UPC
UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Telecomunicació
i Aeroespacial de Castelldefels

i2cat

# BACHELOR DEGREE THESIS

**TFG TITLE: Virtual Satellite Network Simulator (VSNeS) - A Novel Engine to Evaluate Satellite Networks Over Virtual Infrastructure, and Networks**

**DEGREE: Bachelor degrees in Aerospace Systems Engineering and Telecommunications Sytems Engineering**

**AUTHOR: Uriel López Fernández**

**ADVISOR: Joan A. Ruiz-de-Azua**

**SUPERVISOR: David Rincón Rivera**

**DATE: October 18, 2022**

**Títol:** Virtual Satellite Network Simulator (VSNeS) - Un nou programa per a analitzar les xarxes de satèl·lits en infraestructures i xarxes virtualizadas
**Autor:** Uriel López Fernández

**Director:** Joan A. Ruiz-de-Azua

**Supervisor:** David Rincón Rivera

**Data:** 18 d'octubre de 2022

## Resum

L'espai ha estat poblat per una àmplia gamma de sistemes de satèl·lit d'entitats espacials tant governamentals com privades. Els satèl·lits han estat dissenyats per proporcionar un disseny personalitzat que realitza una missió específica. No obstant això, les noves demandes d'usuari requereixen una cobertura global, un temps de revisita baix i un servei omnipresent. Durant els últims anys s'ha debatut de manera persistent la possibilitat d'integrar la infraestructura en òrbita per a donar suport als sistemes de comunicacions actuals. En concret, el concepte de desplegar xarxes compostes d'aeronaus i naus espacials (creant les anomenades Non-Terrestrial Networks), ha sorgit com una arquitectura potencial per satisfer aquesta nova demanda. Aquest concepte nou ha permès investigar tecnologies de xarxa en infraestructura espacial. Per exemple, aquest és el cas de Software Defined Satellite, que té com a objectiu la gestió de la infraestructura en òrbita mitjançant l'ús de tècniques de Software-Defined Network.

Aquests conceptes nous plantegen múltiples reptes que han d'abordar els desenvolupaments expecifics. De la mateixa manera, es necessiten equips específics i entorns de simulació que donin suport. Actualment, els emuladors de xarxa de satèl·lit de codi obert tenen certes limitacions o no són fàcilment accessibles. Aquest projecte té com a objectiu presentar Virtual Satellite Network Simulator, un nou motor de simulació capaç de representar satèl·lits, així com nodes terrestres en màquines virtuals, i desplegar una xarxa virtual que representa els efectes del canal i la seva dinàmica.

VSNeS s'ha generat a partir de diferents mòduls, que gràcies al treball conjunt és capaç de generar la virtualització. En primer lloc, s'ha desenvolupat un programa Python3, que funciona com a gestor, i és responsable d'executar la resta dels mòduls d'acord amb l'escenari virtualitzat. A més, Kernel-based Virtual Machine, s'ha utilitzat per l'execució de les màquines virtuals. La gestió del canal es realitza des de l'emulador NetEm. I finalment, una interfície gràfica d'usuari és lliurada per Cesium.

Aquesta document presenta formalment un disseny preliminar amb les passes essencials per seleccionar cada tecnologia. També es discuteix el disseny de la xarxa que s'utilitza. Més endavant es mostren diferents proves per verificar el correcte funcionament de l'eina. A més de demostrar com el prototip final afecta el rendiment del host.

El programa s'ha provat amb els protocols TCP, UDP i ICMP. Això ens va permetre verificar el correcte funcionament del programa, comprovar els retards i les pèrdues del canal. A més, es demostra com el protocol TCP no és funcional amb satèl·lits geoestacionaris, a causa de l'alta latència, causada per les grans distàncies.

**Title :** Virtual Satellite Network Simulator (VSNeS) - A Novel Engine to Evaluate Satellite Networks Over Virtual Infrastructure and Networks
**Author:** Uriel López Fernández

**Advisor:** Joan A. Ruiz-de-Azua

**Supervisor:** David Rincón Rivera

**Date:** October 18, 2022

## Overview

Space has been populated by a wide range of satellite systems from governmental and private space entities. Monolithic satellites have been ruling it by providing a custom design that accomplishes a specific mission. Nevertheless, novel user demands emerged have required global coverage, low revisit time, and ubiquitous service. The possibility to integrate in-orbit infrastructure to support current communications systems has been discussed persistently during the last years. Specifically, the concept of deploying networks composed of aircraft and spacecraft (creating the so-called Non-Terrestrial Networks), has emerged as a potential architecture to satisfy this new demand. This novel concept has enabled to investigate mobile technologies in space infrastructure. For example, this is the case of the Software-Defined Satellite, which aims at managing in-orbit infrastructure by using Software-Defined Network techniques.

These novel concepts pose multiple challenges which dedicated developments shall address. Likewise, specific equipment and simulation environments shall support them. Currently, open source satellite network emulators have certain limitations or are not easily accessible. This project aims at presenting the Virtual Satellite Network Simulator, a novel simulation engine capable to represent satellites as well as ground nodes in virtual machines and deploy a virtual network that depicts the channel effects and dynamics.

VSNeS has been generated from different modules, that thanks to the joint work is able to generate the virtualization. First of all, a Python3 program has been developed, which works as a manager and is responsible for running the rest of the modules according to the virtualized scenario. Furthermore, Kernel-based Virtual Machine has been implemented for the execution of the virtual machines. The channel management is done with the NetEm emulator. Finally, a graphical user interface is delivered by Cesium.

This dissertation presents formally a preliminary design with the essential steps to select each technology. Then, the networking design is also discussed. Different tests are also shown in order to verify the correct functioning of the tool. In addition, tests about the performance of the final release have been performed.

The program has been tested with the following protocols in different realistic scenarios: TCP, UDP, and ICMP. This allowed us to verify the correct operation of the program, checking the delays and channel losses. Moreover, it is empirically demonstrated that some protocols are not functional for geostationary satellites, due to the long latency caused by the large distances.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# INTRODUCTION

The New Space ecosystem is a growing field which makes more accessible to private companies and public entities the space. As a result, a large number of satellites for Earth observation, communications, and navigation are being launched. However, novel user demands require global coverage, low revisit time, and ubiquitous service. With the envision to satisfy this new demand, space systems evolved to more distributed architectures, so-called Distributed Satellite Systems (DSS) [1], this encompasses sets of satellites that work with the same functionalities and interact to fulfill an objective, instead of individual satellites with an independent purpose.

