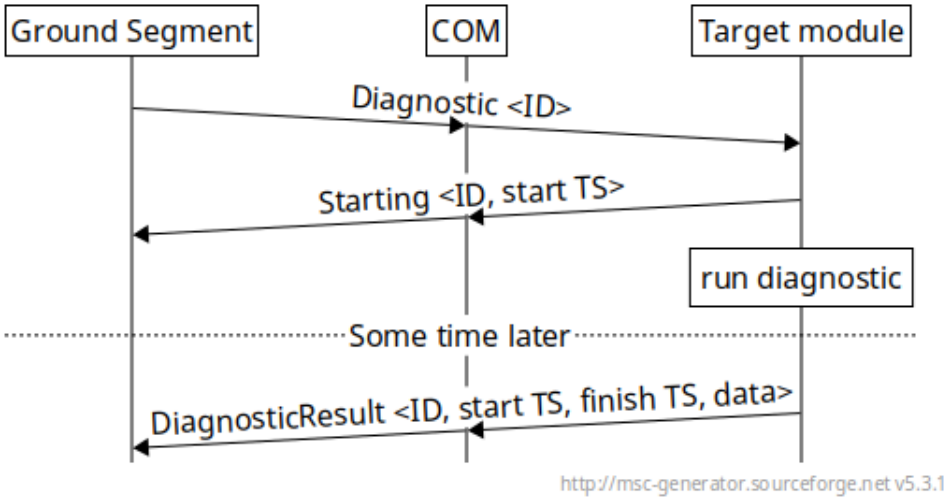


Figure 3.6: Normal case scenario for executing a diagnostic.

StartDiagostic Asks the receptor to perform a diagnostic indicated in the message.

Starting The subsystem reply indicating that it has started performing a diagnostic.

DiagnosticResult This message that contains the result of a diagnostic.

ReqLastResult Message used to request the result of the last ran diagnostic of a certain type. Useful when a diagnostic ended after the satellite lost connection to a ground station.

3.4.5 Errors

Every subsystem of the ISTNanosat-1 can produce a range of possible errors. Like the diagnostics, errors are identified by a per subsystem *ID*. The defined messages can then be used to not only retrieve error occurrences, but also related data. The size (in bits) of the each error's associated data, like in diagnostics and commands is opaque to the INCP, which only needs to know the size of the data to transmit.

The transmission of this error related information is divided into two phases, shown in Figure 3.7. First, a *reqErrorLog* message is sent, shown in Figure 3.8a. This indicates to the receiving subsystem, that the sender is interested in all errors that occurred after the time instant indicate by the *TS* field of the request message. The response message *ErrorLog* then contains a sequence of *(ID, TS)* tuples for

the relevant time period, as shown in **??**. The second phase, is simply a request-response exchange for each reported error, to retrieve data associated with the error.

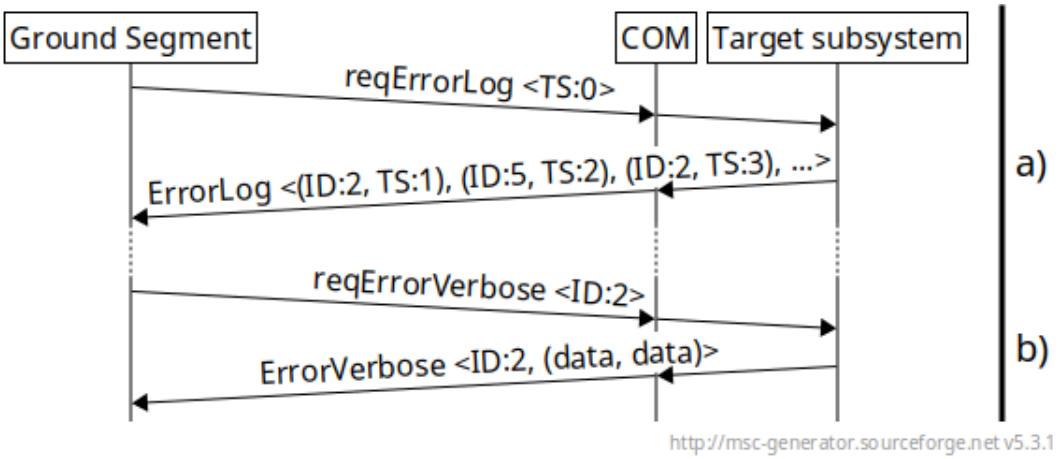


Figure 3.7: Error related message exchanges.

Note that the usage of *TS* in *reqErrorLog* messages is important since error and data requests do not cause the underlying information to be deleted. Therefore, multiple requests in a short time would cause most of the same information to be retransmitted unnecessarily. By specifying the time period that we are interested in we avoid this issue.

reqErrorLog Asks the receptor to send a log of the error occurrences in the subsystem. Timestamp indicates that we are only errors more recent than the TS should be transmitted.

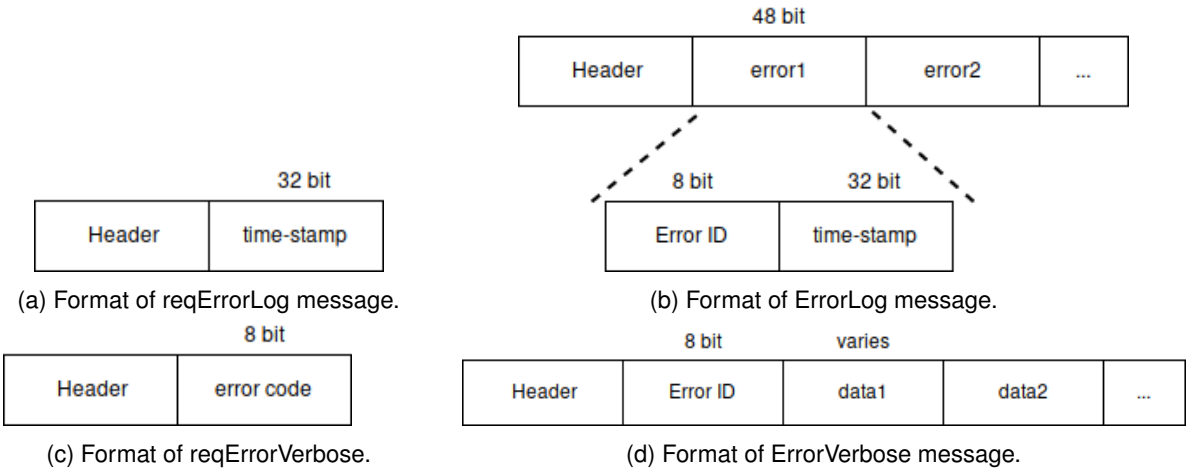
ErrorLog Contains a list of tuples *(ID, TS* for each error's occurence. The *ID* identifies the error and the *TS* field indicates the instant the error occurred. An empty list means no errors.

reqErrorVerbose This message requests all data pertaining to a certain error.

ErrorVerbose Response to previous message. Note that the example response message has two data fields. This is because the error in question occurred twice and the subsystem stores the data pertaining to both occurrences. Furthermore, and in order for the client (in the example the Ground Segment) can associate data to *TS*, the data fields are sorted from most to least recent error. The subsystem could also only store the data for the last occurrence, at which point it would send a *ErrorVerbose* message with only 1 field. This may make sense for some errors but not all. As such, this message is prepared to send data regarding multiple errors.

3.4.6 Data retrieval

Each subsystem produces different kinds of data during the lifetime of the satellite. These can, for instance, be the result from sensors readings or be the operational status of the subsystems. Examples include the attitude, battery charge, solar panel current, components temperature, subsystem uptime, etc. For most kinds of data, we are interested in both the instantaneous and the accumulated values.



(a) Format of reqErrorLog message.

(b) Format of ErrorLog message.

(c) Format of reqErrorVerbose.

(d) Format of ErrorVerbose message.

Figure 3.8: Error related messages: *reqErrorLog* (a), *ErrorLog* (b), *reqErrorVerbose* (c) and *ErrorVerbose* (d).

Consider the battery charge, we are interested in all values so that we can see its change over time. This data class is designated as periodic variables. Another class are those that only the instantaneous value is important while previous values are not, such as the uptime. We designate this latter class as singular variables.

Retrieval of singular variables is relatively simple. First, one *reqData* message, Figure 3.9, requesting a certain variable identified with *ID*, example in figure 3.10a. Note that, the message's *TS* field is ignored for this class of variable. This request is then replied with a *Data* message, Figure 3.11, carrying the value. Each variable has a specified opaque data type, which indicates to the INCP its size. Note that *Data* messages are composed by one or more data fields where each can be of one of two types:

Periodic A sequence of values of the same variable. The dataID field identifies the data. The number of data values indicates how many data values there were sent for this particular dataID. Then, a sequence of (data, time-stamp) tuples follows.

Singular This type of data field, only contains one value, therefore it does not need a data length field nor a instance number field as in the period type.

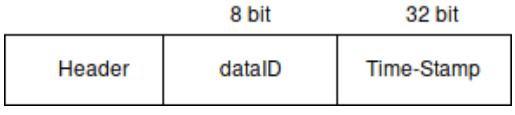


Figure 3.9: Format of a *reqData* message.

After a period of no connectivity between the ground and the space segment large amounts of data can be accumulated. In such situations, the data to be transferred cannot fit inside a single AX25 or CSP mtu, and consequently a single *Data* message. Therefore, the data is divided between several *Data* messages.

When requesting data, we are usually interested in more that a single variable. For instance, when we want the overall status of a subsystem. This case is exemplified in the exchange of figure 3.10b, were

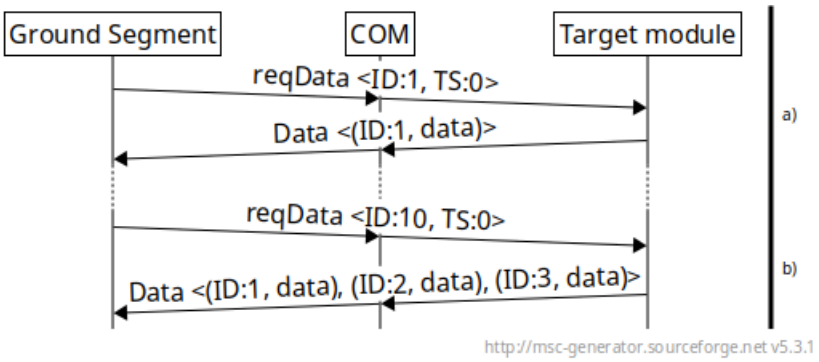


Figure 3.10: Retrieving singular data variables

we request the value of variable with *ID* 10 (an arbitrary id). This *ID* is a special identifier. Whenever a request for this virtual variable is performed, the subsystem interprets the request as if it were a request for variables with *IDs* 1, 2 and 3. This example shows that using these virtual identifiers we can substantially reduce the number of needed messages for requesting certain sets of variables.

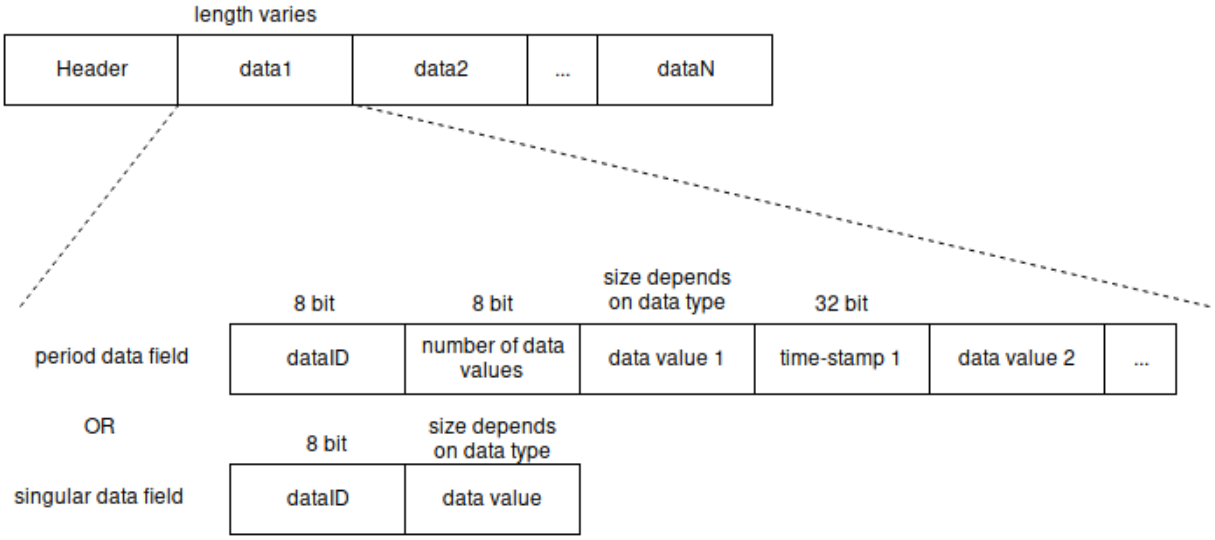


Figure 3.11: Data message format.

Retrieval of periodic variables is slighly more complex than for singular ones. Each periodic variable value is stored alongside a timestamp corresponding to the measurement time instant. Requesting the values of this kind of variable also starts with a *reqData* message. Like for singular variables, the *ID* field indicates the intented variable and the *TS* indicates the only values measured more recently than the timestamp are to be transmitted. The response message *Data*, then contains a sequence of *(data, TS)* tuples corresponding to the requested variable. The data field is the data we are interested in retrieving and the TS is the time instant at which the value was measured. The message, *Data*, is the same as in the non period variables, and can in fact contain a sequence of both singular and periodic variables. The reason being that all are predefined meaning we can use its *ID* to determine its class.

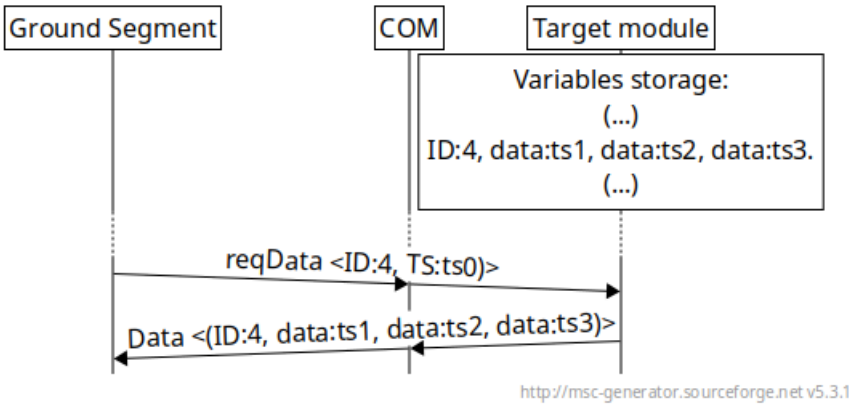


Figure 3.12: Retrieving sequences of periodic data from the satellite using reqData and Data messages.

3.5 Ground-Space communications

In order to perform its function, the Core server needs to communicate with the satellite. This is done indirectly through the ground stations. As mentioned before, each ground station only has a 10-15 minutes communication window in addition to the low throughput of the radio link, hence the need for multiple ground stations. However, this leads to several issues which must be addressed. Particularly, how are the ground stations controlled, how is the Core-Satellite communication multiplexed between them and which types of ground stations or functionalities are required. These issues are detailed in subsection 3.5.2 were the designed solution is explained. Furthermore, in section 3.5.1 we detail the major options taken which resulted in the end design.

3.5.1 Design Decisions

Core embedded ground station

In the first iteration of the Ground Segment architecture, the Core and the main ground station were to be the same entity. The remaining ground stations were then to be controlled by this main one.

Ultimately, this first design was abandoned because the current one is equal in terms of functionality, i.e. both support multiple ground stations, but the current design is much easier to implement while providing greater flexibility. In fact, if both the Core and main ground station processes were executed in the same machine, we achieve the same topology as in the first design. However, the main ground station's machine can now fail without bringing down the entire Ground Segment. Furthermore, the decoupling between Core and main ground station means that we could run it in an environment with better availability guarantees. Particularly, because the ground station application needs to be executed in a machine connected to a radio. Thus, its limited to the conditions of said machine and the space were it's located.

Regarding the implementation, the Core does not have to deal with CSP and AX25 which are implemented in C, at least not directly. This was a major factor in deciding to develop the Core server using Python. And therefore benefit from the advantages of using a high level language.

“Smart” vs “Dumb” Ground Stations

Having decided to separate the Core from the ground stations, the next choice to be made was if the ground station should “understand” INCP or if these would simply forward INCP messages between the Core and the ISTNanosat-1. These two choices were, respectively, denominated dumb and smart ground stations.

Regarding the “dumb” ground stations option, these would be limited to forwarding messages at the behalf of the Core, i.e. as simple gateways. The INCP messages would then be constructed in the Core which sends sequences of bytes ready to be sent. The ground station’s task would then involve encapsulate them in a CSP packet or AX25 frame and sending the resulting frame or packet. Inversely, it would have to unpack the raw INCP message from frames/packets received from the satellite and forward them to the Core.

In the “smart” ground stations option, these would be somewhat autonomous entities. These could then communicate with the satellite using the INCP even without the direct intervention of the Core Server. However, the ground stations would ultimately have to relay the telemetry to the Core segment and forward commands to the satellite.

The ability to schedule instructions to be sent from the ground stations is possible in both options. This would mean that a ground station with poor internet connection could still be used for ground-satellite communications even during the time in which the the ISTNanosat-1’s is in LOS the ground station is not connected to the Core. However, the “smart” ground stations could conceivably interpret the satellite’s replies and send a second round of requests or more.

Secondly, and the critical advantage of the first option, is that it enables the usage of Distributed ground stations networks. Recall from section 2.1.3, that these ground stations are from third parties with different software which do not understand INCP. Using these ground station networks we thus increase the overall number of ground stations available.

Ultimately, we opted to use the “dumb” ground stations.

Profiles

Additionally, and as previously addressed, third-party ground stations will also participate in the ISTnanosat-1 Ground Segment. However, only some of these will be trusted to also handle critical messages. Even before the security scheme is applied. This problem motivates the need for a rather flexible assignment of permissions. One solution is to use a profile system were each ground station is assigned one such profile. Each profile would have different permissions which Core uses when selecting which ground station to use as gateway.