Among the different initiatives in this evolving space trend [2], the possibility to integrate in-orbit infrastructure to support current mobile system is being discussed persistently during the last years. These arguments have materialized, with the effort from the members of the 3rd Generation Partnership Project (3GPP), into the concept of Non-Terrestrial Networks (NTN) [3]. The deployment of these NTN poses considerable technological challenges that needs to be addressed in the future. Among them, the management of the network resources is a challenge that is being currently investigated. Specifically, the concept of Software Defined Networks (SDN) [4] promotes the allocation of infrastructure resources for Virtual Network Functions (VNF) in order to generate virtual networks in the space infrastructures (like a cloud). However, this is only one of the points of research in the field of satellite communications.

Satellite communications are affected by channel characteristics closely related to their relative motion in regard to the Earth, meaning that it is not possible to work with the same technologies used in terrestrial protocols. The dynamics of satellites generates a constantly changing network, which is affected by delay and temporal contacts. The novel NTN functionalities have to take into account the previous characteristics.

The design and implementation of these new technologies require a progressive verification procedure, finalizing it on the in-orbit validation. This is the reason why the deployment of satellite networks entails a high cost, meaning it is not feasible to develop constellations for specific tests. Instead, it would be necessary to access to topologies intended for testing, which have specific configurations. Furthermore, there is no control over the scenario, that is defined by external conditions such as the weather or the competition to access satellite resources. This means that specific tests cannot be carried out and scenarios are difficult to replicate. This type of testing is intended for a final phase of development, due to its costs and the need to access a satellite network.

An alternative approach and part of the development, is the virtualization of a satellite network. This allows the creation of topologies and their testing under specific characteristics. Novel developments can be evaluated by using simulators and emulators. In a simulated environment, all parts of the system are based on models that aim to replicate the network that will be represented. In the case of emulators, instead, all end host operating systems, end-to-end protocols, and applications are real. Only the network (the communication links) is simulated. This implies a more advanced stage of analysis of the novel technologies than in the simulators. Currently, there are several simulation and emulator platforms that meet very different requirements.

Despite the existence of several software focused on the virtualization of satellite networks, all of them are limited by some factors or are not easily accessible. Owing to these limitations, this work presents a simulation engine that uses network virtualization to interconnect Virtual Machines (VM) that represents satellites and ground nodes. This engine, known as VSNeS, is able to build different machine architectures for satellite and ground devices, making the represented scenario more realistic. VSNeS has been developed in Python3 and is prepared to be run only on an Ubuntu machine using NetEm, QEMU/KVM, and Cesium. The engine is conceived with a high-flexible configuration and a satellite dynamics tool to highly represent orbits and dynamics. The concept, design and results of this engine are presented in this thesis. Additionally, the VSNeS is an active open-source project that is expected to realize contributions from the research community.

This report presents basic concepts to understand the motivation and the necessity of the project. Furthermore, we will expose the different plans that have been pursued to reach the final design. The thesis also includes a description of the structure and operation of the software. Moreover, it contains a set of tests that demonstrate the software runs accurately. Finally, VSNeS is tested in three scenarios to observe how different orbits interact with TCP, UDP, and ICMP protocols.

The thesis is divided into seven chapters as follows: Chapter 1 presents important concepts to the development of this report. Chapter 2 defines the project and shows the objective and the work packages that have been created to complete the project and how we have managed the schedule. Chapter 3 explains the initial idea of VSNeS in detail, it lists the requirements and compares them to the current technologies. Chapter 4 is concerned with the changes regarding the initial approach and the final implementation. Chapter 5 is the main chapter, as it explains the current version of VSNeS, its structure and how it works. Chapter 6 describes the verification of the program, it specifies some tests that check if the software does what is expected to, and its performance and limitations. In Chapter 7, we build realistic scenarios to show the possibilities of the software. The final part will be focused on the conclusions and some suggestions for future research lines.

# CHAPTER 1. STATE OF THE ART

This chapter presents the background knowledge necessary to understand both the technical concepts and the project motivation. First, a brief introduction to satellite communications and new tools for satellite network management is given. Then, a more specific view of satellite dynamics is introduced in order to understand their effects on communications and to know how to characterize their orbits. Subsequently, an analysis of satellite network emulators is made. In this section, we give an overview of what they are and what programs have been developed up to date. The last section of this chapter focuses on the presentation of virtual machines and their operation, showing why they are intended to be one of the key concepts within our virtual system.

## 1.1. Non-Terrestrial Networks and Software-Defined Network over Satellites

The demand for communication resources is growing due to an increase in the number of smart devices, data flow and number of end users [5]. Non-Terrestrial Networks (NTNs) are an effective solution to complement terrestrial networks providing services over uncovered or under-served geographical areas. NTN is defined by a network with elements overhead in the form of spaceborne or airborne vehicles. Spaceborne includes principally Low Earth Orbit (LEO), Medium Earth Orbit (MEO) and Geostationary Equatorial Orbit (GEO) satellites, while airbone includes High Altitude Platform Stations (HAPS) and Unmanned Aircraft Systems (UAS).

The main benefits of incorporating NTNs are extending coverage, supporting of terrestrial services in critical case and providing reliability and resilience. NTNs provide coverage to areas without any terrestrial coverage like rural areas or the sea. Traditional networks may become inoperative after a natural disaster or other situations. In these cases, satellites can give support to the critical communications. In addition, advances in NTNs will allow them to play an important role in 5G and future 6G applications [6].

As shown in Figure 1.1, the satellite network architecture [7] is composed of the space segment, ground segment, control and management segment, and user segment. The space segment is composed of all the satellites of the network, which can have up-links and down-links or also Inter-Satellite Links (ISL) [8]. The ground segment consists of two types of receivers. First, the satellite gateway (GW) provide access to an external network. On the other hand, the satellite terminal (ST) is responsible for providing service to the end user. The control and management segment is composed of network control centers (NCCs) and network management centers (NMCs). They provide real-time control and management functions for satellite systems. NCCs, NMCs, and GWs are usually co-located at the same site, which is called a satellite hub. Finally, we find the user segment, which groups all end users that use satellite resources, whether they are fixed or mobile.

Traditionally, satellites are part of an ad hoc architecture [7]. This creates a challenge for integrating NTNs with terrestrial networks. Software-Defined Network (SDN) is a promising technique to be applied to the integration of NTNs, offering programmability, flexibility, reconfigurability, and logical centralization. Thereby, increasing network resource utilization, simplifies network management and reduces operating cost. SDN logically separates the

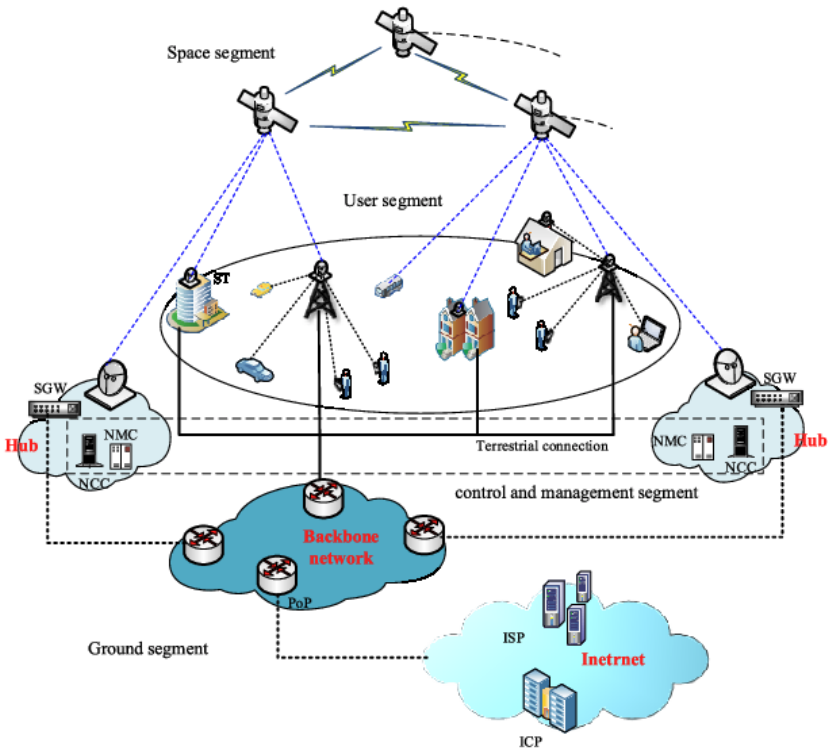


Figure 1.1: Satellite network architecture, from [7].

control and data planes of networks, thus allowing the control plane to be run on software. Thus consists on the creation and control of virtual networks or hardware by software. Network Functions Virtualization (NFV) is the replacement of network appliance hardware with virtual machines. Its function is to abstract network functionalities from the hardware by virtualizing them. In this way NFV offers resources that SDN can control and orchestrate for specific uses. As a general rule, SDN runs its functions from a virtual machine within the network. Furthermore, the adoption of SDN/NFV into the satellite networks enables towards more flexible and agile integration of NTNs and Terrestrial Networks (TNs).

The use of SDN/NFV over satellites is a process in full development that involves different difficulties. The authors of [9] show different scenarios where it can be implemented. In addition, solutions are proposed to the different challenges that arise in [7]. Another use case is presented in [10], where they solve the controller placement problem given in SDN-based satellite networks by the Static Placement with Dynamic Assignment (SPDA) method. These research initiatives needs to be supported with realistic and accurate equipment.

## 1.2. Satellite dynamics

The dynamics of the satellites change in relation to the type of orbit they follow. This leads to the need to study the different orbits and how each one affects the communication channels. In addition, for the correct representation of these orbits, it is important to define tools that facilitate the management of the satellite position. For this, it is important to understand the different coordinate systems and the relationships between them. Moreover, it is necessary to understand which parameters define an orbit and how to obtain them.

### 1.2.1. Orbit Regions

Knowledge of the satellite orbits is fundamental for the right configuration of the network topology. The principal effects that satellite movement produce are delays, Doppler frequency shift, and attenuation of the signal [11]. These channel effects have a different impact depending on the orbit type. Satellite orbits around the Earth are classified by regions according to the altitude. Specifically, there are LEO, MEO, and GEO [12], like Figure 1.2 is showing.

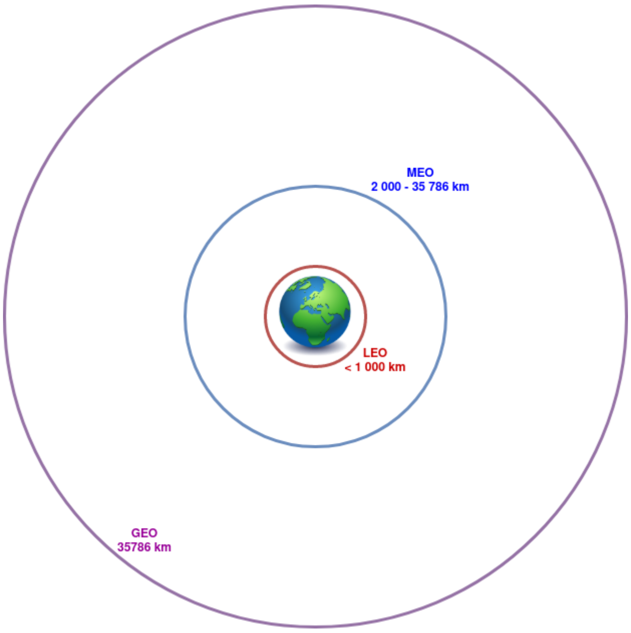


Figure 1.2: Representation of the GEO, MEO, and LEO orbit regions.

LEO is an orbit relatively close to Earth's surface. The typical altitude is greater than 160 km, but less than 1000 km. In orbit, the period is completely related with the semi-major axis. There are LEO satellites that take approximately 90 minutes of period. The high speed produces that the ground track does not follow a particular path above the Earth's surface. Satellites in LEO have a small field of view and so can observe and communicate with a section of the Earth during a short lapse of time. This produces temporal contacts. On the other hand, it requires the lowest amount of energy for satellite placement and simpler radiofrequency front-end and antennas aboard, because the signal losses are lower than in the other orbits. In addition, LEO satellites provides high-bandwidth and low-communication latency. Actually, communications satellites in LEO often work as part of a large constellation, to give constant coverage and create a network. Although this is the development trend, offering high coverage and reduced latency, the implementation of these networks involves many challenges.

GEO satellites have the same period as the Earth, one sidereal day, approximately 23h 56 min 4.0905 s and circle Earth above the equator from west to east following Earth's rotation. This makes satellites in GEO appear to be 'stationary' over a fixed position on the Earth surface. Their altitude is 36786 km. Satellites in GEO cover a large range of Earth, that three equally-spaced satellites can provide near global coverage, although the poles remain uncovered.