These profiles would be divided into different levels, each more permissive than the last. The base level would have a profile which only allow the transmission of telemetry. This would be the default for all ground stations. Then a second level would enable the ground stations to also send critical messages and thus use the entire INCP. The third and final level is the main ground station, which is to have priority over all others in terms of multiplexing to interact with the satellite, since we want to prioritize the usage

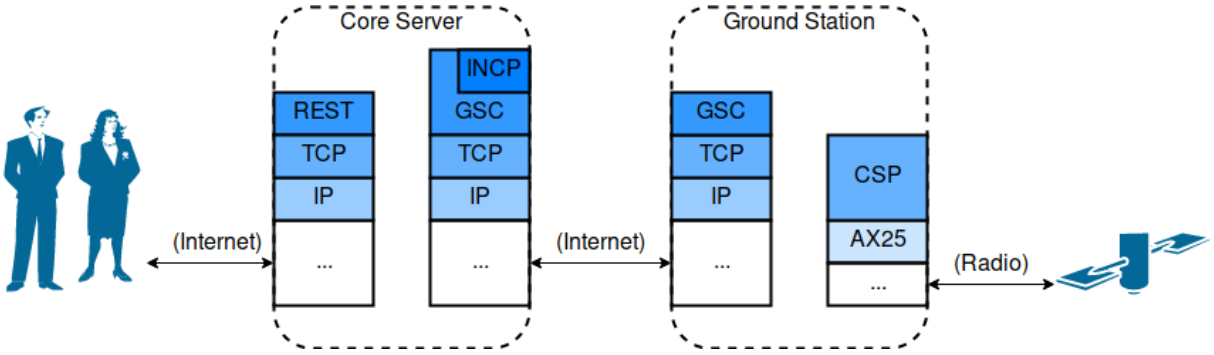


Figure 3.13: Ground Segment protocols.

of our own ground station. Additionally, each profile could be further configured with characteristics not directly related with ground-space communications. For instance, we may define that the main ground station can only connect to the Core if it satisfies a network address and mask.

3.5.2 Ground Station Command and Control

As mentioned in the previous subsection, the ground stations serve as gateways for the Core to communicate with the satellite. Furthermore, and to support ground stations in remote areas with unstable internet connections, these must store all received data while not connected to the Core. Data which is then forwarded to the Core once a connection is reestablished. Additionally, the reverse should also be possible. The Core may send messages to the ground station for it to later relay to the satellite when it enters into LOS.

The command and control of the ground stations has three major components, the Core's ground station handling logic, the ground station logic itself and a, simple, control protocol for communications between the two. In the Core server this logic component can be seen in the *Ground Station Manager* of figure 3.2. The overall protocols of the Ground Segment can be see in figure 3.13 including the Ground Station Control (GSC).

The aforementioned GSC is a message passing scheme used for communications and control of the ground stations by the core. This is what allows for the core to instruct the ground station to forward messages, order it to connect with the satellite and other control logic.

The Core side ground station management, seen in figure 3.2, is what allows for the ground stations to connect. Then once, properly authenticated, it's trough it that the services transmit and receive messages with the satellite. In particular, it's what handles the multiplexing of the Core-satellite communications trough the ground stains abstracting this issue from the rest of the Core server.

To avoid collisions in the ground-space link the chosen approach is for the Core to select a Designated Ground Station (DGS). The selection process is relatively simple and is performed in real-time by first filtering all ground stations in range of satellite, known by received beacons, and then selecting the one with the highest priority. However, if there is no difference in priority, then current DGS is kept.

Selection of the designated DGS is based on its profile. Each of the above profiles has a priority matching its capabilities were the main ground station has the highest priority.

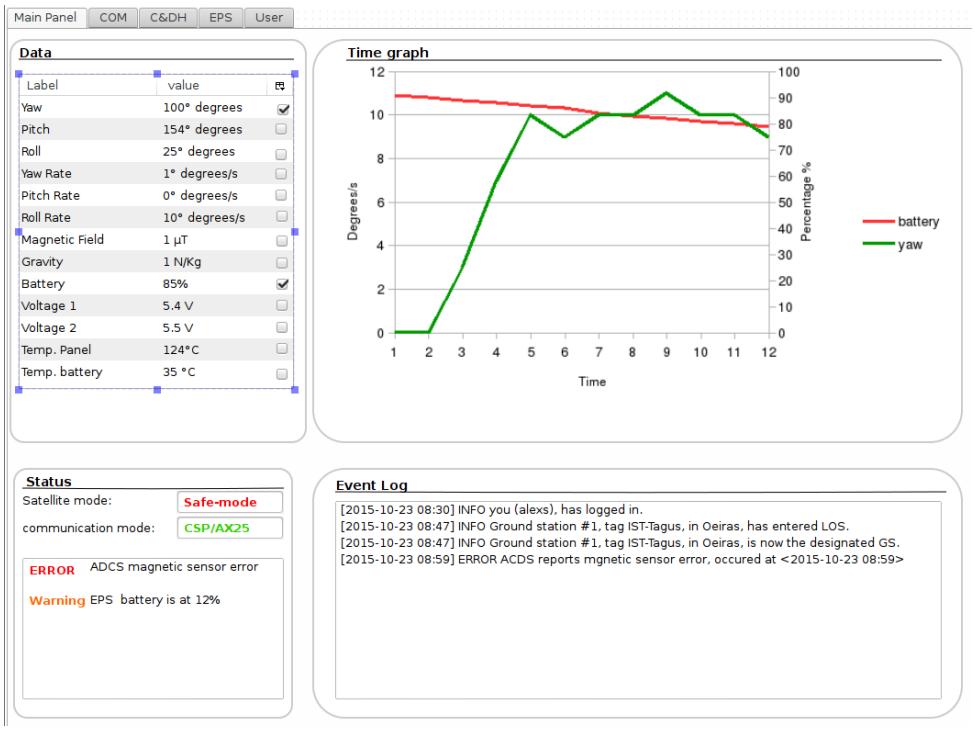


Figure 3.14: Main Panel's Non-functional Prototype

3.6 User Interface

In this section we will address two issues. First, in subsection 3.6.1 we detail the requirements of the interface itself. Then in subsection 3.6.2, we detail the necessary Core's components to support the interface. In particular, how the mapping between the services described in section 3.3 and the RESTfull interface.

3.6.1 Interface Requirements

As previously mentioned, there are several types of users. Specifically and by increasing order of permissions: guest, privileged and administrator. The guest user is anyone who wishes to access the data collected from the satellite but which is not a member of the ISTNanosat-1 project. Privileged users, on the other hand, are active members of the project and therefore have permission to issue instructions. Administrator is simply a privileged user who can create or remove other privileged user accounts and alter the profile of ground stations.

A non functional prototype was produced, shown in figure 3.14, which helped guide the development of the interface and validate the design with the entire ISTNanosat-1 team.

As shown in the prototype, several noteworthy items should be mentioned. Firstly, the time-graph shown allows the users to visualize the collected data. Note that the lines in the time graph are configurable by the user, such that it can select which periodic variables are to be shown. Secondly, the most important status information are shown in section to the effect. Finally, all errors, commands, and diagnostics are shown in an event log.

3.6.2 Core Front-end

As mentioned, Client-Core communication is performed over HTTPS via a REST interface in the Core. This interface is built upon Flask[1] which also serves the various static resources (CSS, JS, HTML, images, etc) of that make up the web application.

3.7 Security

The ISTNanosat-1's ground and space segments are a distributed system with several different entities and communication channels. From these entities, only the Core server and satellite are directly under our project's control and are thus, the ones we trust without reservations. The necessary mechanisms which assure that only authorized users can issue instructions executed by the satellite is specified in this section.

Before addressing the designed scheme, we detail the context of the problem in subsection 3.7.1. Then, the designed solution is detailed in four subsections: The user-core communications are detailed in subsection 3.7.2; communications between the Core and the ground stations in subsection 3.7.3; and the mechanism for ground to space segment communications in subsection 3.7.4 and subsection 3.7.5.

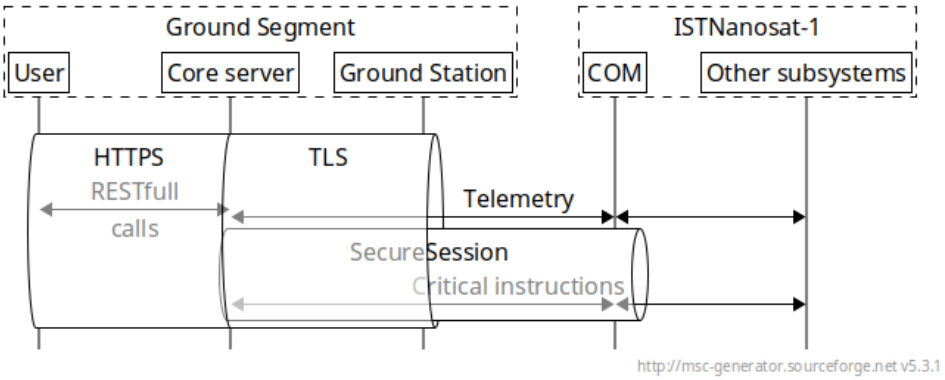


Figure 3.15: Security scheme overview

3.7.1 Requirements and threat-model

In order for overal overall communications to be considered secure, the designed scheme must assure certain requirements. In particular, the authentication of 1) users and ground stations towards the Core server; And 2) critical messages exchanged between the Ground Segment and the Satellite/Space Segment. Consequently, this means assuring the origin authentication for each exchanged message, as to guarantee the sender's identity and the message's integrity. [2]

In addition the above requirements, the designed security scheme also takes in to account the operational constraints it will have to support, specifically in terms of:

[1] The Flask homepage can be fount at: `http://flask.pocoo.org/`
[2] The terms used, such as Authentication, Identity Verification and Integrity, are defined in [37].

Computation Since the satellite's subsystems are essentially embedded systems, these have restrictive levels of processing power and memory capacity. Furthermore, different subsystems have different hardware characteristics, hence, the processing and memory overhead of the solution must be minimized.

Channel Throughput The ISTNanosat-1 ground and space segments encompasses several different channels with different characteristics, specifications and throughput. However, the bottleneck in ground-space communications is the radio link between the ground stations and the Satellite. This link, has three possible throughputs: 1200 bit/s, 9600 bit/s and 48000 bit/s. Therefore, the designed scheme must be able to operate in the lowest possible throughput setting.

Connection Window Since the satellite will be in LEO, each ground station will only have a small window of communication in which it can reach the satellite. This fact, compounded by the low throughput of the channel, motivates the need for a fast initialization phase.

Note that purely data retrieval related messages, such those pertaining to data or errors, do not alter the state the satellite's state. Indeed, these messages are typically inconsequential to the normal functioning of the satellite. Since the goal of this security scheme is to protect the satellite from harm, the focus should then be securing messages which can be used for nefarious ends. Therefore, the designed security scheme will focus on securing critical messages, which do alter the satellite's state.

Having defined the operational constraints, we then defined the attacker's capabilities. Firstly, we assume that an attacker can access the channels used, and interact both passively, i.e. listening to the communication, or actively, i.e. sending and receiving messages. Secondly, the attacker can spoof the AX25 callsigns of the hosts in ground-space link. Thirdly, we assume that the entire protocols stack, including the INCP, in is known and thus anyone can "understand" the communications taking place if given the opportunity.

3.7.2 User-Core

Since users interact with a RESTfull interface the standard way of authentication is through passwords and cookies over HTTPS. In this approach, when an user interacts with the web application, he first provides its username and password. The server then validates these credentials and returns a non-persistent cookie to be used for posterior REST invocations, as per [38].

In order to prevent timing-attacks discover valid usernames, when validating credentials the verifying entity always performs the expensive calculations, i.e. the hashing, even if the provided username does not exist [39]. Furthermore, and for the same reason, both in the case of either the username or the password being invalid, the same error message is returned to the user.

To safeguard against database leaks, the server only stores the MAC of an iterated, salted hash using the Argon2 [40] algorithm as best practices dictate [41]. Furthermore, the database is also encrypted at rest. Using this method, password verification and offline brute-force type attacks are impossible without crypto key access even if the database is leaked. However, there is always the possibility that the key

is also leaked. In this eventuality, the protection of this method is equal to that of storing the passwords' iterated, salted hash.

Since the RESTfull interface is only exposed over HTTPS the integrity and secrecy of the client-Core communications are assured. Per the National Institute of Standards and Technology recommendations [42], the server is configured to only use TLS version 1.0, 1.1 or 1.2 since previous versions have known weaknesses. For the initial key exchanges perfect forward secrecy is desirable. Therefore, the server must use Ephemeral Diffie-Hellman key exchange (DHE) or Elliptic Curve Ephemeral Diffie-Hellman exchange (ECDHE). Furthermore, the uses cipher suite must only include those recommended in [42, pp. 14-19].

3.7.3 Core-Ground stations

As mentioned in section 3.5, each ground station has a profile which the Core server uses to determine the designed ground station and its capabilities. As such, and at least for this reason, these must be authenticated by the Core. Core-ground stations communication uses a message passing scheme over a long lasting session. As such, only one authentication at the start of the session is needed as it's valid throughout its life-time. The authentication process and considerations, i.e. credential storage, are similar to how users authenticate themselves with a but few differences. Firstly, note that the used channel is established directly over TLS, as seen in figure 3.15, since it's a long lasting session. Secondly, the Core renegotiates the TLS sessions parameters once the user is successfully authenticated. Furthermore, this renegotiation must be performed securely as per RFC 5746 [43].

This renegotiation is required because the data in the radio link is in transmitted in clear text. Therefore, the encryption keys (but not the integrity ones) of the TLS session could conceivably be broken. Note that during normal operation we only wish to assure the integrity of the session. However, as part of the authentication, the ground station sends it's secret to the Core trough this channel. Since the renegotiation forces a different set of keys to be used, before and after the secret is sent, it's thus secured.

3.7.4 Secure Session

The main scheme for securing instructions between the Ground and Space segment is based on Message Authentication Codes (MACs). Simply put, a Secure Session (SS) is established between the Core server and the COM subsystem. In this session, all critical instructions are appended a MAC using equation 3.1, which use a secret key, the message itself and few other parameters.

$$\textit{MAC}(K, N, s, d, sn, m) = \textit{HMAC-SHA256}\Big(K, (N \mid\mid s \mid\mid d \mid\mid sn \mid\mid m)\Big) \tag{3.1}$$

Where:

K	is the secret key,
N	is the nonce provided by the remote host,
s	is the address of the sending host,
d	is the address of the destination host,
sn	is the sequence number of the message,
m	is the message to be authenticated,
HMAC-SHA256	is the HMAC algorithm using the SHA256 crypto hash function,
||	denotes concatenation,

The MAC is created using the HMAC algorithm with the SHA256 crypto hash function. The HMAC [44] protects against data extension attacks, something that the naive approaches, $Hash(\ m||K\ )$ and $Hash(K\ ||\ m)$, would not. Note that the HMAC algorithm is independent of any particular hash function. Therefore any one can be used. From the recommended functions, the weakest and computationally cheapest hash function but still safe to use in HMAC operations is SHA-1 [45]. However, the 2015 document fails to address the recent (2017) SHA-1 collision by Google [46]. Therefore, we cautiously opted to use SHA256.

As shown in equation 3.1 the used HMAC function uses more than the message itself to compute the MAC. Indeed, each host stores two 32 byte nonces and two 16 bit integer Sequence Numbers (SNs). One *(nonce, SN)* pair for receiving messages and one for sending messages. The nonces, are exchanged in handshake phase as shown in figure 3.16. During the handshake phase, each host generates a nonce using a secure random number generator and send it to the remote host using a *SSHello* message. The nonce is then used to validate the MACs of messages received by the original host. This way, different sessions use different nonces and are thus resilient against inter-session replay attacks.

Once the session is established, the two sequence numbers maintained by each host are set to zero. From that point forward, whenever a host sends a critical message it uses the transmission SN for calculating the MAC and then increments it by one. The receiving host, then uses it's reception SN to validate the received message. If valid, it delivers it to the appropriate subsystem and increments the SN by one, otherwise discards the message. This way, intra-session replay-attacks are prevent. Note that the receiving host only accepts MACs with exactly the SN it's expecting. Therefore, if a message is lost in transit, the sender and receiver sequence numbers would become out of sync and prevent further messages from being validated. Fortunately, both *core-ground station* and *ground station-COM* channels, are established using TCP and AX25 connection-oriented. Thus, out-of-order, corrupt and lost messages should not occur as these events should be handled by these protocols. If and when the receiver sequence number reaches the value 65535, the maximum value of a 16 bit unsigned integer, all subsequently received messages discarded and returned with an error message. When it's the sender's SN reaches the aforementioned value, the host must reestablish the SS as no further messages are accepted.

For reasons explained below, the SS is established between the Core server and the COM, or C&DH

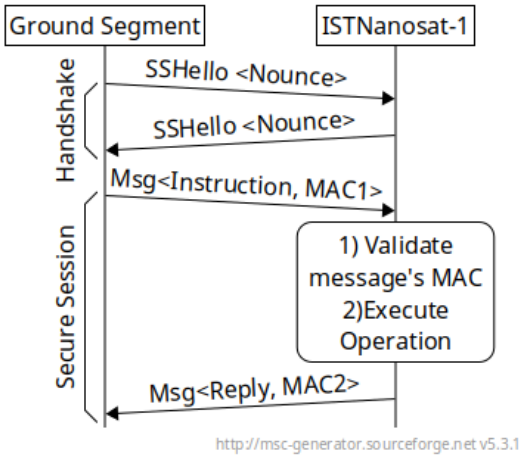


Figure 3.16: MSC of Handshake and Secure Session phases

when in safe mode. Therefore, had we not included the addresses of the end subsystems in the MAC calculation, a man-in-the-middle attack could be employed to redirect a critical messages to different subsystems.

In the space segment, the context of the SS is established and maintained between the Core server and the COM as shown in figure 3.15. In this approach, the COM is then responsible for validating received messages and inserting MACs to messages sent by the end subsystems on their behalf. The benefit being the 1) zero computational overhead for the end-subsystems as these tend to be computationally less powerful; and 2) the session needs to only to be established once, rather one per subsystem and thus faster to establish.

In the Ground Segment it's the Core who maintains the session, as opposed to the ground stations. This has the benefit of preventing ground stations from unilaterally sending critical messages. In fact, establishing the SS in the ground stations would mean that we would **have** to trust these and risk losing the secret key or design and implement a complex way for managing the SS.

Regarding the MAC's length, due to the limited throughput of the radio link, the size has to be carefully chosen. While it's true that critical messages are to be sent sporadically and thus don't have a large impact on the overall throughput. However, we may send several such messages in a short time span. Therefore, the size of MAC must be the smallest possible without compromising the SS. From this perspective two attacks were considered, specifically, an online and a offline brute-force attack.

In the offline attack's goal is to discover the secret key used to generate the MAC's by repeatedly testing keys until discovering the one used.

Using specialized hardware (used to mine Bitcoins[3]) to calculate SHA-256 message digests, an attacker could produce 14×10^{12} digests per second. Using a 128-bit key, to have a 50% probability of discovering the key would require:

$$(2^{128} \times 0.5)/(14 \times 10^{12} \times (3600 \times 24 \times 364) = 3.86 \times 10^{17} years.$$

In the online variant, the attacker tries to send commands with random MAC's to try and see if the

[3] List of SHA256 hash calculator hardware: `https://en.bitcoin.it/wiki/Mining_hardware_comparison`

satellite accepts one of these commands. Here, the low troughput of the link plays in our favor since it limits how many attempts per second are possible to perform.

Taking into account that the protocol stack is AX25, CSP and INCP. Each message has a length:

Headers	$(16 + 4 + 1) = 21$ bytes
Payload	16 bytes (assumed average for instructions)
MAC	8 bytes (64 bits)
Trailers	2 bytes
Total	$(21 + 16 + 8 + 2) = 47bytes = 376bits$

Assuming the worst case with a throughput of $48000bits/s$, the attacker can perform $\frac{48000}{376} = 128$ attempts per second. With a 4-month mission and a 64-bit MAC, $128 \times 3600 \times 24 \times 31 \times 4 = 1.37 \times 10^9$ attempts can be made. Thus, the probability of message being accepted by the satellite is $\frac{1.37 \times 10^9}{2^{64}} \times 100 = 6.33 \times 10^{-11}\%$. Note that this attack, assumes the 128 attempts per second of every second, of every minute, of every hour, of every day, of all 4 months that the satellite will be in orbit. Although this scenario is not possible in practice, the 64-bit hash is sufficient to withstand the attack. Therefore, there is no feasible attack (of this nature) that can be successful and as such we conclude that a 64 bit MAC is safe to use.

3.7.5 Persistent Message Authentication

The described SS works by establishing a session between the Core and the satellite. However, this means scheduling instructions to be sent to the satellite from a ground station is impossible. Therefore, we designed a second scheme denominated Persistent Message Authentication (PMA) which does not require a session to be established.

$$\textit{MAC}(K, s, d, sc, sn, m) = \textit{HMAC-SHA256}\Big(K, (\ s \mid\mid d \mid\mid sc \mid\mid sn \mid\mid m)\Big) \tag{3.2}$$

Where:

K	is the secret key,
s	is the address of the sending host,
d	is the address of the destination host,
sc	is the session counter of the message,
sn	is the sequence number of the message,
m	is the message to be authenticated,
HMAC-SHA256	is the HMAC algorithm using the SHA256 crypto hash function,
$\mid\mid$	denotes concatenation,

When the Core is to schedule a sequence of messages to be sent, their MACs as calculated using Equation 3.2. The differences between this scheme and the previous one, is the absence of nonces and the addition of the Session Counter (SC). This counter's value is persistently stored number by both the Core and the satellite. When a sequence of instructions is to be scheduled, these are assigned the

same Session Counter (SC) but sequential SNs starting from zero, as shown in Figure 3.17. Once the ground station relays the instructions, the satellite only accepts those which have a SC greater than the one it currently possesses. After the last instruction is sent and the connection closed, the satellite sets its SC to the value carried by the instructions.

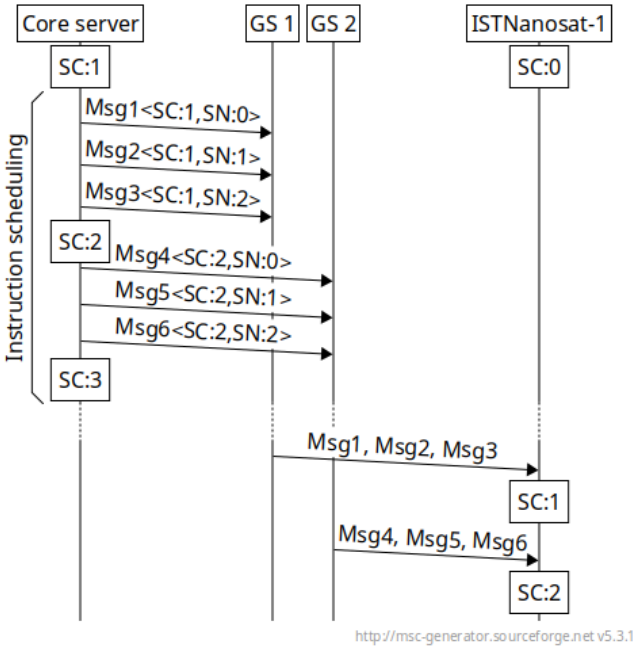


Figure 3.17: Message exchange using PMA

This way, multiple sequences of instructions can be scheduled on multiple ground stations. Additionally, even if one fails ground station the remaining can still deliver their respective instructions. When the core and the satellite do mange to communicate in real time these need to synchronize their SC since the last ground station with scheduled instructions may have failed to deliver them. In this case, the core issues them itself, and sets satellite's SC to be consistent with its own via a command for this purpose.

3.7.6 Closing remarks

Using this approach critical commands have some additional overhead. However, this is not crippling to the system's bandwidth because only critical messages need to be secured. In other words, data and telemetry retrieval commands, which should be make up the bulk of the used traffic, don't need to be secured and therefore, do not add any additional overhead.

On a final note, When in safe-mode the C&DH subsystem takes the role of the COM in terms of communications and will thus have to maintain the SS. In this mode the increased responsibilities may be impossible to support given its computational resources. However, this is still to be determined and we will know for sure once the C&DH software if fully implemented and proper analyze can be performed. If however, this reveals to be true, a possible alternative is to use some sort of challenge-response scheme at the start of the connection. Then, until the underlying AX25 connection is lost, critical messages could be transmitted and executed.

Chapter 4

Implementation

There are four major software components that were implemented. Specifically, the INCP implementation (i.e. libINCP), the ground station application, the Core server application and the User Interface. Be aware that these sections dot not directly map to the ones of chapter 3. However, before discussing the aforementioned components' implementations we detail the development environment used.

4.1 Development environment

For the development environment, there are two topics of interest. Firstly, an emulator was implemented to replace the satellite and is detailed in 4.1.1. Secondly, the actual topology of this environment is detailed in section 4.1.2.

4.1.1 Satellite emulator

Since the ISTNanosat-1 is an ongoing project involving multiple people, each working on different components and subsystems, we still do not have a physical satellite with which to test the Ground Segment. Even if we did, to perform all testing during development with it, would pose a serious logistical challenge due to the many people involved. Therefore, we implemented a emulated satellite in order to perform the development and testing of the Ground Segment.

Since without a space segment the ground segment serves no purpose, the emulator's value is obvious. However, to properly test the ground segment, the protocol stack must be the one shown in figure 3.13 with the INCP, CSP and AX25 protocols. Furthermore, the external behavior of the satellite must be one that, from the Ground Segment's perspective, is the same as the final satellite. Therefore, the emulator fully responses to the various services which the INCP enables, such as data and error log retrieval; diagnostic and command execution.

Regarding the data service, there was already a field test were some preliminary data was collected pertaining to the ADCS. This test was conducted by the Balua Project[1] team together With AMRAD, sponsor of the ISTNanosat project. The test consisted of on releasing an high altitude balloon carrying a prototype of the ADCS board in order to test its sensors. Specifically, periodic readings of the board's

sensors were made and stored. Since the resulting data is the closest thing to the end product that is available, the emulator will read the collected data from a CSV file at the same rate at which the measurements were made and thus imitate the prototype's test. The Ground Segment will then retrieve the data as expected.

For testing purposes the above data is enough to validate the retrieval of periodic variables. However, it does not allow for an exhaustive testing of the INCP and has such an additional mechanism was implemented. It involves a set of special commands, used only for the development and testing phases, which enable instructing the emulator to generate specific events. For instance, one command will instruct the satellite to generate an error with a specific ID and subsystem. Another, enables setting the result and duration of the next execution of a particular command or diagnostic.

4.1.2 Topology

The testing environment has to respect the topology previously described and shown in figure 3.1. Thus, the Core was set to run on a Virtual Private Server provided by CIIST, with a publicly exposed ip address. Running the ground station software, is a computer on satellite tracking room in the Taguspark campus, which will also serve as the main ground station of the ISTNanosat-1 project. The emulated satellite in running on a Raspberry-PI3 connected to the ground station via serial port. Note that on the final setup, the ground station will instead be connected to a radio. Since the Core has a public ip any one can connect to it and serve as a user.

For development purposes, the ability to run the entire setup in same machine is rather valuable. To this end, running the user, Core and ground station software is trivial since communications between these use IP. The issue lies in the ground station-satellite communications since AX25 expects a underlying serial port. Therefore, we use the *socat* utility[2] to create a virtual serial port which is then associated with a AX25 callsign on either end, one for the satellite and one for the ground station. This way, the above described environment can be recreated in a single machine. Additionally, a simple Python script was written which serves as a hub were multiple virtual ports can connect. This way, we can execute multiple ground stations on the same machine, exemplified in Figure 4.1, which is particularly useful for testing the core-ground station control mechanism detailed in section 3.5.2.

[1] The test in question: `http://balua.org/1855-2/` - Accessed on 25-04-2017
[2] Socat man page: `https://linux.die.net/man/1/socat`

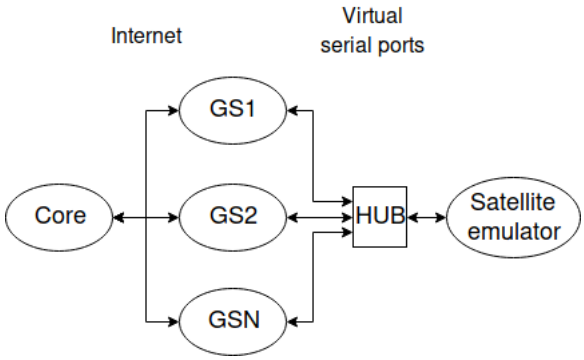


Figure 4.1: Development Environment topology

4.2 ISTNanosat-1 Command Protocol

In this section we detail how the ISTnanosat Control Protocol (INCP) was implemented, which decisions were taken and why.

First, we had to decide which programming language was to be used. The simplest option would be Python as the protocol is to be used by the Core server. However, we opted for using C after taking into consideration the overall ISTNanosat-1 project's context. Firstly, a C implementation allows for a single implementation for both the ground segment and for the ISTNanosat-1's subsystems, thus reducing the code surface area and consequently the likelihood of bugs. Secondly, implementation effort is reduced for the entire team at the cost of we having to 1) implement the protocol in a lower level language and 2) having to handle the binding between the C implementation and the Core server written in Python. Thirdly, and if the reader recalls from section 3.4, the INCP requires some information about the *IDs* and data types of the telemetry and instructions for its normal functioning. By having a single lib, we can also "codify" the meta-data about the variables, errors, commands input/output arguments and diagnostics results. Therefore, a single source of truth exists across both the ground and space segments. For the subsystems, this implementation is provided as a collection of source and header files which can then be used in the compilation of each subsystem software. For the Core, this implementation is compiled into a dynamic library (i.e. .so file) which can then be used from the Core's Python code. The mechanisms for binding the two languages is detailed in section 4.2.5.

A consequence of having a shared lib which is that it must run both in the ISTNanosat-1's subsystems, i.e. an embedded system running FreeRTOS, and in the Core, i.e. Linux. Therefore, we must also respect the computation constraints of the lowest denominator, in this case the subsystems and the two very distinct operating systems. Furthermore, NASA's guidelines for safety critical software [49] must be taken into consideration.

Unlike other protocol libraries, the libINCP only handles the message construction, serialization and unserialization (in addition to a few other miscellaneous tasks). This design greatly simplifies the issue about portability since it makes smaller assumptions about the environment it will be running in. In particular, the implementation is not dependent upon the expected protocol stack or how to use it. However, this means that the logic for how to handle a message exchange is not included. From the subsystems perspective this is not a problem. Recall that these simply respond to incoming requests,

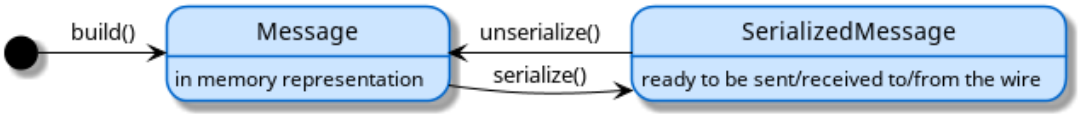


Figure 4.2: Message state transitions.

Thus, their message handling logic can (mostly) be reduced to a large switch case statement. For the Core server this logic is then implemented in Python and detailed in section 4.4.2.

The major operations which this implementation must achieve are thus message construction, serialization and unserialization, as shown in figure 4.2. As indicated in [49], memory allocations can not occur after the initialization phase of the application. We thus use a *builder_t* structure, which is little more than a bound-checking buffer allocated during the initialization phase. It enables appending data (in byte arrays form) tracking it's used and maximum lengths such that buffer overflows do not occur. All messages building and (un)serialization thus occur with the messages structures being stored inside one such *builder_t*. The produced lib is thus divided into two major sections, one for the actual messages and one introspection meta-data required for describing the data variables, errors, commands input/output arguments and diagnostic result format, as detailed in section 3.4 and shown in Figure 4.3.

All messages have a header, which contains the opcode, and a set of operations which it must support shown as shown in figure 4.2. Message unserialization requires knowledge of the data contained in the message (i.e. the introspection mentioned above). Information which is contained in a set of *subsystem_t* structures (one per subsystem). These contain variables description and *stable_t*s (as in Serialization Table) for the messages which have parameters such as commands.

In the following sections we address the three kinds of messages and how each is implemented. Specifically, messages with static fields, represented in figure 4.3 by *reqData*, the *Data* message and messages which contain a specified set of arguments, represented by *function* in the figure.

4.2.1 Messages with static fields

This type of messages only have simple and constant fields and are thus the simplest in terms of (un)serialization. These, represented by *reqData* message in figure 4.3, include all messages which do not have a body_t field and the *Data* message. Serialization is straight forward, involving only calling the appropriate serialization function for each of it's components. For example, integers have to be placed in network byte-order (Big-endian). In *reqData*'s case that would be the header, *id* and *ts* fields. Unserialization involves simply calling each field type's unserialization function on the raw message by the same order.

4.2.2 *Data* messages

The challenge of *Data* message implementation is that this type of message has a varying number of data fields, each with a varying type and size, where the periodic kind has a varying number of values. See figure 3.11 for a visual representation.

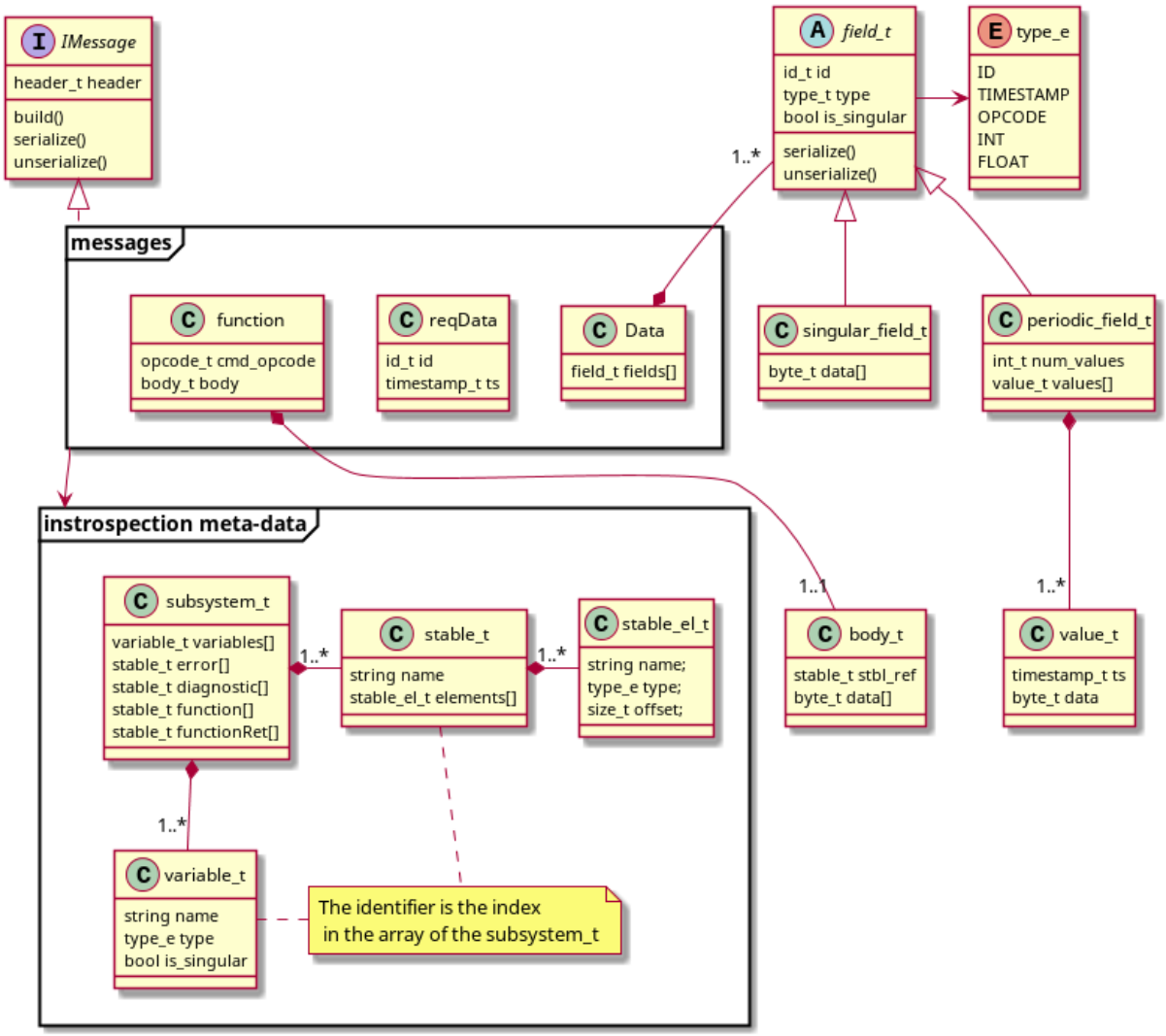


Figure 4.3: INCP message class diagram. Note that C does not have interfaces or classes. Therefore, when looking at the diagram think of the concepts and not the actual construct.

The implementation of this message consists of a *msg_data_t* structure which contains an array of *field_t*, which has three important members. Namely, its id, type and a boolean indicating if it's a singular or periodic field. This last flag is crucial because what is actually inside the *field_t* array is in fact, either a *singular_field_t* or a *periodic_field_t* structure. Thus, when want to perform an operation on a *field_t*, we check the flag, cast the *field_t* into the appropriate singular or periodic structure and then apply the operation. Note however, that each operation (such as serialization and unserialization) must have two function variations, one for each of the two field types.