The most powerful characteristic is the simplicity to configure the Earth segment, for example an antenna on Earth can be fixed to always stay pointed towards that satellite without moving. Also, no relevant Doppler effects appear and the delay is constant (in mean) for the fixed ground stations. Instead, the channel is limited by the required power and the large delay. The theoretical Round Trip Time (RTT) from this orbit is 240 ms or even 275 ms for ground stations far away from the equator. These RTTs pose major limitations on transmissions such as voice or data flows based on Automatic Repeat-reQuest (ARQ) techniques (e.g. TCP).

Although there are more orbits, the last one on which make focus on is MEO. The satellites with these orbit have an altitude between 2000 km and 35786 km. It is very commonly used by navigation satellites, The period of Global Navigation Satellite System (GNSS) satellites is typically around the half sidereal day. The theoretical RTT of these orbit is between 50 ms to 150 ms and the launch cost is higher than in the LEO. Instead, the necessity of handling handovers and Doppler effect are present, but in less measure than in LEO cases.

### 1.2.2. Keplerian Elements

An orbit is the trajectory that an object follows around another object under the influence of attraction between them, the gravitational force. A perfect orbit , i.e. only influenced by the gravitational forces of the two bodies when the mass of one of them is negligible, can be defined with the Keplerian elements. In addition, Kepler defined a set of laws that these orbits follow. Talking about the movement of a satellite around the Earth, the laws are:

1. Satellite orbits describe an ellipsoid with the Earth in one of the focal points.
2. The line between a satellite and the Earth's center sweeps equal areas at equal time intervals.
3. The squared of the orbital period is proportional to the cube of the cube of the semi-major axis of the ellipse, as shown in Equation 1.1

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \tag{1.1}$$

Keplerian Elements define the shape of the orbit and its orientation. They are 3 elements that defines the form ($a$, $e$, and $v$) and other three to define the orientation ($i$, $\Omega$, and $\omega$).

$a$ Semi-major axis. It is the longest semidiameter of the ellipse and thus runs from the center, through a focus and to the perimeter.

$e$ Eccentricity. It is the ratio between the semi-minor axis and the semi-major axis, as shown in Equation 1.2 where b is the semi-minor axis. In the case that the eccentricity is equals to 0 the orbit is circular.

$$e = \frac{b}{a} \tag{1.2}$$

- $v$ True anomaly. It is an angular parameter that defines the position of a body in the orbit, in respect to the perigee. True anomaly extends from 0 degrees to 360 degrees.

- $i$ Orbital inclination. It is expressed as the angle between a reference plane (Equator) and the orbital plane or axis of direction of the orbiting object. Orbital inclination is usually restricted to the range 0-180 degrees.

- $\Omega$ Right ascension of the ascending node. It is the angle measured between the Vernal Equinox direction and the ascending node, the point where a satellite crosses the equator moving upward. It is composed by values between 0 and 360 degrees.

- $\omega$ Argument of the perigee. It is defined as the angle measured from the ascending node to the perigee, along the satellite's direction. Its values can be from 0 to 360 degrees.

Figure 1.3 shows how the Keplerian elements define an orbit. P1 represents the orbital plane while P2 represents the reference plane, in the case of Earth's satellites, it is the equatorial plane.

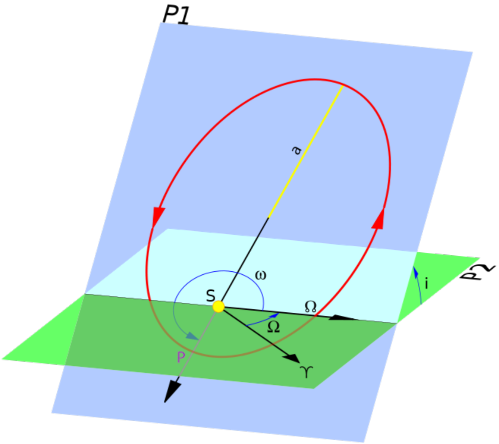


Figure 1.3: Keplerian Elements, from [13].

### 1.2.3. Coordinate Systems

The position of a satellite is understood as the separation between a given point and the reference axes. The coordinate systems define their own reference axes and their type of measure (distance or angles). Several coordinate systems are used in the study of the Earth satellites and other celestial bodies [14] [15]. In regard to the artificial satellites, the most suitable coordinate systems are the ones which have the center of the Earth as the origin.

**Earth-Centered Inertial (ECI)** [14] is a geocentric-equatorial coordinate system, as shown in Figure 1.4. Its fundamental plane is the equator, where there are x-axis and y-axis. The positive x-axis points in the vernal equinox direction. Vernal equinox is the line from the Earth to the Sun on the first day of spring, March 21st. The z-axis points in the direction of the North Pole. The x, y, z axis are not fixed to the Earth, so the system is non-rotating with respect to the stars and the Earth turns relatively to it. The characteristics of ECI make it an ideal system to represent all kind of celestial bodies, but not suitable to define the position in the Earth's surface. When we represent artificial satellite orbits in the ECI system, an orbit is on a static plane. These makes easier to define the orbit, we only need to know the plane and the shape of it. The positions will be repeated periodically.

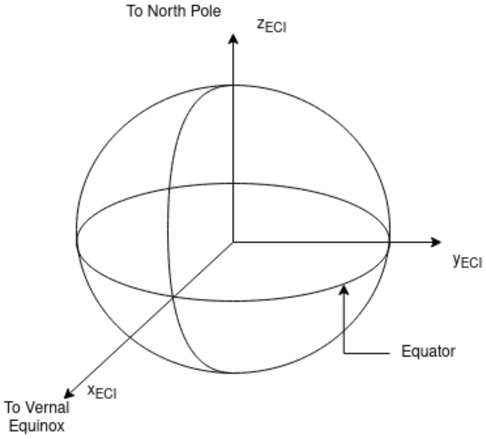


Figure 1.4: ECI Coordinate System.

The next coordinate system to be analyzed is **Earth-Centered, Earth-Fixed (ECEF)** [15] coordinate system, shown in Figure 1.5. Its fundamental plane is the equator and the positive x-axis points in the Greenwich meridian. The z-axis points in the direction of the North Pole, like in ECI case. Its main application is in GNSS, because ECEF defines the position of a point in the Earth's surface with a static value and it works with distance. This makes it possible to create an equation system with the pseudoranges and the positions.