The C construct that enables this behavior is exemplified in listing 4.1. In it, notice how in line 8 the *singular_field_t* contains a *field_t* inside it. Since in C the order of the members of a structure declaration are respected during compilation we can cast a *singular_field_t* in to a *field_t* structure and seamlessly perform operations meant this latter type.

```
1  typedef struct {
2    id_t id;
3    type_t type;
4    bool is_singular;
5  } field_t;
6
7  typedef struct {
8    field_t base;
9    byte data[];
10 } singular_field_t;
```

Listing 4.1: *field_t* and *singular_field_t* declaration

Note however, that since each field has a different field with a varying size, the normal way of accessing it (i.e. array[index]) does not work. Thus, to access an individual field we have to iterate over previous fields in order to reach on desired one. However, on all four important operations (i.e. building, serializing, unserializing and reading the values) we always want to iterator over all fields meaning this limitation is of not consequence. Additionally, we implemented a an iterator construct, which allows for trivial iteration over all the fields of the message.

Regarding how the actual data is stored in the fields, notice the *data* member of the *singular_field_t* and *value_t* structures. These contain a byte array of size equal to the size of the type specified in the *field_t*'s *type* member. Unfortunately, this strategy raises the issue of how to guarantee the memory alignment[3] of the fields and values' data.

One possible solution to this problem, consists in tracking the alignment ourselves (in dynamic fashion) and inserting padding when needed. However, this padding is wasted space, something which does not bode well for embedded systems. Instead, we compact the data of *singular_field_t'* data and *value_t's* values with no regard for alignment. Then, we implemented a set of functions (one for each possible type shown in *type_e*) that given a reference to this possibly unaligned data, it copies it (via *memmove()*) into an internal variable and then returns the value in this variable. Therefore, reading the data inside the *Data* message's struct is always done using this set of functions.

[3] Wikipedia article regarding data structure alignment: `https://en.wikipedia.org/wiki/Data_structure_alignment`

Building

After the message (i.e. *data_msg_t* structure) is initialized, we can insert the data. This insertion is perform via *add_singular()* and *add_periodic()* functions which are continuously called until either all data is inserted or the maximum message size is reached (i.e. the MTU of the protocol stack). Note that this maximum limit is checked by the aforementioned functions by tracking the size of the would-be serialized message before and after each insertion.

(Un)Serialization

We must first recall that the underlying protocol stack is datagram oriented. Therefore, we know the size, in bytes, of each received message. Thus, the size of the payload (and contained message) can then be determined by subtracting the size of the INCP header. Now that the size of the payload can be determined, we need to decode each of the data fields. However, due to their varying size, at first, we can only know the position of the first data field. Therefore, the payload can be decoded by continuously decoding a field and determining the positing of next one. This process is repeated until the entire message is unserialized.

Unserializing a data field is simple when dealing with a singular data. The size of the data value can be determined by checking the associated size of the type with the given *ID* for the sender subsystem in question, see Figure 4.3. If however, we're dealing with a period data field, the procedure is similar but instead of decoding one data value we have to decode an array of *(data, timestamp)* tuples. But by knowing the id and the size of the array, decoding the data field is straight-forward.

The mapping of *(subsystem, id)* is done via a static and constant table which contains a list of all variables of all subsystems, shown in figure 4.3. By looking up a variable (i.e. this lookup returns a reference to a *variable_t*) we determine it's type and if it's singular or periodic. Since, by knowing the type we know it's size, this table is thus essential for the unserialization of data fields and, consequentially, of *Data* messages.

Knowing the procedure for serialization of data messages, the serialization is then easy to deduce.

4.2.3 *function* message

As detailed in section 3.4 there are multiple messages which have a body (with a set of fields) defined outside of the INCP. This, raises the issue of how to implement the libINCP and these messages in way that is both simple and avoids repetition of work between the subsystems and the Ground Segment.

Design considerations

Ideally, we would want to defined a single plain C structure, included in libICNP and thus declared only once. Each member of the structure is then, for example, a specified input argument of a command. Then we pass a reference to this structure to libINCP which auto-magically creates a serialized message

(in this case a *function* message) ready to be sent to the wire. Finally, and on the receiving host, a single unserialization function call results in a ready to use structure with the original values.

The ISTNanosat-1 project is still under development, thus, the set of telemetry and instructions are bound to change over time. Therefore, using plain C structures to describe the bodies this kind of messages greatly reduces development costs, since these are can easily be created and/or changed. Furthermore, using plain C structures the compiler can type-check their usage, therefore:

- If a new field is added to an existing structure will cause a compile time error if the code sections which depend on the structure are also changed. Assuming of course, a compilation with the appropriate set of flags, where use-before-initialization results in an error.
- Changing the type of a member of the structure also results in a compilation error if the new type is inconsistent with the use cases.
- Finally, removal of a member from the declaration will, of course, result in a compilation error.

Implemented Solution

In order to create a mechanism described above, the libINCP needs to know each of the *function* message body's fields and their types for each possible command. Then, knowing this information we need a way to retrieve the values of the original structure as to (un)serialize them. To accomplish this task, we implemented a solution which centers around the usage of serialization tables (i.e. *stable_t* in Figure 4.3). This structures are used to describe a plain C structure in such a way as to allow for introspection. Lets consider the example of a command add2 which has 2 input arguments, both 32 bit integers. For every INCP *function* message of every subsystem we declare a plain C structure with all the arguments. In the case of the example command "add2" see *cmd_add2_t* in line 1 of listing 4.2.

```
1  typedef struct { int32_t a; int32_t b; } cmd_add2_t;
2
3  stable_el_t cmd_add2_stbl_el[] = {
4      STBL_EL(cmd_add2_t, a, "param a"),
5      STBL_EL(cmd_add2_t, b, "param b"),
6  }
7
8  // inside the function[] array of subsystem_t
9  stable_t function[] = {
10   // (...)
11   {"Add2", sizeof(cmd_add2_t), &cmd_add2_stbl_el},
12   // (...)
13 }
```

Listing 4.2: Add2 serialization table

Given the *cmd_add2_t* structure we need to fill the serialization table which describes the message. In lines 3-6 of the example, each member of the original structure is given an entry using the *STBL_EL()* macro explained below. Then, a reference to this array is inserted in the final *stable_t* structure seen

in line 11 of Listing 4.2. This last structure, is in turn inserted in a array of *stable_t* of the *subsystem_t* pertaining to the subsystem which implements the command, as shown in Figure 4.3.

The libICNP needs two critical pieces of information for member of the original *cmd_add2_t*, the type of the member (i.e. the incp_type_e) and it's offset. This offset is the space in bytes between the start of the structure, in this case *cmd_add2_t*, and that particular member. For instance, the off of the member *a* is zero, as it's the first member. But the offset of *b* is 4. Note that in lines 4 and 5 an additional description of the member is given in the form of a string, but its only used for debugging purposes. The macro itself can be seen in Listing 4.3 and the expanded result of using it for member b of *cmd_add2_t* is shown in Listing 4.4.

```
1  #define STBL_EL(_struct, _member, _description) \
2    {                                             \
3      INCP_TYPE_GENERIC(_struct, _member),        \
4      offsetof(_struct, _member),                 \
5      #_member,                                   \
6      _description                                \
7    }
8
9  #define INCP_TYPE_GENERIC(_struct, _member)     \
10   _Generic((((_struct *)0)->_member),           \
11     float:        INCP_TYPE_FLOAT,              \
12     int8_t:       INCP_TYPE_I8,                 \
13     int16_t:      INCP_TYPE_I16,                \
14 // (...)
```

Listing 4.3: STBL_EL() and INCP_TYPE_GENERIC() macros

In line 3 of the listing, we use the *INCP_TYPE_GENERIC()* macro to determine the *incp_type_e* based on the type of *_member* of structure with name *_struct*. Internally, and as shown in lines 10-15, the builtin *_Generic*[4] is used. This builtin provides a way to choose one of several expressions at compile time (lines 12-15), based on a type of a controlling expression (line 11).

The offset is determined by the *offsetof()*[5] macro as seen in line 4 of Listing 4.3. Then the expression in line 5 simply stores the name of the member as a string (the expression expands to line 4 of Listing 4.4).

```
1  {
2    INCP_TYPE_INT32,
3    4,
4    "b",
5    "param b"
6  }
```

Listing 4.4: STBL_EL() expanded example

[4] *_Generic* explanation: `http://en.cppreference.com/w/c/language/generic`
[5] Explanation of the 'offsetof' macro: `https://en.wikipedia.org/wiki/Offsetof`

Building

As can be seen in listing 4.5 building the message is performed via *msg_function_build()*, which needs the opcode and a reference to the structure described by the serialization table. Then serialization of the body is easily performed as shown in line 4 of the example.

(Un)Serialization

For each *stable_el_t* we serialize and insert the corresponding member of *cmd_add2_t* into the serialized message body. This serialization and copy is only possible because we know it's type via the *stable_t*. This table, is in turn, determined by knowing the subsystem which produced the message and opcode of them message (if command) or it's identifier (if *diagnoticResult* or *ErrorVerbose* messages). When unserializing a message, a similar process is applied. The example of listing 4.6 is only possible if we knew that the serialized message was a *function* message of the add2 command. In practice however, there is a single msg_unserialize() function which returns the type of the message (i.e. if it's a *reqData, function*, *functionRet*, etc). Knowing the message type, the action to perform is determined checking it's static fields.

```
1 void before_send(builder_t *builder, builder_t *builder_ser) {
2   cmd_add2_t cmd_add2 = { 1, 2 };
3   msg_function_build(builder, OPCODE_CMD_ADD2, &cmd_add2};
4   msg_function_serialize((msg_function_t *) builder->buffer, builder_ser);
5   // ready to send
6 }
```

Listing 4.5: Add2 serialize example

```
1 void after_recv(void *msg, sise_t size, csp_addr_t sender_addr, builder_t *builder) {
2   cmd_add2_t cmd_add2;
3   msg_function_unserialize(builder, sender_addr, msg, size, &cmd_add2};
4   printf("params a:%d b:%d\n", cmd_add2.a, cmd_add2.b);
5   // params a:1 b:2
6 }
```

Listing 4.6: Add2 unserialize example

Using this approach means that we will have to maintain the serialization tables consistent with structures they describe. Unfortunately, there is no compile time mechanism to check this consistency. However, a run-time check can be devised and are detailed in section 5.2

4.2.4 Security

The libICNP also implements the necessary cartographic algorithms and logic for supporting the SS, detailed in subsection 3.7.4, and Persistent Message Authentication (PMA), detailed in subsection 3.7.5. Simply put, we provide the HMAC and SHA256 algorithms, and the necessary structure for the server (i.e. the satellite) logic of the SS in a *ss_ctx_t* structure, which maintains the state of the session. This

structure contains the key, sequence number and nonces and has two main accompanying functions. One for inserting a MAC to a diagnostic or command message yet to be sent, and one for validating the MAC of a received message. Note that the we still need a secure random number generator for the satellite. However, at this stage this is not required, because the Core server can leverage the Linux's *urandom()* system call for just that. The PMA is similarly implemented in a *pma_ctx_t* structure but no assumptions are made about how the SC is to be persistently stored.

4.2.5 Python bindings

For the reasons detailed above, the libICNP is written in C. However, the Core server which depends upon this library is written in Python. Therefore, some sort of mechanism for binding the two language was needed. In order to test the possible alternatives for doing this binding a small C code snippet was written with some base constructs (e.g as functions, macros and structures), shown in listing 4.7.

```
1  /* foo.h */
2  #define CONST 3
3  #define INC1(_a) (_a + 1)
4  extern int somevar;
5  int add(int a, int b);
6  float add_float(float a, float b);
7  typedef struct { int a; float b; } some_struct_t;
8
9  /* foo.c */
10 #include "foo.h"
11 int somevar = 1;
12 int add(int a, int b) { return a + b; }
13 float add_float(float a, float b) { return a + b; }
```

Listing 4.7: Test C code

The first considered solution is the *cypes*[6] module from Python's standard lib. Ctypes is a foreign function library for Python. It provides C compatible data types, and allows calling functions in shared libraries (e.g .dll or .so files). It can be used to wrap these libraries in pure Python code without any compile time build requirements. However, since ctypes does not parse the header files it lacks some information about the underlying .so file. Firstly, function input types must be specified or risk silent failures as seen in lines 7-9 of listing 4.8. Secondly, if the return type is not specified the result is cast into an int, as shown in lines 11-13 of the example. Thus, if the return value is to be used, it must first be specified by hand. Thirdly, using this approach means we cannot access and use macros and defines, as shown in the lines 17-18. Finally, any structure which is not a base type (i.e. int, short, float, double, etc) must be declared manually, as shown in lines 20-23 and kept consistent with the C counterpart.

```
1  # load the library
2  libfoo = CDLL("libfoo.so")
3
4  add = libfoo.add
```

[6] Ctypes documentation `docs.python.org/3/library/ctypes.html`

```
5  add_float = libfoo.add_float
6
7  add(1, 2)  # returns 3
8  add(1, "3")  # returns garbage as it casts the string pointer to an int
9  add(1, 2.3)  # raises a ctypes.ArgumentError exception
10
11 add_float(c_float(1.1), c_float(2.2)) # returns 2, of type ctypes.c_int
12 add_float.restype = c_float # but if we specify the return type
13 add_float(c_float(1.1), c_float(2.2)) # returns 3.3000001907348633, of type ctypes.c_float
14
15 libfoo.somevar += 1  # somevar now has value 3
16
17 libfoo.CONST  # ERROR: undefined symbol
18 libfoo.INC1(1)  # ERROR: undefined symbol
19
20 # some_struct needs to be redeclared before it can be used
21 class some_struct(Structure):
22     _fields_ = [("a", c_int),
23                 ("b", c_float)]
24
25 s = some_struct(1, c_float(3.0))
26 s.a  # returns 1
27 s.b  # returns 3.0
```

Listing 4.8: Ctypes test code

The second considered alternative was SWIG[7]. SWIG (Simplified Wrapper and Interface Generator) is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. It's typically used to parse C/C++ interfaces and generates the "glue code" (i.e. the bindings) required for the target languages to call into the C/C++ code.

Like with ctypes, we tested swig on the C code of listing 4.7. Since SWIG generates wrapper code, an additional compile time step is required. The configuration of this wrapper generation step took some effort due the somewhat confusing documentation.

Once wrapper code was generated, we could then proceed with the test from which we extracted several conclusions. The first, it that swig provides a runtime type check of the arguments and return value of wrapped functions, as shown in lines 7-11 of listing 4.9. Furthermore, SWIG can handle macros without arguments (i.e. regular #defines statements) since it processes the header files, as shown in line 15. Unfortunately, macros with arguments are still unsupported, as shown in line 16. Finally, SWIG crates wrappers for all types it processed from header files creating Python classes, as seen in lines 18-21 of the example.

```
1 # load the wrapped library which internally loads libfoo.so
2 import libfoo
3
4 add = libfoo.add
5 add_float = libfoo.add_float
```

[7] SWIG Homepage: `www.swig.org`

```
6
7  add(1, 2)    # returns 3
8  add(1, "3")  # raises TypeError: in method 'add', argument 2 of type 'int'
9  add(1, 2.3)  # raises TypeError: in method 'add', argument 2 of type 'int'
10
11 add_float(1.1, 2.2) # returns 3.3000001907348633
12
13 libfoo.somevar += 1  # somevar now has value 3
14
15 libfoo.CONST  # returns 3
16 libfoo.INC1(1)  # ERROR: undefined symbol
17
18 # SWIG creates a wrapper for some_struct_t
19 s = libfoo.some_struct_t(1, 3.0)
20 s.a  # returns 1
21 s.b  # returns 3.0
```

Listing 4.9: SWIG test code

After comparing the two alternatives we concluded that SWIG has a superior type checking mechanism which does not require manual intervention, unlike in ctypes. Furthermore, it creates wrapper classes automatically and allows access to defined constants of the header files. Therefore, SWIG appeared to be the solution which requires the least overall effort with the best set of functionalities. Unfortunately, it the up-front cost of configuring the wrapper generation. However, this was done once and we expect not having to touch it again.

Once opted for using SWIG we had some final issues to address:

Firstly, SWIG does not support casting between two wrapped types which in C can be done inline, line 1 and 2 of listing 4.10. Therefore some functions had to be implemented for handling this limitation. These are essentially variations of the *float2double()* function, shown in line 4 of the example.

```
1 float f = 1.0
2 double d = (double) f; // inline cast
3
4 double float2double(float f){ return (double) f; }
```

Listing 4.10: Casting

Secondly, SWIG expect to be given a set list of header files were only these contain C code to be wrapped. Were the expect behavior is to update this list whenever a header file is created or deleted. To automate this, a bash script was written which searches for all headers in a provided directory hierarchy and produces a .i file (SWIG interface file) which is then used by SWIG for generating the Python bindings.

4.3 Ground Station

In this section, we addres the implementation of the ground station application. In particular, we detail their functioning in subsection 4.3.1. Additionally, the used protocols and their implementing libraries are

addressed in subsections 4.3.2, 4.3.3 and 4.3.4.

4.3.1 Implemented Application

As shown in Figure 4.4, the ground station is essentially an event driven application, were the events are received messages. Since the messages can be received from two different hosts, the Core server and the Satellite, we opted to divide the program into three threads namely, the Main Loop, the Core and Sat proxies. The main thread (i.e. Main Loop) is were the business logic is handled, as shown in figure 4.6.

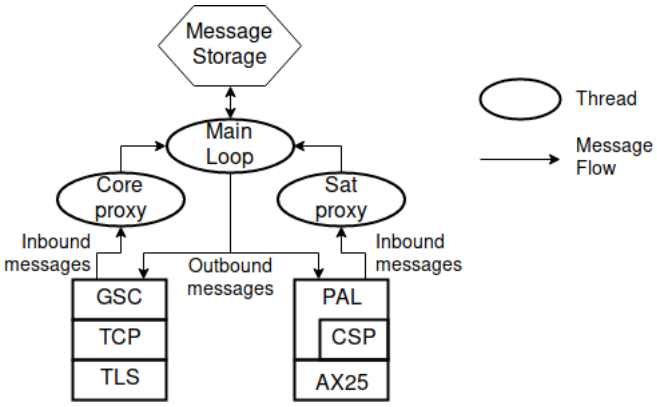


Figure 4.4: Ground Station Software Architecture

The core proxy, is responsible for establishing and maintaining the connection with the Core server. Particularly, it follows the flowchart of Figure 4.5. It first authenticates the ground station, and then sends any stored messages (i.e. beacons and periodically sent data) it received while unconnected with the Core. Then, and while connected, it's blocked waiting for messages to be received which are then placed in a thread safe queue (from where the Main loop will retrieve them).

The satellite proxy, like the core proxy, unserializes and places any received message into the aforementioned message queue. However, it does not establish connections with the satellite on its own. Instead, the *Main Loop* thread establishes the connection, and only then, does the proxy block itself waiting for messages. The beacons however, which are constantly sent by the satellite, are always received and place in the message queue. The *Main Loop* then retrieves these messages, and either stores them or sends them to the Core.

The *Main Loop* thread, is the one that actually performs the operations expressed in Figure 4.6. On the *wait for message*, it's blocked waiting for messages to be inserted into the thread safe queue, which are placed there by the two proxies. The messages in question are thus:

Hello Sent by ground station to identify and authenticate itself towards the Core.