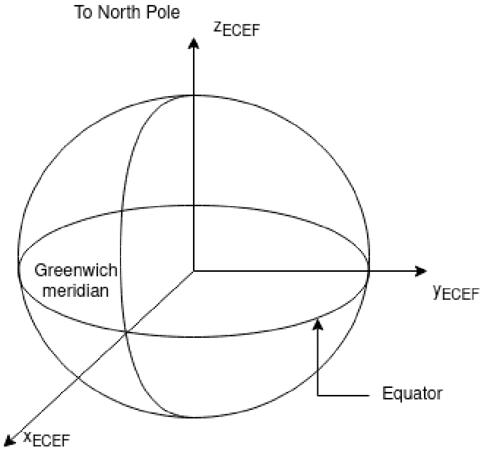


Figure 1.5: ECEF Coordinate System.

Other form to define the position above the Earth's surface is the **Geographic Coordinate System** (**GCS**) [14] [15]. It is composed by two angular coordinates (latitude, longitude) and altitude above the Earth's surface. As we can see in Figure 1.6, the latitude is the parameter which defines the position with respect to the North and South, with angles between $\pm 90^\circ$ and the equator as the origin. The longitude shows the deviation to the East or West. Its origin is in the Greenwich meridian and its values are between $\pm 180^\circ$. This system is the best to represent a ground path. Moreover, it can be used with different reference ellipsoids, the most common one being World Geodetic System 84 (WGS-84).

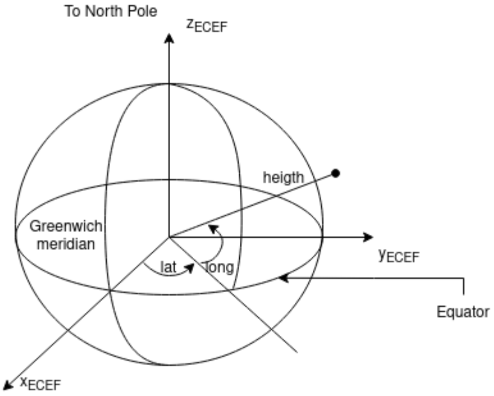


Figure 1.6: Geographic Coordinate System.

The last system that is worth talking about is **Azimuth-Elevation Coordinate System** [14]. In this case, the origin is at an observer standing at a particular point on the Earth's surface. The fundamental plane is the local horizon. The system is composed by two angular coordinates (azimuth, elevation) and the distance from the observer to the object (range). The azimuth is an angle from the North to the object, measured in the fundamental plane. Elevation is an angle between the horizon plane and the object's direction, with values between $\pm 90^\circ$. Figure 1.7 shows the coordinate system in detail. These coordinate system is useful to compute the visibility of the object in relation to the elevation, it must be greater than 0 deg.

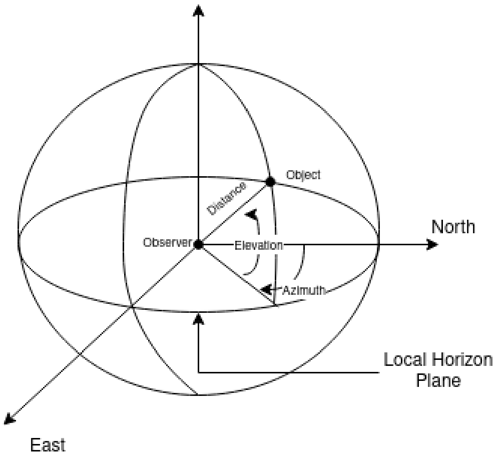


Figure 1.7: Azimuth-Elevation Coordinate System.

### 1.2.4. Two Line Elements (TLE)

**Two Line Elements** (**TLE**) [16] is the standard format to represent the Keplerian elements, which are the parameters that characterize an orbit. TLEs have two lines with 69 characters in each one, which contain all the information to compute the satellite position and velocity at a given moment. Undoubtedly, it has a specific format determined by lines and columns. Below we can see the distribution of the columns and the type of characters allowed in each one. Columns with a space cannot have other characters. Columns with an N can have any number 0-9 or, in some cases, a space. Columns with an A can have any character A-Z or a space. Columns with a '+' can have either a plus sign, a minus sign, or a space and columns with a '-' can have either a plus or minus sign. Column 1 of each line of TLE set indicates the line number. This format is presented as follows:

```
1 NNNNNC NNNNNAAA NNNNN.NNNNNNNN +.NNNNNNNN +NNNNN-N +NNNNN-N N NNNNN
2 NNNNN NNN.NNNN NNN.NNNN NNNNNNN NNN.NNNN NNN.NNNN NN.NNNNNNNNNNNNNN
```

Tables 1.1 and 1.2 define each of the individual fields for lines 1 and 2, respectively.

| Field | Columns | Description |
|---|---|---|
| 1.1 | 01 | Line Number of Element Data |
| 1.2 | 03-07 | Satellite Number |
| 1.3 | 08 | Classification (U: unclassified, C: classified, S: secret) |
| 1.4 | 10-11 | International Designator (Last two digits of launch year) |
| 1.5 | 12-14 | International Designator (Launch number of the year) |
| 1.6 | 15-17 | International Designator (Piece of the launch) |
| 1.7 | 19-20 | Epoch Year (Last two digits of year) |
| 1.8 | 21-32 | Epoch (Day of the year and fractional portion of the day) |
| 1.9 | 34-42 | First Time Derivative of the Mean Motion |
| 1.10 | 45-52 | Second Time Derivative of Mean Motion (decimal point assumed) |
| 1.11 | 54-61 | BSTAR drag term (decimal point assumed) |
| 1.12 | 63 | Ephemeris type |
| 1.13 | 65-68 | Element number |
| 1.14 | 69 | Checksum (Modulo 10) |

Table 1.1: Two-Line Element Set Format Definition, Line 1, from [16].

In the first line, only three fields are useful to our interest. In the first point, the field 1.1 indicates the satellite number, a unique identifier by North American Aerospace Defense Command (NORAD) for each Earth-orbiting artificial satellite in their Satellite Catalog (SATCAT). It allows us to define an ID to identify all the elements in the simulation.

The fields 1.7 and 1.8 are the next key points. These two fields together define the reference time for the element set and are jointly referred to as the epoch. Field 1.7 is the two-digit year (which will be explained further along the project) and field 1.8 is the day of that year. The epoch defines the time to which all the time-varying fields in the element set are referenced.

Line 2 contains the data needed to propagate an orbit, this means we will come across the most interesting datum. First, we find the satellite number in field 2.1 again, which can be used as a verification of the parameters. Next, we encounter the Keplerian elements,

| Field | Columns | Description |
|---|---|---|
| 2.1 | 01 | Line Number of Element Data |
| 2.2 | 03-07 | Satellite Number |
| 2.3 | 09-16 | Inclination [Degrees] |
| 2.4 | 18-25 | Right Ascension of the Ascending Node [Degrees] |
| 2.5 | 27-33 | Eccentricity (decimal point assumed) |
| 2.6 | 35-42 | Argument of Perigee [Degrees] |
| 2.7 | 44-51 | Mean Anomaly [Degrees] |
| 2.8 | 53-63 | Mean Motion [Revs per day] |
| 2.9 | 64-68 | Revolution number at epoch [Revs] |
| 2.10 | 69 | Checksum (Modulo 10) |

Table 1.2: Two-Line Element Set Format Definition, Line 2, from [16].

field 2.3, 2.4, 2.6, and 2.7 are defined in degrees between 0 and 360, except for 2.3 which only reaches 180 degrees. Additionally, field 2.5, which also defines a Keplerian element, represents the eccentricity multiplied by a factor of $10^7$. Finally, the Mean Motion, field 2.8, is the last parameter that we need to calculate the orbits.

All these data gives the possibility to know the position of the satellite at any time with the help of an orbit propagator [17]. The basic one is the Two-Body propagator, also known as Keplerian motion. The propagator considers only the force of the gravity from the Earth and follows the Newton's laws. Another one, with more precision, is the Simplified General Perturbations Satellite Orbit Model 4 (SGP4). It considers secular and periodic variations due to Earth oblateness, solar and lunar gravitational effects, gravitational resonance results, and orbital decay using a simple drag model.

## 1.3. Satellite Network Emulators

### 1.3.1. Introduction To Satellite Simulation and Emulation

Network emulation is a technique that implements hardware or software for testing the performance of real applications over a virtual network. It usually permits the creation of a specific topology that introduces delay, errors, drop packets and others in the different connections of the network. Network emulator tests the behavior of a network from a lab by running a software in a computer or a virtual machine or by a dedicated device.

Emulation differs from simulation in that a network emulator appears to be a network; end-systems, such as computers can be attached to the emulator and will behave as if they are attached to a network. While network simulators do not have connection with external machines. All parts of the system are based on models that try to replicate the network that you want to represent.

There are currently various open source simulators that replicate different effects in the field of satellite communications, such as Satellite Network Simulator 3 (SNS3) [18] and Open Source Satellite Simulator (OS3) [19] [20] [21]. On the one hand, SNS3 belongs to the net4sat software development platform, developed by Centre National d'Études Spatiales (CNES), which is an extension of Network Simulator 3 (ns-3) [22]. On the other hand, OS3 is based on Objective Modular Network Testbed in C++ (OMNeT++) [23] and it is also able to automatically import real satellite tracks and weather data to simulate conditions at a certain point in the past or in the future and offer powerful visualization.

However, in our case we do not focus on simulators, but we go into detail in the world of emulators. This is because one of the key approaches of the project is the use of VMs as part of the test bed, so the engine must be able to interact with the real world.

### 1.3.2. Survey of Satellite Network Emulators

There are currently different emulators of communication networks with satellites. Each one offers a different approach and qualities that make it stand out in different fields.

**OpenSAND** [24] is a continuation of the Platine project. This platform was initially developed by Thales Alenia Space and promoted by CNES and it is another project of net4sat. OpenSAND emulates an end-to-end satellite communication system over standard Digital Video Broadcasting - Return Channel via Satellite (DVB-RCS) and Digital Video Broadcasting – Second Generation (DVB-S2). It supports IPv4, IPv6, and Ethernet connectivity with which it can be interconnected with real equipment, as VM. Furthermore, it consists of different configurations where the delay can be modified to simulate different orbits. However, it is not possible to replicate the movement they produce. Its objectives are to provide a tool to validate access and network innovative functionalities with measurement points for performance evaluation, interconnection with real terrestrial networks, as well as applications for demonstration purposes. The source codes of OpenSAND components is distributed “as is” under the terms and conditions of the GNU GPLv3 license or the GNU LGPLv3 license.

**Satellite Constellation Network Emulator (SCNE)** is an European Space Agency (ESA) development project [25], [26]. SCNE aims at offering a simulation tool that allows the study of the effect on end-to-end Quality of Service (QoS) of routing solutions, security, new transport protocols, caching mechanisms or payload functionalities over LEO constellations. It offers simulations of physical layer aspects and higher layer protocols. SCNE shall design satellite constellations, study ILS routing algorithms, different transport layer protocols and mobility management schemes.

SCNE consists of three components: Systems Tool Kit (STK) [27], ns-3 [22], and SCNE orchestrator. STK is a 3D modeling software, developed by Ansys Government Initiatives (AGI), for complex systems, including ground vehicles, satellites, unmanned aerial vehicles, and more that offers generation of satellite movement, visibility, and link budget for SCNE. Ns-3 is an open source software maintained by a worldwide community. It simulates the application traffic and related communication protocols, on top of the satellite constellation. Finally, the SCNE orchestrator manages the scenarios and the data transfer between the other programs. One of its strong points is the high number of terminals that can be simulated at the same time. The default demonstration scenario is 24 hours long and consists of 288 satellites, 10 gateways (GW), and 50 user terminals (UT). We have only been able to find the planning for this project, however the final work could not be obtained.

**SATIP6 satellite emulation platform** [28], [29] is a project designed by Oliver Alphand, Pascal Berthou, and Thierry Gayraud with the support of entities like Laboratory for Analysis and Architecture of Systems - French National Centre for Scientific Research (LAAS-CNRS). It works on connections with GEO satellites through IP connections over DVB-RCS in order to study QoS. The architectures of the scenarios are formed by the Satellite Emulator (SE), Satellite Terminals (ST), and Network Control Center (NCC). ST acts as routers and gives access to the external end users. SE represents a regenerative satellite.

The functionality of the software are the simulation of typical satellite bit errors and delay, implementation of Demand Assigned Multiple Access (DAMA) algorithms, dynamic address resolution protocol and IP QoS Package among others. The experimental platform to validate the project was formed by one computer for each component of the scenario that generated a hard configuration.