INCP Carries a INCP message and must specify to/from which subsystem the message is to be sent or was received from.

Connect Sent by the Core instructing the ground station to connect to the satellite in CSP/AX25 mode, AX25 only mode or terminate the current connection.

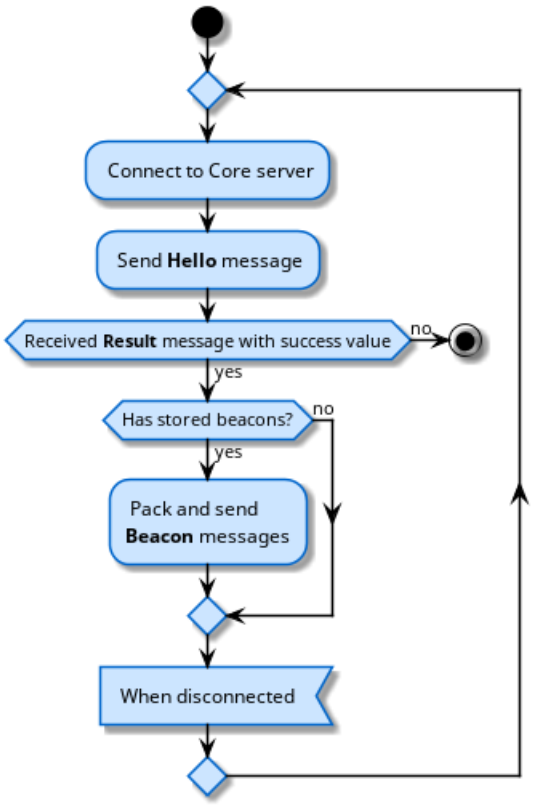


Figure 4.5: Ground station core connection flowchart.

Result Used for 1) acknowledging that the Ground Station successfully sent a INCP message (i.e. the system call send() was successfully) or if it failed and the connection was lost; And 2) the core sends it to the ground station to indicate that the secret used to authenticate it is valid.

Beacon Transmits a beacon message to the Core.

Using the aforementioned messages, the main loop execution flow can be condensed in the flowchart shown in figure 4.6. Simply put, the ground station forwards the various messages shown to/from the satellite. The *Connect* messages cause the ground station to (dis)connect to the satellite in the appropriate mode. Once connected, it sends to satellite any and all stored INCP messages which correspond to scheduled diagnostics and commands.

Regarding the used programming language, since the ground station mostly handles low level operations, thus we opted for using C. However, we use a C++ compiler since it allows for the usage of a library for handling JSON later described. Note, that this is possible since C is a subset of C++. The vast majority of the application however, in written in plain C since it facilitates it's maintainability as it's a more widely known language.

4.3.2 Communication Protocols

The ground station uses two different protocol stacks, one for communications with the Core server and the other for the satellite.

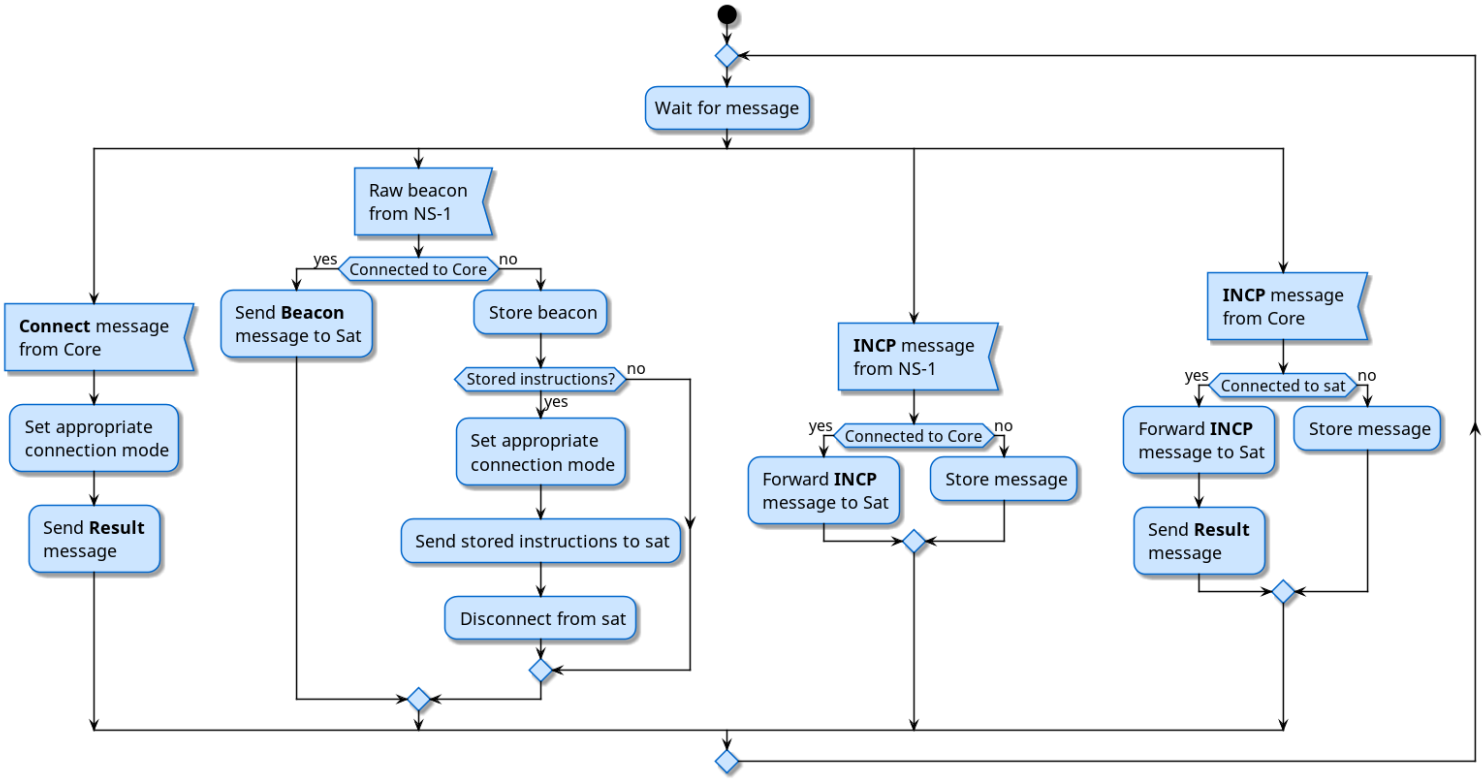


Figure 4.6: Ground station execution flowchart.

In the core-ground station stack, we opted for implementing the GSC message in a JSON based format for several reasons. Firstly, the Core-ground station link is not very restrictive. Thus, a binary format which would save bandwidth is not essential. Furthermore, the introduced overhead of using JSON should have no impact since the bottleneck is the ground-space link with a $48000Kb/sec$ bit-rate in the best case scenario. Secondly, JSON is trivial to use in Python with it's standard lib, i.e. in the Core server, and relatively easy in C, i.e. in the ground station, using the appropriate library[1]. For the TLS implementation we chose to use OpenSSL[2] since it's a well tested and widely available library.

The satellite-ground station protocol stack has two protocols. Specifically, the AX25 protocol, provided by the Linux's libAX25[3], and the CSP, provided by a libCSP[4], modified in order to support AX25 as detailed in section 4.3.3.

However, these two protocols are not interacted with directly. Instead, the Protocol Abstraction Layer (PAL) serves as an adapter between the protocols' libraries and the rest of the application. This adapter serves three major goals:

Firstly, since libCSP is designed to be used in a both Linux and FreeRTOS it does not follow the Posix Socket design and API. For example, it uses an internal buffer for storing packets and thus avoids memory allocations during run-time, as is best practice for embedded systems. Furthermore, it uses both its own API for creating packets and *csp_send()/csp_recv()* functions, rather than the Linux's *send()/recv()* system calls. The PAL then provides a single API which hides both protocol's APIs.

[1]JSONCPP Homepage: `https://github.com/open-source-parsers/jsoncpp`
[2]OpenSSL Homepage: `https://www.openssl.org/`
[3]libAX25 Homepage: `http://www.linux-ax25.org/wiki/Libax25`
[4]libCSP Homepage: `https://www.libcsp.org`

Secondly, and as detailed in section 4.3.3, the used stack is different depending on whether the satellite is in safe or normal mode. Thus, the *pal_send()/pal_recv()* calls to the appropriate protocol by maintaining a state machine for just this purpose. Furthermore, whenever a state transition occurs, the PAL handles the necessary configurations of each protocol.

Finally, the libAX25 implementation as a few limitations when dealing with connections, detailed in section 4.3.3. Thus, the implemented solution to this issues is also handled by the PAL.

4.3.3 libAX25

Once we started developing the ground station application we observed an unexpected behavior pertaining to AX25 sockets. Specifically, each socket appears to have a lock that does not support two actions to be executed on the same socket at the same time instant. For example, if a thread is blocked in a *recv()* call and a second thread calls *send()*, then the second call will block without performing the desired action until the first thread exits the *recv()*. This is particular bad because on the used approach the *Main Loop* may try to send a message to the satellite while the *Sat Proxy* thread is blocked waiting for a message.

To work around this problem, the PAL first sets the socket in non-blocking mode. As the name implies, whenever an operation is applied to the socket, such as *send()* and *recv()*, that would normally block, a special error code is return instead. This way, *send()* calls are no longer prevented from executing by simultaneous *recv()* calls. However, we still a way of knowing when the socket has data ready to be read. One possible solution would be to periodically call *recv()* and process the returned data if any. However, this would introduce a delay between when a message is received and when it's processed. Which is particularly bad considering the small ground-space connection window. To attenuate the delay we could increase the check frequency. However, that would mean a waste of CPU time as most checks would indicate that no data is ready to be read. Instead, we opted for using the Linux's EPOLL API[12].

Simply put, given a list of file descriptors, such as sockets, the *epoll_wait()* system call blocks waiting for configurable set of events to occur on the given file descriptors. This system call can be further configured to work in either level-triggered or edge-triggered modes. In level-triggered mode, *epoll_wait()* returns whenever there is, for instance, data to be read on the provided file descriptor. In edge-triggered mode, the caller is only notified whenever the file descriptor transitions from having no data into having data to be read. The most expedient option appeared to be level-triggered mode. Thus, whenever a *pal_send()* call is made, if first blocks on the *epoll_wait()* system and only when it returns indicating that the AX25 socket has data to be read is the *recv()* system call performed.

4.3.4 libCSP

CSP is a network/transport layer protocol and is implemented by the libCSP library. The library also includes a UDP and RDP transport protocol implementation. Furthermore, the several interfaces are

[12]EPOLL manual page: `http://man7.org/linux/man-pages/man7/epoll.7.html`

implemented such that CSP can be used over link layer protocols. The default being, I^2C, CAN, KISS and loop-back interfaces.

Each CSP interface instantiates a *csp_interface_t* structure which has two main fields: 1) the *next-hop* field which stores a pointer to the function to be called when a CSP packet is to be sent to trough the interface; And 2) the MTU of the interface which limits the packet's size. Each interface should also have a RX thread blocked waiting for frames to be received. Once one is received, the CSP packet must be extracted from the received frame and delivered to the libCSP's routing mechanism via the *csp_new_packet()* function.

The goal was then to create a new interface such that CSP can be used over AX25, specifically in connection oriented mode. We started with the implemented proof-of-concept by João Ferreira in the context of [50]. Being a proof-of-concept, there several limitations which needed to be addressed.

Firstly, once started, the AX25 interface could not be stopped. Therefore, the underlying AX25 socket which had the host's AX25 callsign bound to it, would prevent any other AX25 socket from using the callsign. Thus, switching between CSP/AX25 and AX25-only stacks was impossible. Now, when the libCSP's AX25 interface is to be stopped, the *csp_ax25_stop()* closes the socket and joins the RX thread.

Secondly, the used callsigns were hard-coded which prevent the usage of multiple ground stations. The implemented solution, then contains a table which maps CSP addresses to AX25 callsigns. This way, when a ground station or the satellite emulator are first initialized, these set the appropriate address-callsign map of the various subsystems to the satellite's callsign. Furthermore, the satellite emulator sets the ground station's CSP-callsign map to the callsign of the most recently connected ground station.

Thirdly, the AX25 protocol was initially to be used with UI frames in a connectionless mode. But for the reasons detailed in section 3.4.2, we instead opted for using the connection-oriented mode. Similarly, the starting implementation functioned with UI frames. In order to support connections in the CSP-AX25 interface, we needed way for establishing connections and handle connection termination by the remote host. Unfortunately, CSP does not provide a standard way for handling connections in the interfaces.

The first step to the CSP-AX25 interface starts with a call to *csp_ax25_start_co()* which must be provided with a socket with an already established connection. Internally, it creates a new RX thread for the CSP-AX25 interface which is only joined when the host calls *csp_ax25_stop()*. We specifically opted not to have any handling of connection accepting in the libCSP's codebase, as it would greatly increase complexity for no tangible benefit.

The next step was implementing a mechanism for notifying the higher layers when the AX25 connection is terminated by a remote host. This problem was solved by having the RX thread first allocate a CSP packet with size 0, only has the header, and set the size field to $MTU + 1$. Thus, when *csp_recvfrom()* is called, it returns a packed with such size and the caller knows that the AX25 connection was terminated.

4.4 Core

The Core server overall architecture can be seen in figure Figure 3.2. As mentioned before, the Core is implemented in Python. Several choices mentioned in this section were only possible due to this choice. Particularly, due to Pythons's library ecosystem and intrinsic capabilities as a high-level language.

In order to reduce implementation and maintenance costs, the Core dynamically adapts itself to the libINCP, as shown in Figure 4.7. In particular, part of the database schema is generated by introspection of the library, and is detailed in subsection 4.4.3. Furthermore, a INCP.json file is generated and is used by the UI to create parts of the interface, and detailed in section 4.5. Using this interface, the UI can then request data from the services via the RESTfull interface, which is stored in the database, or executed functionalities implemented by the services such as commands, as detailed in subsection 4.4.2. Finally, and in order for the services to accomplish their tasks, the ground stations need to be commanded and controlled, a process which is discussed in subsection 4.4.1.

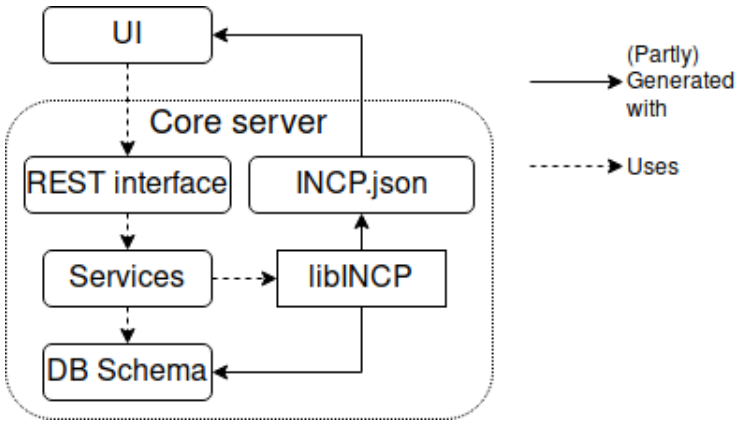


Figure 4.7: Creation and dependency graph

4.4.1 Core to ground station communications

The Core side ground station communications are assured by the three components of the *Ground Station Handling* block of Figure 4.8 which corresponds to the *Ground Station Manager* of Figure 3.2. Simply put, the *Listener* waits for new ground stations to connect. Once one does, it creates a new *Proxy* instance which serves as a wrapper for each ground station. The *Coordinator* it what handles the multiplexing of the Core-satellite communications based on events (e.g. new proxy or received message). Events which are placed in a thread safe **Q**ueue, were the coordinator is blocked waiting to process them.

Listener

The Listener is a thread blocked on listening socket, waiting for ground stations to connect. Once one does, it instantiates a new Proxy object and passes it the newly established connection's socket. After which, a reference to this object is inserted into a thread safe queue used to inform the Coordinator of the newly connected ground station.

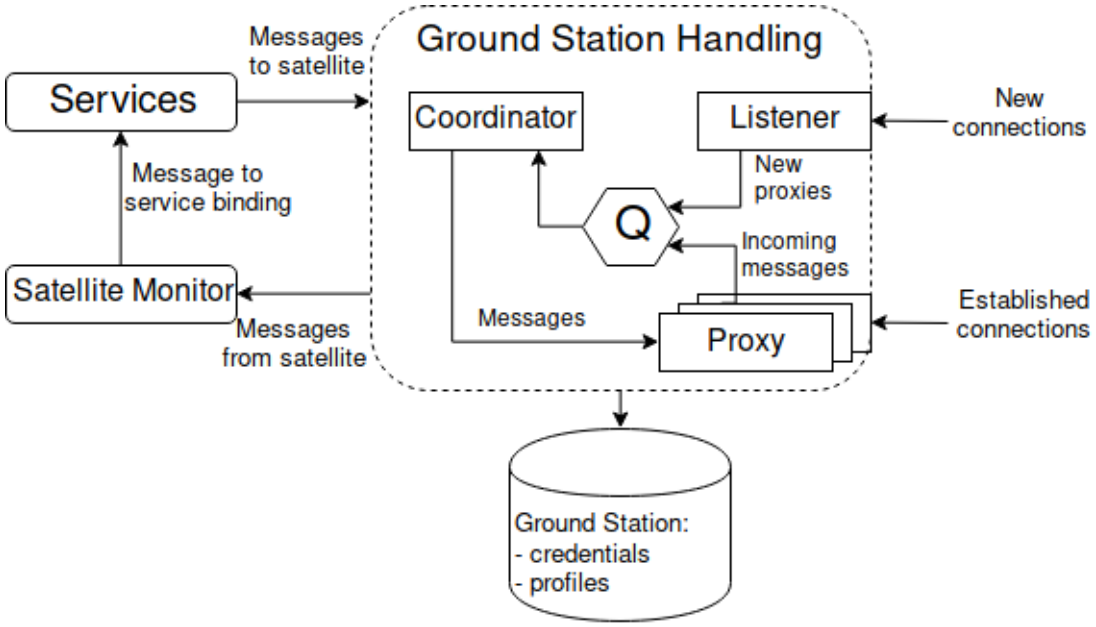


Figure 4.8: Core's ground station handling

Proxy

Once created, the proxy then handles the authentication of the ground station. Only then, and if successful, can it be used for normal communications. Internally, each proxy has thread blocked on its socket waiting for messages, Once one is received, it unserializes it (i.e. converts the JSON object into the appropriate internal class) and places the message on the queue, were the Coordinator will pick it up. Furthermore, when the connection is lost a corresponding message is place in the queue and the thread terminates.

Coordinator

The coordinator is were most of the ground station handling logic is handled. In particular, whenever a new proxy is established or lost (i.e. disconnected) or a ground station enters or leaves the satellite's LOS it assigns a new DGS using the process described in subsection 3.5.2. Once a new ground station is designed it crafts the appropriate connect messages, mentioned in subsection 4.3.1, one to disconnect the current DGS and another to order the new DGS to connect to the satellite. When transitioning from one to no DGS only the disconnect order is sent. Similarly, when transitioning from no DGS to one only the connect order is sent.

The coordinator is also responsible, for sending and receiving messages from the satellite. Sending messages, involves passing a byte array containing the crafted INCP message which is then encapsulated in the appropriate GSC message and sent to the DGS. If however, a message pertains to a scheduled diagnostic, it enables the caller to specify the ground station to which send the message. Reception of messages from the satellite is performed by a thread blocked on the aforementioned queue. When a INCP message is received, contained in a GSC one, it unpacks it and delivers it the *Satellite Monitor*, as shown in Figure 4.8.

Satellite Monitor

The received INCP messages, which are still serialized, are passed on to the monitor seen in Figure 4.8. This entity has two roles. First, is unserializes the received INCP message. Then it delivers the message to the appropriate service. Additionally, the monitor is also responsible for notifying the services of several other events regarding the satellite. In particular, when the DGS changes it informs the services such that they know 1) if the satellite is in LOS a ground station and 2) the profile of the current DGS.

4.4.2 Services

In this subsection, we address the several services which composed the Core. Since multiple users may simultaneously invoke functionalities of the service, each must handles its own concurrency issues. This way, the exposed API, used by the RESTfull interface, is oblivious to such issues.

Data

When the satellite enter in LOS and a DGS is chosen, the data service sends a INCP *reqData* message to each subsystem requesting all data since the last communication window. Recall from subsection 3.4.6, that we can specify the time instant we are interested in, and thus only relevant data is sent. Then, while the connection holds, a new request is sent for all data more recent than the one received from the initial request. All received *Data* message's data is stored in the database using the scheme detailed in subsection 4.4.3.

Two major external functions are provided by this service to the users (i.e. REST interface). One for retrieving data stored in the database and another for requesting specific data from the satellite (without relying on the automated process). The data is retrieved from the DB were the parameters specify *(subsystem, id)* tuple and the time range.

Error

Like the in data service, the error service periodically requests the logs when the satellite enters LOS of a DGS and periodically afterwards. The requesting of errors is performed via the process detailed in subsection 3.4.5. After all information is received, it's stored in the database and a function is exposed to retrieve said information which can be accessed by the RESTfull interface.

Commands

The INCP process which allows for commands is detailed in subsection 3.4.4 but simply put a reply message is sent with set format and a reply is received with another set format.

The command service provides one function to execute a command, which accepts as arguments the subsystem which is to execute it, the opcode and a dictionary[13] (a set of key-values) containing the input arguments of the command. How the caller (e.g. users) knows which arguments any one command has is addressed in subsection 4.4.4. Note that validation of user permissions is not performed here, in fact,

no service has any knowledge of users. These details are handled at the RESTfull interface layer and are addressed in subsection 4.4.4 instead.

Based on this function signature the *function* messages are constructed dynamically. Therefore, the service needs to validate that the input arguments are those described in the associated *stable_t* detailed in subsection 4.2.3. This is performed as shown in Listing 4.11:

```
def valid_args(self, stbl, **kargs):
  # A (name, incp_type_e) dictionary
  expd_args = incp.py_stbl_members(stbl)

  # The number and names of the arguments must
  # match the ones defined in the stable_t
  if len(kargs.keys() & set(expd_args)) != len(expd_args.keys()):
    return False

  # Validate each type
  for name, arg in kargs.items():
    if incp.py2incp_type(arg) != expd_args[name]:
      return False

  return True
```

Listing 4.11: Argument validation (some details were omitted)

Construction of the message can then be seen with the example in Listing 4.12 (some details were removed for clarity. Additionally, *getattr()* and *setattr()* are Python builtin functions for retrieving and setting values from an object given the attribute's name).

Recall from Figure 4.3, that the name of the message type (e.g. the *msg_add2_t* from before) is stored in the *stable_t*. Using it, we can extract the constructor function from the SWIG generated code, see line 4 of Listing 4.12. Then, a new object wrapping the body of the INCP *function* message (e.g. the *msg_add2_t*) is instantiated, line 6 of the example. And the input arguments of the command are set, lines 6-7. Finally, the *function* is then complete in line 10.

```
def build_msg(self, subsystem, opcode, **kargs):
  stbl = incp.stbl_fn(subsystem, opcode)
  if valid_args(stbl, kargs):
    constructor = getattr(incp, stbl.name)

    body = constructor()
    for k, v in kargs.items():
        setattr(body, k, v)

    incp.msg_function_build(builder, opcode, body};
  # ...
```

Listing 4.12: building a *function* message

[13]Python dictionaries explained: `https://docs.python.org/3/tutorial/datastructures.html#dictionaries`

Once the message is built it's delivered to the SS service which calculates and adds a MAC before sending it to the Ground Station Coordinator.

Once a reply is received (and validated by the SS service), the results are stored in the database and made available to the retrieved via a set of functions like for data and errors.

After the message is sent, the caller is returned a future[14]. This object can then be used by the caller to block itself (via an internal mutex) waiting for the command service to receive the reply and place the output arguments in the future in such a way that the caller is notified and can then access them.

Diagnostics

The diagnostics service supports two major services, one for issuing diagnostics for the satellite to execute, and one to schedule them. Additionally, way of accessing their results is also provided.

For issuing normal diagnostics a simple function can be used were the subsystem and diagnostic *ID* are provided as arguments and a future is returned. Since diagnostics involve critical instructions these must, like commands, be sent trough the SS service. Once the *diagnosticStart* message is received an entry in added to the database indicating that the diagnostic is ongoing. Finally, once the Diagnostic Result is received the future is set with the resulting data. However, the communication window may close before a response is received. In such cases, at the start of each window, the service consults the database for outstanding diagnostics and requests the results of each one, as detailed in subsection 3.4.4. The replies are then, matched using the start timestamp stored in the database with that of the replies. If the issued diagnostic has no matching reply (e.g. the subsystem was restart in the meantime) the database is updated accordingly.

Scheduling diagnostics, in addition to the previous arguments, the caller must also specify the ground station. The service, then craft the request message as normal, but instead of the SS service, uses the PMA service to add the MAC to the message and then send the message to the ground station.

Secure Session

The Secure Session service extends upon the *ss_ctx_t* structure detailed in subsection 4.2.4. In particular, it establishes the connection with the satellite once the other services need to send messages securely, and the ground station's profile supports it. The functioning of the state machine which governs the service can be seen in Table 4.1.

PMA

Since the PMA is relatively simpler version of the SS, as detailed in subsection 3.7.5, the implementation is also simpler. However, the SC needs to be stored persistently. This if performed by using a dedicated table with a single row keeping the counter.

[14] Explanation of the concept: `https://en.wikipedia.org/wiki/Futures_and_promises`
Python's Future implementation: `https://docs.python.org/3/library/concurrent.futures.html`

Table 4.1: SS service state machine.

	Disconnected	**Handshake**	**Connected**
send message	Save message and send *SSHello*; → Handshake	Save message	Add MAC and send message; Increment TXSN
received message	discard message	discard message	Validate MAC; If valid increment RXSN, otherwise discard message
receive *SSHello*	N/A	Set nonce; Send saved messages; → Connected	N/A
connection window closes	Do Nothing	→ Disconnected	→ Disconnected

4.4.3 DataBase

In order avoid manually handling the low level details of interaction with the database we opted to use SQLAlchemy[15]. In part due to its object-relational mapper (ORM), which allows for the data mapper pattern, where classes can be mapped to the database. Which is how SQLAlchemy handles the laborious and tedious work of creating queries for insertion and retrieval and tables and rows. Furthermore, it does this without hiding the underlying SQL related processes. Instead, it provides abstractions transparently built on top of these processes. It's this transparency what we leverage in order to the create the data, errors, commands and diagnostics tables with SQL types equivalent to the ones found by introspection of libINCP. Furthermore, it enables to chose from several DB backends (sqlite, MySQL, PostgreSQL, etc). Thus, we can have one backend for the final version, and one for unit testing were we want speed and simplicity (i.e. no server configuration required). In the latter case, an in memory version of sqlite is used.

The database's schema can be seen in Figure 4.9. In it, and starting at the bottom, we see three tables: Users, GroundStations and Events. The *Users* and *GroundStations* are self explanatory. However, note that the expected fields for credentials (i.e *password*/*secret* columns) store the iterated salted hash as described in section 3.7.

The *Events* table stores the timestamp when it occurred and the subsystem to which it pertains to. As can be seen, an event can be one of four types: a command execution, a diagnostic, retrieved data or a reported error. Each of the tables' columns matches the information of the INCP messages, as detailed in section 3.4. Additionally, commands and diagnostic also save which user issue the instruction. Furthermore, the diagnostics can be in either an issued or done state, and store which ground station it was issued from.

Recall from section 3.4 that each of the INCP messages (e.g *Data* and *DiagnosticResult*) carries some specified information which can be determined by analyzing the libINCP, as detailed in section 4.2. Since the message's data must of course be stored for later access, we needed a way to do just that in a SQL database context. The way we opted to perform this task is by generating the tables were this data is stored (in particular the columns with "*FROM INCP*") by introspection of libICNP. For instance, the *DataValues* table is in fact several tables (1 per variable) where the *value* column's type is defined

[15]SQLAlchemy homepage: `https://www.sqlalchemy.org/`

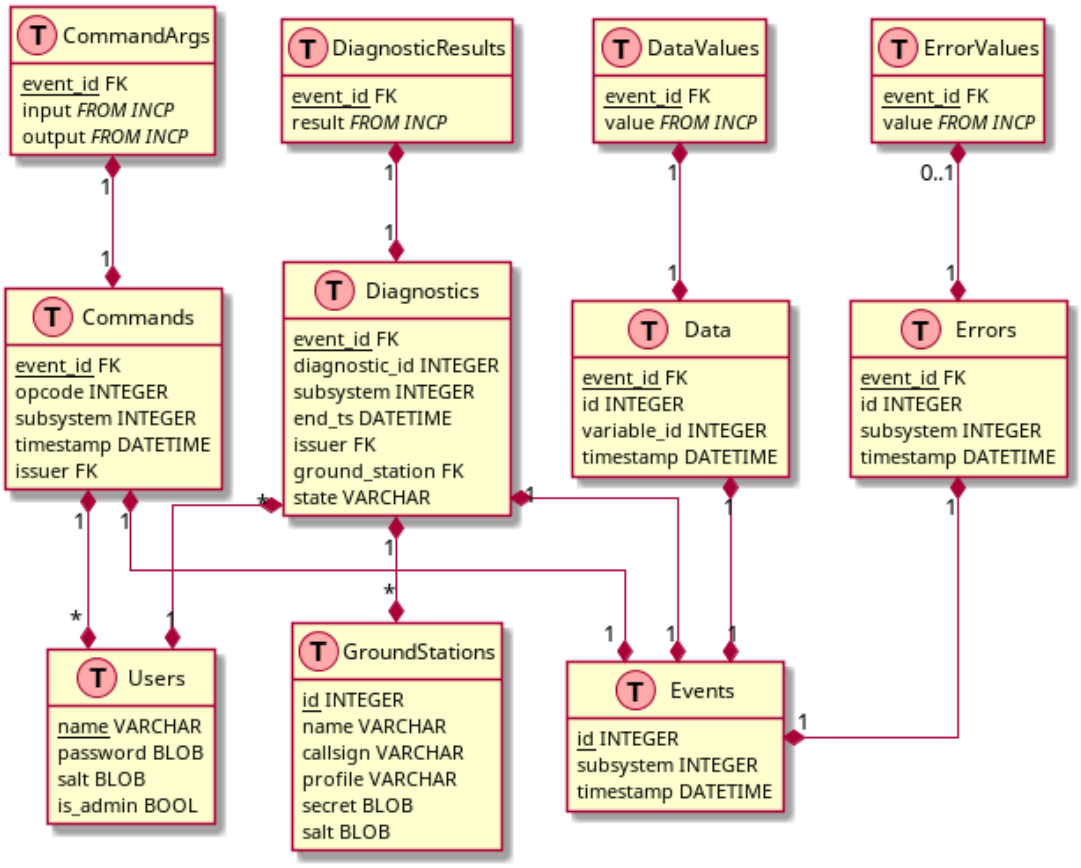


Figure 4.9: Database Entity-Association model

by the one in libINCP. For the *ErrorValues*, *CommandArgs* and *DiagnosticResults* a similar process is used. Therefore, the database schema is always up-to-date with the libINCP. This does, mean that at the start of the Core's program executing, when it determines the schema to use, it must validate the one in the database. This however, is merely a question of comparing the generated schema to the one in the database.

It's important to note that we could have chosen instead to stored the data in a BLOB like column and convert it to the appropriate type in the Core server. However, we would still have to implement processes to handle this. Thus, the real choice is between handling this issue at the Core level (i.e. its services) or at the DB level. As shown, we opted for the DB as it has the advantage of storing the data in the appropriate format and thus does not require any transformation to use.

4.4.4 External interface

The external interface which the core exposes to the users as four main components and functions, as shown in Figure 4.10. Firstly, it provides the static content needed for building the web application interface. Secondly, it provides a JSON file (i.e. the mentioned INCP.json) such that the interface knows which data and functions the Core supports regarding its various services. Thirdly, it provides a RESTfull interface providing access the to the Core's services. And fourthly, authenticates and maintains user sessions.

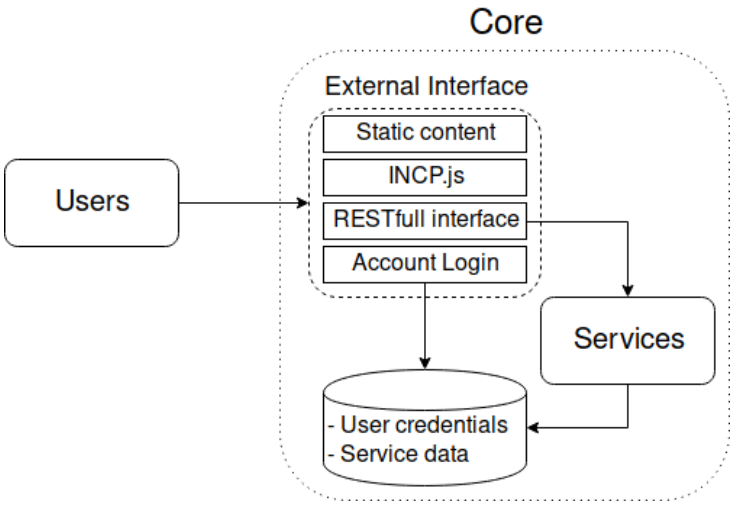


Figure 4.10: External Interface

Currently, the static resources of the web application are being served by the Flask framework. For development purposes this is preferred since its simple to setup. However, it does not scale well, meaning that for the final version we will have to use a different WSGI server such as Apache webserver.

The *INCP.json* is a JSON version of the introspection data present in libINCP. It's this file, created using Flask's template mechanism, that is then passed on to the web application such that it can request and shown the available data sets, error, diagnostic and command logs, as detailed in section 4.5.

regarding the RESTfull interface it serves as a wrapper for the service's, detailed in subsection 4.4.2, such that these can be invoke remotely. Furthermore, it's also in this layer, that the users are authenticated and access to critical functions (e.g diagnostics and commands) is controlled.

The authentication and handling of user session is relatively straightforward and follows what was described in section 3.7. The users provide their username and password which is then validated with the stored credentials. Then a cookie is generated and returned to the user which in turn provides it when using REST calls.

4.5 User Interface

At the time this document's writing the user interace is still in a rather raw state as can be seen in figure Figure 4.11. Altough, it creates the data view and fetches the data from the Core based on the INCP.json file commands are still hard-coded. Furthermore, only limited status information of the ground segment is shown.

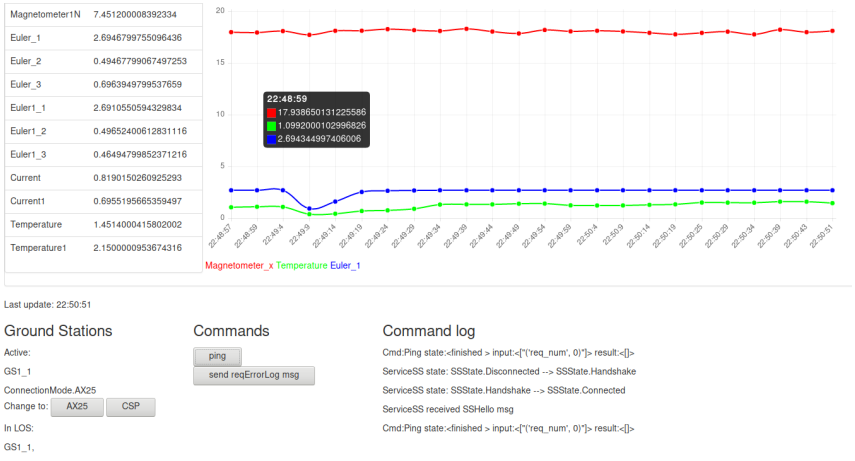


Figure 4.11: User Interface screenshot

Chapter 5

Validation

Following the guidelines in NASA's Software Safety Guidebook[51] we designed and used several checks which validate the expected functioning of the ISTNanosat-1 Ground Segment. Specifically, we have four kinds of testing, each with an increasing level of granularity:

Firstly, in section 5.1, we detail how we used static code analyses tools to diagnose issues with the developed code. Whole classes of errors are eliminated without having to dedicate any effort other than an initial configuration and setup phase.

Secondly, in section 5.2, we briefly discuss how, which and what unit test were implemented. These units tend to either be classes, or types and functions logically connected. Note that unit tests can be used both for testing and development purposes. However, both ends depend on the ability to test one and only one "unit" in isolation and with as much thoroughness as desired.

Thirdly, in section 5.3, we detail the employed integration tests used to validate the end-to-end functionalities of the entire Ground Segment. These tests build upon the unit tests to validate areas which the latter do not cover. Furthermore, integration tests validate the units, when combined, work as intended. Note that another class of tests are the so called functional tests. Although, there are some differences between the two, we combined them and address them only as integration tests.

Fourthly, in section 5.4, we performed a test were the Ground Segment deploy in an environment were the operating conditions are beyond the expected range. The goal is essentially to stress the system in such a way that if there is a flaw, it's revealed by the test.

Additionally, in section 5.5, we go over the checklist provided in [51]. This checklist basically has a series of checks which cover the most common bugs and errors found in critical software.

5.1 Static Code Analyses

In this section we detail the used tools with which we perform the static analyzes of the developed code. All of these checks are intended to be executed on every successful compilation. The goal being to detect errors as soon as possible such that correcting them is has fast and occurs before any damage is caused. Note that all of these diagnostics could hypothetically be performed manually. However, these

tools enable shifting the burden from the developer onto the used tool in fast and predictable fashion.