SATIP6 consists of the development of a test-bed rather than software and it is not possible to access the program.

**Mobility Satellite Emulation Testbed (MSET)** [30] was developed by the Department of Computer Science at the University of California, Santa Barbara. It was designed with four goals in mind: the test-bed must be based on commodity hardware and open source software. It must be highly-configurable and scalable to allow a wide range of scenarios. Likewise, it must offer high-fidelity with minimal emulation artifacts and it shall be compatible with the University of Utah Emulab test-bed.

MSET provides multibeam, multi-satellite, TDMA, and mobility functionality. MSET is capable of achieving the desired delay, loss, and jitter effects associated with a mobile satellite link.

**Scalable Cloud-Base LEO Satellite Constellation Simulator** [31] was developed by Karim Sobh, Khaled El-Ayat, Fady Morcos, and Amr El-Kadi with the Department of Computer Science and Engineering of the American University in Cairo. It used cloud IaaS virtual machines to simulate LEO satellites and ground stations distributed software.

The simulator is built up of a number of subsystems. Foremost, the Constellation Backend Database, a back-end MySQL database, which stores the constellation configuration where satellite and Ground stations are members. The Cloud Integration Module is used to import cloud VM configuration and network attributes into the simulator database. SGP4 is an open source C++ library that calculates the Cartesian coordinates of the satellite based on its TLE and time. Line-of-Sight (LoS) Daemon is responsible for calculating a LoS matrix between all satellites and ground stations in a constellation. Netfilter Kernel Module Extension is at the heart of the simulator, meaning that it is a hook into the Network IP layer. This module is located in an extra VM to the communication elements, where all the data is processed. Cesium GUI [32] is a client for simulating the Earth and objects rotating around it, as well as locations within. Finally, the Administration Console is a web application used to create constellations and launch the Cesium simulator GUI. This combination of subsystems offers a very complete implementation for the satellite constellation emulation, but the code or an implementation manual is not accessible.

**Common Open Research Emulator (CORE)** [33] was derived from the open source IMUNES project from the University of Zagreb. CORE has been released by Boeing to the open source community under the BSD license. It is a tool which builds virtual networks. As an emulator, CORE frames a representation of a real computer network that runs in real time, as opposed to simulation, where abstract models are used.

CORE uses different node types. CORE Nodes are the standard nodes which will have a directory uniquely created for them as a place to keep their files and mounted directories. Docker Nodes and LXC Nodesthat use predefined images and file systems. Ultimately, we can find the implementation of Physical Nodes, which includes physical computers and VM.

CORE works only on Linux, due to the way it performs emulation. To perform channel emulation, the software first generates network interfaces on the host for each Ethernet port of the simulated nodes. Afterwards, it applies the required characteristics on each of the generated interfaces. This is achieved by using the **NetEm** tool. NetEm [34] is an enhancement of the Linux traffic control facilities that provides network emulation functionality for testing protocols such as add delay, packet loss, duplication, and more other characteristics to packets outgoing from a selected network interface. Netem is controlled by the command line tool tc which is part of the iproute2 package of tools.

CORE implements a GUI that is used to draw nodes and network devices on a canvas, linking them together to create an emulated network session. Inside the options of view, the GUI has a 3D model that can represent the Earth. This works with the Scripted Display Tool (SDT) from Naval Research Laboratory (NRL) and is based on National Aeronautics and Space Administration's (NASA) Java-based WorldWind virtual globe software [35]. In addition, it offers easy measurements on all nodes with the incorporation of the Wireshark program. Moreover, it includes the generation of configuration files in format XML or IMN. Finally, it is important to highlight the possibility of introducing mobility scripts, which associate movements to wireless nodes.

CORE is typically used for network and protocol research, demonstrations, application, platform testing, evaluating networking scenarios, security studies and increasing the size of physical test networks. Although CORE is not a software designed entirely for simulating satellite networks, it offers a great number of facilities for its implementation. An example of this type of implementation, although not entirely related to satellites, is the one developed by NASA in the Delay-Tolerant Networking (DTN) Development/Deployment Kit (DevKit) [36] [37], which incorporates a set of pre-built scenarios to implement the Interplanetary Overly Network (ION).

Finally, we have considered Network **Network Emulator ONE (NE-ONE)**, a project developed by the company ITRINEGY [38]. It is the first emulator mentioned that is not free source, which offers two versions: Professional and Enterprise. Those are available in hardware or in virtual appliances, which is compatible with OpenStack. Both work with a web service.

This simulator is not a full satellite network simulator but includes LAN, WAN Wi-Fi, (A)DSL, Mobile, and satellite networks. The simulator combines the creation of networks in a canvas or in a 3D-GIS, allowing the study of the different networks and combining them. Regarding the emulation of satellite networks, the software focuses on latency, wavebands, intermittent connectivity, and jitter.

Each of the different emulators have features that make them of great interest and at the same time, they offer limitations to our objectives. Throughout Chapter 3 we compare in detail the features of the most remarkable tools with the requirements that we have considered relevant for our software. This comparison has led us to discard practically all the existing emulators.

## 1.4. Virtual Machines

A virtual machine is a piece of software that emulates a computer, due to the fact that its physical parts (like CPU, memory, network interface, and storage) are virtual. A VM makes possible two important concepts: On one hand, we can emulate a computer with a specific hardware architecture; on the other hand, we can have isolated machines on a single piece of hardware.

A hypervisor, sometimes called a virtual machine monitor, is a software that manages and runs VMs. This is accomplished by managing the hardware resources of a physical computer. The physical hardware, where a hypervisor runs, is named host, while the VMs are the guests. The hypervisor has access to host resources, which it distributes among the different guests. The guests work as if they were independent physical computers with specific resources. Existing hypervisors can be classified into two types, depending on the level on which they work:

- Type 1, also known as native or bare-metal hypervisor, is a hypervisor that works directly on the host hardware. The hypervisor directly manages the resources allocated to the guest operating systems.
- Type 2, also known as hosted hypervisor, is the one that works on the host operating system as an application. In this case, the hypervisor does not have direct access to the resources, as it is the operating system that accesses them.

VMs are attractive for several reasons. First, it is possible to run multiple operating systems simultaneously. In other words, the hypervisors make software installation easier than on a real machine. VMs are useful to test and for disaster recovery purposes, because they can be arbitrarily frozen, woken up, copied, backed up, and transported between hosts. Finally, virtualization can significantly reduce hardware and electricity costs. For instance, servers do not often use all its physical resources, but virtualization allows them to use the available resources in other virtual servers.