We perform our analyses of C based code in two stages:

First, we considered and activated all relevant warnings/diagnostic which the used compiler (clang[1]) supports. However these diagnostics are very specific. Typically these only look at one source code statement at a time and only perform very objective checks.

Secondly, we use the *clang-tidy* tool to perform another more comprehensive level of checks, which end up being more time consuming. Unlike the compiler warnings, these can look at execution scopes. A good example for a clang-tidy diagnostic, is a function that at one point performs an allocation but discards the pointer without ever releasing the allocated memory.

For the Python code-base, we use a tool called *mypy* which enables type checking to the otherwise dynamically typed language.

5.1.1 Compiler Diagnostics

The selection of the compile time warnings was performed by looking the recommendations of [51] and the compiler's diagnostics reference table[2].

-Werror All warnings are treated as errors meaning a compilation fails if a warning is found.

-Wall This flag actives a set of other flags and should be enable in any non-legacy code. Its most important checks are: **-Wswitch** for branch checking in switch case statements. **-Wunused** for unused variables typedefs and function. **-Wuninitialized** for detecting uninitialized variables.

-Wextra Like **-Wall**, this flag should also be used for any non legacy project. However, this group of flags tend to have a higher false-positive rate (i.e. the warning triggers but there is no bug). Its two most important warnings are: **-Wsign-compare** which triggers when comparing signed and unsigned integers; **-Wunused-parameter** for detecting unused function parameter; And **-Wmissing-field-initializers** which triggers when a structure is only partly initialized (i.e. some member is forgotten).

-Wshadow-all This flags activates a set of warnings for detecting name repetition (i.e. shadowing) for types, variables, parameters, etc.

-Wstrict-prototypes, -Wmissing-prototypes and -Wmissing-declarations These flags relate to the consistency between source and header files. Respectively, these assure function declaration and implementation have the same signature; All implemented functions have a declaration in a header (unless marked as private via the *static* keyword); And checks if functions,types,global variables, etc, are being used without including the required header.

-Wcast-qual This flags triggers an warning when performing a cast which drops qualifiers, for example, a *const int* to an *int*.

[1] Clang homepage: `https://clang.llvm.org/`
[2] Clang diagnostic reference table: `https://clang.llvm.org/docs/DiagnosticsReference.html`

-Wpointer-arith Prevents invalid void pointer (i.e. *void **) manipulation and certain *sizeof()* or *alignof()* usages.

-Wconversion Activates a set of flags which prevents implicit conversions between types which lose precision (e.g. *int32_t* to *int16_t*).

-Wfloat-equal Prevents comparing floats using *==* and *!=*. When we do want to compare two floats, an error interval must be defined and the difference between the values check to see it's within that interval.

5.1.2 Clang-tidy

In addition to compile time checks, we used a static code analyzer named clang-tidy, which is part of the clang/LLVM compiler toolkit. We run clang-tidy using a bash script, as shown in appendix A, on all source files with **all**[3] but a set of configured checks. In particular, those which are still in alpha, are OSX specific or assume LLVM's code style were disabled. Clang-tidy's checks are organized in groups. The more important ones are:

cert- Checks related to CERT Secure Coding Guidelines[4]. Note that not all rules have corresponding diagnostics.

clang-analyzer- Clang Static Analyzer checks. Some of the checks in this group are already covered by the compilation flags. However, it also performs general-purpose checks such as division by zero, null pointer dereference, unsafe api usage (e.g. *gets()*) and Unix system calls consistency (e.g. memory leaks, double free, and use-after-free and offset problems involving malloc.)

readability- Checks that target readability-related issues such as unnecessary castings and redundant declarations.

5.1.3 Mypy

Mypy[5] is a static type checker for Python which uses annotations defined in PEP 484 to verify functions, variables and classes signatures and usage. Since mypy is a static analyzer, or a lint-like tool, the type annotations are just hints and don't interfere when executing the program. Essentially, the Python interpreter ignores all annotations treating them as comments. The advantages of mypy are threefold: It eliminates certain classes of bugs at compile-time by type checking function, variable and class usage; Increases readability of the code which makes it simpler to reason with and thus to extend and maintain; Finally, type annotations greatly increase the usefulness of auto completion tools (which tend to have trouble with dynamically typed languages), decreasing development effort.

[3]Clang-tidy's check list: `https://clang.llvm.org/extra/clang-tidy/checks/list.html`
[4]CERT homepage: `https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard`
[5]Mypy homepage: `http://mypy-lang.org/`

5.2 Unit tests

Unit tests consist in isolating the developed code in logical “units”, such as classes, types and functions and then exhaustively testing them in isolation. Since the code is likely to contain defects, these tests will be re-executed with each new iteration of the unit, usually after each successful compilation.

Testing the possible inputs to a unit is important, since it may not be feasible or practical once the unit is integrated into the system. Unit level testing can also identify implementation problems requiring changes to the software. The earlier these issues are identified, the less costly they are to fix.

The two main areas which are to be tested are the Core, ground station application and the libINCP. These are implemented in either Python or C. Therefore, we use two different frameworks for implementing and executing the tests. PyTest[6] for the Python code-base and bcunit[7] for the C one.

5.2.1 Core

The Core has two main components to test, the services and the ground station handling. As previously mentioned, the used framework for implementing and executing the tests is PyTest and an example execution output is seen in Listing 5.1.

In order to test the services we must consider their context. Firstly, most services involve some database access and communication with the satellite via message exchanges. Secondly, while communicating with the satellite, it may enter or leave the LOS of the DGS. For each functionality of each service we must first test the ”happy-case” and then each of the failure cases triggered by the aforementioned events. For example, for sending a command we compare the produced INCP *function* message to the expected values, then a generated *functionRet* message is injected to the monitor as if it were received by the Coordinator. Finally, we assert the database and the returned value for the expected results using the services API dedicated to this task. In this process the services does not use the real Coordinator class of figure 4.8. Instead, a mocked version is used such that the service (i.e. the unit) is tested in isolation and in the conditions we so specify. Note that the command service expects a set of input arguments passed via a Python dictionary (the data structure). This input argument validation is checked by a separate set of tests. For the remaining services, a varying number of tests are employed based on the service’s complexity. For example, the SS service’s state machine is fully covered.

The Core’s ground station handling does not have much easily testable code since it mostly handles external resources (i.e. sockets). However, the DGS selection and the GSC messages can and are tested. Specifically, testing the selection process involves defining various combinations of ground station profiles and comparing the GSC select to the one expected. Furthermore, testing the messages involves constructing a message which is then serialized and unrealized. The end result is compared to the original message.

Additionally, a test suite for the libINCP bindings is performed. The goal not to test the INCP library, but instead, to validate that the python bindings were correctly generated as expected.

[6] PyTest homepage: `https://docs.pytest.org/en/latest/`
[7] BCUnit homepage: `https://github.com/BelledonneCommunications/bcunit`

```
================ test session starts ================================
platform linux -- Python 3.6.1, pytest-3.0.7, py-1.4.33, pluggy-0.4.0
rootdir: /home/alex/projects/GroundSegment/Software, inifile:
collected 52 items

src/tests/python/test_coordinator.py ....
src/tests/python/test_db.py .
src/tests/python/test_incp.py .....
src/tests/python/test_incp_monitor.py .
src/tests/python/test_internal_msgs.py ....
src/tests/python/test_service_command.py ....
src/tests/python/test_service_data.py ...
src/tests/python/test_service_diagnostic.py .......
src/tests/python/test_service_error.py ..
src/tests/python/test_service_pma.py .....
src/tests/python/test_service_ss.py .....
src/tests/python/test_util.py ..........
================ 52 passed in 0.36 seconds ========================
```

Listing 5.1: PyTest unit test result

5.2.2 ISTNanosat Control Protocol

The libINCP related tests cover three main areas: The messages themselves; the SS, PMA and related algorithms; and the consistency of the serialization tables detailed in subsection 4.2.3. The result of executing the tests can be seen in Listing 5.2.

Most tests of libINCP revolve around testing the building, serialization and unserialization of the various types of messages. The simplest messages, each have a single test. These involve a message which is built, serialized, uninitialized and the end message is compared to the original, asserting that the two are equal. However, certain more complex messages require more tests to be thoroughly validated. In particular, the *data* message has many tests with multiple configurations of singular and periodic fields of each of various sizes, place in the message with several different orders and in various quantities.

Another crucial parts of libINCP are the SS and PMA related components. Therefore, we implemented tests such that these are covered by our test suit. For instance, the SS is tested first by creating a sequence of messages, generating their MACs and then validating them. Thus the normal case of a sequence of messages with their SNs are tested. As with all testing, but especially in security related units, we must assure that it doesn't work in the erroneous cases (e.g. does not accept a message with a repeated SN). These tests consist of creating multiple messages with incorrect SN, nonces, MAC lengths, etc. Then we check if the *ss_ctx_t* accepts the MAC of any of these invalid messages.

The consistency of the serialization tables is performed by a single test. Recall from subsection 4.2.3 that serialization tables are used to describe plain C structures such that these can be (un)serialized. However, the structure's fields must be kept consistent with the table. Therefore, a test was implemented which validates that all the members of the structure are present in the serialization table. The most critical issue is how we know the members of a C structure. Fortunately, the SWGI generated classes

contain this information. Therefore, in python we can at run-time check which members a certain class has. By knowing the opcode of a message we can determine it's generated class via the serialization table. Knowing the class, we can compare it's members with the ones in the serialization table and check if these differ. Therefore, for each serialization table, of each message, of each subsystem, we compare the generated class with the associated table and check if their members match. Ideally, we would want to perform this validation at compile time. Unfortunately, the C macro system is not powerful enough to do it. However, with good practices we can guarantee that this test is always executed and thus guarantee the consistency of the tables.

```
Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites     11     11    n/a      0        0
               tests     54     54     54      0        0
             asserts    803    803    803      0      n/a

Elapsed time = 0 seconds
```

Listing 5.2: BCunit unit test result

5.3 Integration tests

While the unit tests validate each "unit" in isolation, here we join the various components and validate that the overall functionalities of the Ground Segment work as intended. Specifically, we performed three large groups of tests. Firstly, in subsection 5.3.1, we detail the various services and their functionalities exposed trough the RESTfull interface. Note that although the services depend on the INCP we don't test it explicitly. In part, because message (un)serialization was already extensively tested by unit tests. Furthermore, what can't be covered by unit tests is instead tested by the integration tests. Then in subsection 5.3.2, we address the integration tests pertaining to ground station control and multiplexing discussed in section 3.5. Finally, in subsection 5.3.3, we validate the various mechanisms which compose the Ground Segment's security scheme, which was detailed in section 3.7. The test environment corresponds to the one discussed in section 4.1.

5.3.1 Services

In order to test the data and error services, we first defined in the satellite emulator a well know set of telemetry. What exactly comprises this telemetry is not very important. What is important however, is that the retrieved, stored and latter accessed telemetry (from the Core) is exactly the same as the one in the emulator.

Data and errors

After programming the emulator to only transmit this set of telemetry, we initialized the test environment and let Core automatically retrieve it. After some time we collected the information from the Core via the REST interface and compared it to the expected result.

For the error the retrieval involves only the process detailed in subsection 3.4.5. The data retrieval however has two parts. First, we used a virtual variable to request a large set of data. This imitates the behavior of system when the satellite enters in LOS of a ground station.

In a second test, we killed the ground station while in the middle of the telemetry transmission and then re-executed it. After which, the Core continued the data retrieval and still managed to retrieve all data.

Commands

Testing the command service is rather simple. In particular, since the validation of the input and output arguments and (un)serialization is thoroughly verified by unit-tests. What remains is the end-to-end test of the service. We implemented a ping command which is accepts an integer as the input argument and the emulator replies with a *functionRet* message containing, as output argument, that very same value.

Diagnostics

For the diagnostics we defined two special diagnostics in the emulator which both return a known constant result. The difference between the two is that the first starts and ends immediately. While the second one, only finishes after a certain amount of time has passed. Then using these diagnostics several scenarios were devised. Firstly, we order the core to perform the fast diagnostic and then check if the result trough the RESTfull interface. Secondly, the slow diagnostics is issued but before it finishes, we consult the Core to confirm that it marks the diagnostic as ongoing. Thirdly, the scheduling of diagnostics was validated. In this scenario, we first killed the emulator. Then, we scheduled a diagnostic to be issued by the ground station. Subsequently, we killed the Core and then re-executed the emulator. Finally, we re-executed the core and confirmed the successful retrieval of the result of the diagnostic.

User permissions

As explained in previous sections, there are several types of users each with different capabilities.

Therefore, we need to validate that the permissions are checked when invoking certain REST calls. Testing this validation is relatively straightforward. We simply try to execute, without authentication, each REST call which triggers sending messages to satellite. This includes all those for the diagnostics and commands, but also specific telemetry retrieval. Note that guest users should only be able to view the already collect data. Recall that the Core automatically retrieves data without manual intervention.

Validating that critical messages are only sent trough trusted ground station is also simple. We simply changed the profile of the ground station to that of a untrust one and checked that any attempt to issue commands or diagnostics results in an error.

5.3.2 Ground Station Handling

For testing the Core's ground station handling we prepared an environment were instead of one ground station, we had four in the same machine. Of these, one was assigned the the main ground station

profile and two the trusted ground stations profile. Multiplexing of the communications of various ground stations, were each expects a serial port, was performed by using virtual serial ports, as described in subsection 4.1.2.

The first test consisted in killing each ground station process by decreasing order of their profile's priority. First, we killed the main ground station and verified that one of the trusted ground stations was then selected has the DGS. Then, that one was killed and we verified if the remaining one was the new DGS. After killing the remaining two ground stations we inverted the process. We executed each ground station, starting with the not trusted one. This way, we validated that the core was selecting a new DGS whenever one with a higher priority was available. Note that, after each DGS transition we performed a data retrieval operation to verify that the we could still communicate with the satellite by the expected ground station.

After the first test we concluded that the Core's ground station selection algorithm was working as intended. However, we triggered the ground station handover by killing the processes but in normal conditions this is instead triggered by the loss of LOS. Therefore, a second test was performed, similar to the first but with 1 key difference. Instead of killing the processes, we block the communications of the virtual serial port of the intended ground station. Recall, that the multiplexing of the various ground stations is performed by a python script of our design. Since the various DGSs were still selected by the expected order we concluded that the LOS detection by the satellite's beacons was functional.

5.3.3 Security

To test the security we need to validate the various mechanisms described in section 3.7. Specifically, the user and ground station authentication, and ground-space segment security.

User Authentication

The first mechanism to test was the user authentication. Authentication is performed by a REST call were the username and password are provided and either a cookie or an error code is returned. First, we test that a user with correct credentials can successfully authenticate itself. Then, we defined one set of usernames were all but one are non-existent and one of passwords (all of which invalid). Each and every combination of the two sets is used to perform a login attempt. After no login attempt is successful we concluded that the user authentication is working as intended. Note that both sets include edge-case values. For example, usernames with a size larger than that allowed by the database or an empty string (i.e. "").

Ground Station Authentication

The ground station login mechanism is similar to that which the users use. The key difference is that instead of REST call, a GSC message is sent during an established connection. However, and like the previous test, we first authenticate a ground station with valid credentials. Then several combinations

of identifiers and secrets are used, were each is expected to not successfully authenticate the ground station.

Secure Session

Testing the normal case of the SS is already performed when testing the command and diagnostics services. Therefore, we only need to confirm that messages which contain an invalid MAC are not accepted. The test then consisted of several scenarios were we confirmed the satellite to not accept a command INCP *function* message by listening to the replied error. First, we validated the MAC itself. Several crafted messages were sent one had no MAC, and the rest each had one incorrect parameter of Equation 3.1. In particular, we used several different SNs and nonces. Secondly, we verified if the Core's generated nonces were truly random. This was performed by forcing the Core to generate several nonces. both in the same and different executions; and separate reboots of the machine. Then checking for any repeated values.

Persistent Message Authentication

For the PMA, a test similar to that of the SS was employed. Specifically, we constructed several messages with incorrect parameters of Equation 3.2 and verified that none was accepted. Furthermore, we also verified that the Core's SC was being stored persistently.

5.4 Load test

Load tests are designed to see how much the system can handle before it breaks down. The usual aspects of the system that might be stressed are CPU usage, I/O response time, paging frequency, memory utilization, amount of available memory, and network utilization. The closer a system's peak usage is to the breakdown point, the more likely it is that the system will fail under usage.

In order to validate the developed applications and how these perform under full load caused by continuous message reception from the satellite. The goal of this test is to confirm that the available resources are sufficient and that no anomalous behavior is observed in these conditions.

The test itself is simple and has three phases. During the first phase, both the core and ground station application are *not* running such that we can profile the machines resource usage at rest. Then both the ground station and the Core application are executed. Ten seconds afterwards the Core sends a message to the emulated satellite. This causes it to continuously send INCP *data* messages as fast as the serial link between the core and the ground station allow it. We collect measurements of the machines during all of these phases and the last two have a duration of 10 and 50 seconds, respectively.

The test was performed on the development environment detailed in subsection 4.1.2. The Core's machine is a Virtual Private Server with 1GB of RAM and a 1 core virtual CPU (QEMU Virtual CPU version 1.5.3) with operating frequency of 2397 MHz. The ground station is running on an old 2 core CPU (Intel core 2 duo) with 4 GB of RAM. As previously mentioned, the ground station and the emulated

satellite (running on the previously mention Raspberry-PI3) are connected by a serial port. For the purposes of this test we configured it to have 115200 bit/s of bit-rate. Recall that on the end system the maximum bit-rate is 48000 bit/s. We opted for this rate in order to have some safety margin to normal conditions as recommended by NASA's Software Safety Guidebook [51, p. 190].

We measure the system load in percentage of used CPU, the used memory of the system for both machines using kSysGuard[8]. For the ground station, this last measurement is made both for the satellite link and the Core link. Additionally, we used the valgrind[9] tool to confirm that there are no memory leaks.

5.4.1 Core server

The Core's measured results can be seen in Figure 5.1. In it we can see during the first phase of the test, that resource usage in the machine is minimal. Once the application is initialized we see a spike in terms of CPU usage corresponding to both the Pythons's virtual machine and our application initializations. Regarding the used memory, the Core's process consumes 31.7 MBytes (3.1%) of the 1 GByte total memory and remains unchanged during the entire test.

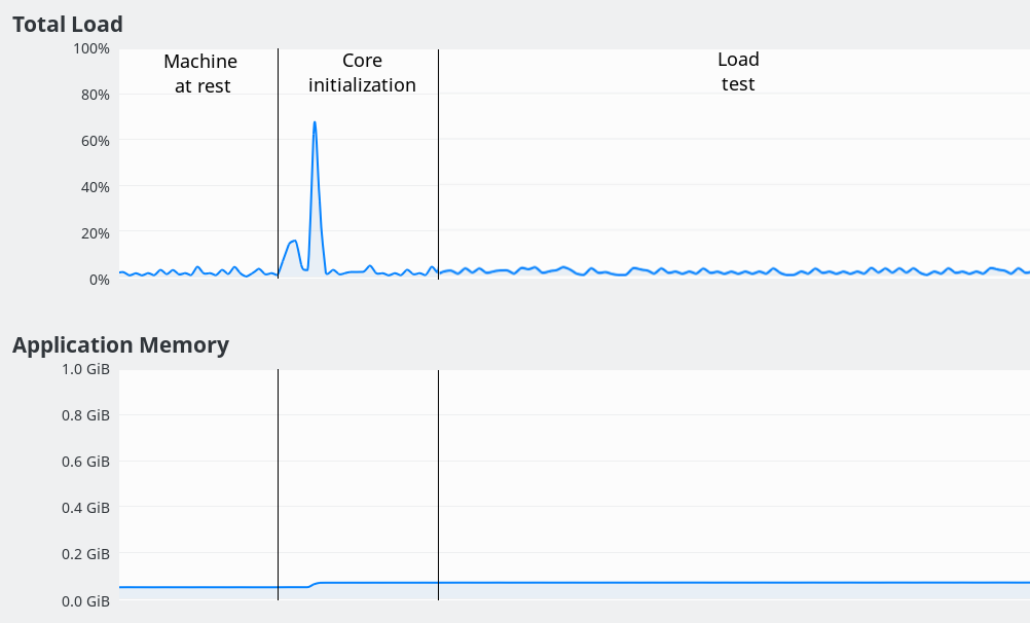


Figure 5.1: Core load, memory during load test

From the figure, we can conclude that the Core server performs as expected. Even during a stress situation were data is transmitted at rate more than double than the normal case.

5.4.2 Ground Station

If viewed closely, in Figure 5.2 we can see some jitter in terms of CPU usage during the last phase. ground station has overall insignificant memory and CPU usage throughout the test. The data rate

[8]kSysGuard homepage: `https://userbase.kde.org/KSysGuard`
[9]Valgrind homepage:`http://valgrind.org/`

however, clearly shows the configured 115200 bit/s for the serial port. Looking at this data we can conclude that ground station application perform perfectly under a stress situation.

Figure 5.2: Ground station load, memory and troughput during load test

After the test we collected the results of the memory analyses using valgrind as shown in Listing 5.3. The arguments of the *gs* application can be ignored for the purposes of this test.

```
valgrind \
  --read-var-info=yes --tool=memcheck --leak-check=yes \
  --show-reachable=yes --track-fds=yes \
  ./build/gs GS1 SAT 11 4 193.136.102.249 GS1_1
```

Listing 5.3: Memory inspection using valgrind

The valgrind inspection result can be seen in Listing 5.4. The heap summary shows that 66,715 bytes being used when the test ended. In particular, 60,591 bytes were still reachable. These correspond to the required working memory such as libCSP, the proxies' and their threads. Starting the application and immediately killing it results in 58,351 bytes of working memory which further supports this conclusion. The 6,060 bytes marked as possibly lost, also appear with exactly the same value when we immediately kill the application after executing it. Therefore, whatever the cause for this number, it's most definitely not a leak. Furthermore, since there are no definitely or indirectly lost memory we conclude that no critical memory leakage exists.

```
==7378== HEAP SUMMARY:
==7378==     in use at exit: 66,715 bytes in 338 blocks
==7378==   total heap usage: 682,844 allocs, 682,506 frees, 46,562,438 bytes allocated
==7378== LEAK SUMMARY:
==7378==    definitely lost: 0 bytes in 0 blocks
==7378==    indirectly lost: 0 bytes in 0 blocks
```

```
==7378==      possibly lost: 6,060 bytes in 100 blocks
==7378==    still reachable: 60,591 bytes in 237 blocks
==7378==         suppressed: 0 bytes in 0 blocks
==7378==
```

Listing 5.4: Valgrind memcheck report

5.5 NASA's Programming Practices for Safety Checklist

In *NASA Software Safety Guidebook* two checklist are provided and recommend for critical code. In particular, appendix *H.5 Checklist of C programming practices for safety* and *H.8 Checklist of generic programming practices* [51, pp. 365-367,370-371]. Since libICNP is to be used in the satellite's subsystems we followed the checklist as to further eliminate the possibility of bugs. Note that some items do not apply to the implemented module and were omitted for brevity.

Minimize use of dynamic memory Only two one-time allocations are needed at startup. One *builder_t* for building messages and one were to serialize them.

Avoid goto's No gotos were used, verified using *grep*.

Initialize variables before use Memory is allocated using callow which zeroes the memory, and variables are guaranteed to be initialized by the compiler flags and *clang-tidy*.

Check the output data validity Great care is taken to assure that only message with correct format are unserialized. Errors during transmission are handled by the AX25 protocol.

Check for dead code Assured by compiler flag.

Avoid pointers to functions None used.

Prototype all functions and procedures Assured by compiler flag.

Always explicitly cast variables Assured by compiler flag.

Avoid mixing signed and unsigned variables Assured by compiler flag.

Don't compare floating point numbers to 0, or expect exact equality Assured by compiler flag.

Do not make assumptions about the sizes of dependent types, such as int The declarations the *stdint.h* were extensively used.

Protect against out-of-sequence transmitted messages Since all links have some sort of transmission control (TCP or AX25-CO) this situation is taken cared of.

Chapter 6

Conclusion

The ISTNanosat-1 Ground Segment presents a system capable of telemetry retrieval and issuing instructions to the satellite. All of the relevant information is then shown to the users using a web application. Furthermore, properly authorized users have the capability to issue the aforementioned instructions. The developed ground segment is composed by multiple ground stations controlled by a Core server. In this centralized design, the Core communicates with the satellite trough the ground stations, which serve as gateways, using the designed and implemented INCP. The INCP protocol depends on CSP whose implementation (libCSP) was extended to support AX25 in connection oriented mode. In the Core, several services permit for the for data and error retrieval, command and diagnostics execution and security of the ground-space segment link.

The INCP was designed to permit the retrieval of only the desired telemetry, produced while no ground stations were in LOS of the satellite. This is performed in such a way that minimizes, as much as possible, the re-transmission of repeated information, by specifying the desired time interval for the desired information. The implementation of the protocol was done in such a way as to permit it's use in the satellite's subsystems. Consequently, the context and limitations of developing software for embedded systems of a satellite were taken into consideration. Since the protocol is to be used by other developers, we took great care in order to maintain a simple API and make the implementation simple to use. Additionally, by coding the commands, data, errors and diagnostics in this implementation the Core can leverage this introspection data is used to, for example, generate part of the database schema.

Unfortunately, we could not test the libINCP on one of the subsystems' boards. Something, which must absolutely be performed. Furthermore, due to the various introspection information some work may be performed in order to, by using compile time flags, exclude unnecessary portions of the implementation. This is especially critical for those subsystems with minimal working memory such as the EPS.

In the Core server, we leveraged the high level nature of Python and it's mature ecosystem in order to produce a solution which is easy to maintain and extend. In particular, the usage of the libINCP introspection information to drive the telemetry and instructions of the services. All of the services functionalities are then exposed via a REST interface. Although the user interface still needs some

polish.

Regarding the ground stations and how these are controlled, the presented solution is both simple and effective. In particular, we present a unified interface which hides the complexity of the AX25-CO and libCSP handling. Furthermore, we also present the implementation of the AX25-CO interface for the libCSP.

The security scheme was designed to minimize the overhead both computationally for satellite's subsystems and in transmission throughput. For the user and ground stations, existing solutions were employed taking into consideration common practices.

Currently, the Core uses the periodically transmitted beacons as indication that the satellite is in LOS of a ground station. Then, it opportunistically selects one DGS which serves as the gateway between it and the satellite. Although in the tested scenarios this scheme works without any major issues, we suspect this will not always be the case when using the radios for ground-space communications. The most obvious case, is the detection of lost communication between the ground station and the satellite when another ground station is in range. In the current approach there will be a time period were the DGS will be out of LOS but before the AX25 connections times-out and a new DGS is selected. If a preemptive handover was performed one may avoid this wasted communication time. In particular, one solution may involve using the known orbit and ground stations' positions. With this information, one may try to predict which is the best DGS at a certain time and when to perform a handover. Although, this may be just possibility that never materializes, this is an issue that needs to be investigated.

Regarding the performed tests and overall validation, it consists manly in functional requirements validation. The load test however, shows without question that the ISTNanosat-1 is a network bound system. Event trough several used libraries and Python itself are known to have significant overheads when compared to naively compiled solutions. Additionally, the performed validation is mainly a functional tests. Therefore, we took the opportunity to automate most of the tests, such that future work is as painless as possible.

Bibliography

[1] Hank Heidt et al. “CubeSat: A New Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation”. In: (2000).

[2] *Erik Kulu’s Cubesat Online Database*. www.nanosats.eu. (Visited on 03/14/2017).

[3] John Scardina. “OVERVIEW OF THE FAA ADS-B LINK DECISION”. In: *Federal Aviation Administration* (June 2002). URL: http://www.faa.gov/asd/ads-b/06-07-02_ADS-B-Overview.pdf.

[4] Richard Van Der Pryt and Ron Vincent. “A Simulation of Reflected ADS-B Signals over the North Atlantic for a Spaceborne Receiver”. In: *Positioning* 7.01 (2015), p. 51.

[5] *GOMX-3 (GomSpace Express-3)*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/g/gomx-3 (visited on 04/14/2017).

[6] Lars Alminde et al. “GOMX-1 Flight Experience and Air Traffic Monitoring Results”. In: (2014).

[7] *GOMX-1 (GomeSpace Express-1)*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/g/gomx-1 (visited on 04/14/2017).

[8] *PROBA-V (Project for On-Board Autonomy - Vegetation)*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/p/proba-v (visited on 04/14/2017).

[9] Sreeja Nag et al. “CubeSat Constellation Design for Air Traffic Monitoring”. In: *Acta Astronautica* 128 (2016), pp. 180–193.

[10] *CanX-7 (Canadian Advanced Nanospace eXperiment-7)*. URL: https://eoportal.org/web/eoportal/satellite-missions/c-missions/canx-7 (visited on 12/04/2017).

[11] Zachary J. Lef f ke. “Distributed Ground Station Network For CubeSat Communications”. Master of Science in Electrical Engineering. Faculty of the Virginia Polytechnic Institute and State University, Dec. 2013.

[12] Dwi Hartanto. “Reliable Ground Segment Data Handling System for Delfi-n3Xt Satellite Mission”. Computer Engineering. Delft Faculty of Technology, 2009.

[13] Christopher A. Kitts James W. Cutler. “Mercury: A Satellite Ground Station Control System”. In: (). URL: http://www.ams.org.ar/lu7eim/SSDL9904.pdf.

[14] Yuya Nakamura, Shinichi Nakasuka, and Yasuhisa Oda. “Low-Cost and Reliable Ground Station Network to Improve Operation Efficiency for Micro/Nano-Satellites”. In: *56th International Astronautical Congress*. 2005.

[15] Robert C Griffith. *Mobile CubeSat Command and Control (MC3)*. Tech. rep. DTIC Document, 2011.

[16] Graham Shirville and Bryan Klofas. “Genso A Global Ground Station Network”. In: *AMSAT Symposium*. 2007.

[17] Dr Ricardo Tubio Dr Antonio J. Vazquez Prof Jordi Puig Dr Naomi Kurahara Prof John Bellardo. *SATNet Presentation, CubeSat WS*. URL: http://mstl.atl.calpoly.edu/~bklofas/Presentations/DevelopersWorkshop2014/Tubio_SATNet.pdf.

[18] *SATNet Github Documentation*. URL: https://github.com/satnet-project/documentation/wiki (visited on 12/23/2015).

[19] Ricardo Tubio-Pardavila. “SATNet Software Architecture”. URL: https://github.com/satnet-project/satnet-main/raw/master/documentation/satnet-3-R1-SoftwareArchitecture-DRAFT-2013.11.04.pdf.

[20] *CAMSAT XW-2 Satellites*. URL: http://amsat-uk.org/satellites/communications/camsat-xw-2/ (visited on 11/30/2015).

[21] Alan Kung. *XW-2 CW Telemetry Encoding Format*. 28th ed. urlhttps://ukamsat.files.wordpress.com/2015/05/xw-2-cw-telemetry-encoding-format.pdf. CAMSAT, Sept. 2015.

[22] “GNU Radio”. In: (). URL: http://gnuradio.org/redmine/projects/gnuradio/wiki.

[23] *Serpens*. URL: http://www.aerospace.unb.br/serpens_radioamateurs (visited on 11/30/2015).

[24] *BisonSat*. URL: http://cubesat.skc.edu/ (visited on 11/30/2015).

[25] *AESP14*. URL: http://www.aer.ita.br/~aesp14/.

[26] *AMSAT FOX Project*. URL: http://www.amsat.org/?page_id=1113 (visited on 11/28/2015).

[27] *LilaSat Satellites*. URL: http://lilacsat.hit.edu.cn/?page_id=257 (visited on 11/30/2015).

[28] *DeorbitSail*. URL: http://amsat-uk.org/satellites/telemetry/deorbitsail/ (visited on 11/30/2015).

[29] *PolyITAN-1*. URL: http://www.qsl.net/py4zbz/satelite.htm (visited on 12/02/2015).

[30] *SOMP - Students Oxygen Measurement Satellite*. URL: https://web.archive.org/web/20130929121727/http://phpweb.tu-dresden.de/stard/SOMP/?page_id=19%5C&lang=en (visited on 11/30/2015).

[31] *SOMP - Browser Decoder*. URL: http://phpweb.tu-dresden.de/stard/SOMP/beacon/Decoder-Javascript/decode.html (visited on 11/30/2015).

[32] *FUNcube Wereshouse*. URL: http://warehouse.funcube.org.uk (visited on 12/02/2015).

[33] *Student Nanosat VELOX-I (AMSAT Page)*. URL: https://web.archive.org/web/20150907170728/http://amsat-uk.org/2012/04/22/student-nanosat-velox-i/ (visited on 12/02/2015).

[34] *Student Nanosat VELOX-I (eoPortal Page)*. URL: https://directory.eoportal.org/web/eoportal/satellite-missions/v-w-x-y-z/velox-1 (visited on 12/02/2015).

[35] *FUNcube Homepage*. URL: https://web.archive.org/web/20150906000712/http://funcube.org.uk/ (visited on 12/02/2015).

[36] *FUNcube Dashboard - Installation and Operating Guidance Notes*. download.funcube.org.uk/gg¡sub¿N¡/sub¿otes¡ 6.pdf.

[37] Richard Kissel. *Glossary of Key Information Security Terms*. Tech. rep. NIST IR 7298r2. National Institute of Standards and Technology, May 2013. DOI: 10.6028/NIST.IR.7298r2. URL: http://nvlpubs.nist.gov/nistpubs/ir/2013/NIST.IR.7298r2.pdf (visited on 04/14/2017).

[38] *Session Management Cheat Sheet*. Jan. 2016. URL: https://www.owasp.org/index.php/Session_Management_Cheat_Sheet (visited on 04/14/2017).

[39] YongBin Zhou and DengGuo Feng. "Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing." In: *IACR Cryptology ePrint Archive* 2005 (2005), p. 388. URL: http://csrc.nist.gov/groups/STM/cmvp/documents/fips140-3/physec/papers/physecpaper19.pdf (visited on 04/14/2017).

[40] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: The Memory-Hard Function for Password Hashing and Other Applications". In: (Dec. 2015), p. 18. URL: https://password-hashing.net/argon2-specs.pdf.

[41] Dinei Florêncio, Cormac Herley, and Paul C. Van Oorschot. "An Administrator's Guide to Internet Password Research." In: *LISA*. 2014, pp. 35–52. URL: https://www.usenix.org/system/files/conference/lisa14/lisa14-paper-florencio.pdf (visited on 04/14/2017).

[42] Tim Polk, Kerry McKay, and Santosh Chokhani. *Guidelines for the Selection, Configuration, and Use of Transport Layer Security (TLS) Implementations*. Tech. rep. NIST SP 800-52r1. National Institute of Standards and Technology, Apr. 2014. DOI: 10.6028/NIST.SP.800-52r1. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r1.pdf (visited on 05/11/2017).

[43] Eric Rescorla et al. *Transport Layer Security (TLS) Renegotiation Indication Extension*. Tech. rep. 2010. URL: https://www.rfc-editor.org/info/rfc5746 (visited on 05/11/2017).

[44] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Keying Hash Functions for Message Authentication". In: Springer-Verlag, 1996, pp. 1–15.

[45] *NIST Policy on Hash Functions*. URL: https://beta.csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions (visited on 04/17/2017).

[46] Marc Stevens et al. *The First Collision for Full SHA-1*. Tech. rep. Cryptology ePrint Archive, Report 2017/190, 2017.

[47] Quynh H. Dang. *Secure Hash Standard*. Tech. rep. NIST FIPS 180-4. National Institute of Standards and Technology, July 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (visited on 04/17/2017).

[48] William Stallings. *Network Security Essentials: Applications and Standards*. 4th ed. OCLC: ocn504276173. Boston: Prentice Hall, 2011. ISBN: 978-0-13-610805-4.

[49] Gerard J. Holzmann. “The Power of 10: Rules for Developing Safety-Critical Code”. In: *Computer* 39.6 (2006), pp. 95–99. URL: http://ieeexplore.ieee.org/abstract/document/1642624/ (visited on 04/26/2017).

[50] Joao Andre Henriques Ferreira. “ISTNanosat-1 Heart”. In: (2012). URL: https://fenix.tecnico.ulisboa.pt/downloadFile/395144741162/EA_JFerreira_vFinal.pdf (visited on 04/25/2017).

[51] *NASA Software Safety Guidebook (ASA-GB-8719.13)*. Mar. 2004.

[52] Ruben Afonso. *Sistema de Gestão e Determinação de Atitude Do ISTNanosat-1*. Oct. 2016.

Appendix A

Appendix A - clang-tidy.sh

```
#!/usr/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

SRC="${DIR}/../../src/c"
BUILD="${DIR}/../../build"

black_list=(
    '-clang-analyzer-alpha.*'
    '-clang-analyzer-cplusplus*'
    '-clang-analyzer-osx*'
    '-cppcoreguidelines*'
    '-llvm-header-guard'
    '-llvm-include-order'
    '-misc-macro-parentheses'
    '-modernize*'
)

checks="-checks=*"
for c in "${black_list[@]}"; do
    checks+=",${c}"
done

clang-tidy "$checks" '-header-filter=.*' -p "${BUILD}" "${SRC}"/**/*.[c,h]
```

Listing A.1: clang-tidy.sh