In research, virtualization allows to create a network of computer without many physical components or configuration and to make test of protocols or security without effects on the host. In our case, we use them to represent a satellite network, where the VMs represent satellites and ground nodes.

There are a variety of hypervisor implementations from both private and open source vendors. Some of them are VMware, KVM, Microsoft Hyper-V, and Oracle VirtualBox. Further, into the project we will be working on VirtualBox and KVM. VirtualBox was the first option that we analyzed, however, it did not fulfill some of our requirements. Instead, we used KVM.

Oracle VM VirtualBox [39] is a type-2 hypervisor for x86 virtualization that can be installed on Microsoft Windows, MacOS, Linux, Solaris, and OpenSolaris. It is freely available as Open Source Software under the terms of the GNU General Public License (GPL) version 2. The other hypervisor, KVM [40] [41], is an open source virtualization technology built into Linux (i.e. it is part of Linux). Specifically, KVM turns Linux into a type 1 hypervisor. This integration in the kernel makes it much more efficient than other hypervisors.

An important concept about VMs is the one of clone. Cloning a VM creates a new machine that is a copy of the original. This implies that it will have the same configuration, the same virtual hardware and the same OS installed, among other properties. This process offers a quick and easy way to create several machines with common characteristics.

Another important feature is the access to the machine. The two most common methods are through the graphical interface of the hypervisor itself or through the SSH connection. SSH is a protocol for accessing machines remotely. This allows the user to manage one of the guest machines from the host terminal. This solution represents a more efficient method of using VMs by not requiring a graphical system.

To sum up, it is important to acknowledge how networks are used to work in hypervisors. A wide variety of networks are possible, such as the internal network between VMs or the host-guest connection. The most commonly used is the LAN, in which the host works as a gateway, giving access to the Internet. As a result, they generate a network interface associated with each VM. These interfaces are attached to a bridge which distributes the traffic and have their own IP address: the gateway. Although several network types have been listed, they all work on the same bridge principle with the addition of certain specifications, such as the host not forwarding to the router (internal network).

# CHAPTER 2. PROJECT DEFINITION

Chapter 2 focuses on how the project has been approached and developed. For this purpose, it has been divided into three sections. Foremost, we shall define the objectives to be achieved during the realization of this project. Secondly, work packages have been created to simplify the objectives, as well as to be able to control the pace of work. Finally, the distribution of the tasks in time and the difficulties encountered during the execution process, are shown in a Gantt diagram.

## 2.1. Objectives

The main purpose of this thesis is to contribute to the research of satellite communications, with a dedicated development of a novel emulation equipment. This project stems from the limitations on the satellite network emulators of current state-of-the-art, remarked in the dissertation *Towards The Deployment of Software Defined Networks Over Satellites - An In-Laboratory Demonstration For GEO Satellite Services* authored by Jose Luis Avila Acosta [42]. To fulfill this purpose, an initial research process has been carried out to understand the needs of a satellite network and the limitations of current emulators. With this information, we established some design requirements to be satisfied by our development of VSNeS.

The starting point after Avila's master's thesis was the implementation of OpenSand. This system allowed him to interconnect virtual machines by simulating a link through a satellite network. This connection was limited to a fixed delay and a transmission only over the DVB-RCS2/DVB-S2 protocol. In addition, the system did not allow analyzing the packets on the satellite before being forwarded. A fixed delay is a good representation on GEO satellites, but when we focus on LEO or MEO orbits the delays change over time and disconnections are generated. The New Space movement is characterized by the launch of satellites and constellations in LEO orbits, so the current focus of interest is here. These factors require for an emulator capable of representing all types of orbits and the channels that are generated.

To fulfill the aims established, we have defined the following objectives:

- Objective 1: analyze the current emulators to understand how they work and improve their limitations.
- Objective 2: develop a satellite network emulator. It has to be representative with the satellite mobility and the characteristics of the network (principally variable delay, variable topology).
- Objective 3: provide a performance profile of the developed engine.
- Objective 4: deploy real scenarios in the emulator that demonstrate its usefulness.

## 2.2. Work Package Structure

The steps followed to fulfill these objectives are defined by Work Packages (WP). They have consisted in a first part of review information about emulators, virtualization and orbit mechanics. The next work packages have been focused on designing and implementing the different modules required in the final software. In addition, there was a work package to link the modules. Once the program is operational, we built the use case. And the final step is to prepare the report.

- WP1 ”Review the network emulators”: it consists on reviewing the existing satellite emulators. Once found, we analyze their qualities to select the one that best adapts to our needs.
- WP2 ”Understand virtualization tools”: It focuses on knowing the principal virtualization tools (Virtual Machine and container) and knowing know how the tools like hypervisors are used.
- WP3 ”Design and implement emulated networks”: It involves of the connection of several VMs in which a parameter of their connection is modified. For its implementation, it has been necessary to decide the best way to connect them to each other and to place a network emulator between them.
- WP4 ”Review satellite dynamics”: it is concerned with searching information about the satellite move effect and how to predict it. Move effects are the variation of delay and temporal connectivity. To predict this, it is necessary the study of TLEs and orbit propagators.
- WP5 ”Design and implement orbit propagator”: The aim here is to develop a system capable of locating a node from a TLE at a precise instant. This data must be processed in such a way that for each instant it is determined if there is vision between nodes and if so, the corresponding delay.
- WP6 ”Design and implement 3D virtual globe GUI”: It focuses on the deployment of an application or a web server capable of displaying 3D data around the Earth. It also requires the translation of the data generated by the orbit propagator into a language understandable by the visor.
- WP7 ”Design and integration of the Software”: It centers on grouping WP4, WP5, and WP6 in a single software that implements the results of the three tasks. The result of this task ends with a software capable of interconnecting in a representative way VMs that represent satellites or ground stations and having a graphical support.
- WP8 ”Tool verification”: Development of a test in which SDN over satellites or a realistic scenario of a satellite network is implemented. Finally, it was considered that implementing SDN functionalities was beyond the scope of this thesis, so a relay satellite scenario has been developed.
- WP9 ”Report preparation”: It will consist of writing a report (the current document) detailing the work procedure and the final results, as well as all the previous knowledge for its comprehension.

Figure 2.1 shows the dependencies that exist between the WPs. This defines how the conclusion of some depends on the beginning of others and at the same time there are independent WPs, which allow parallel work.

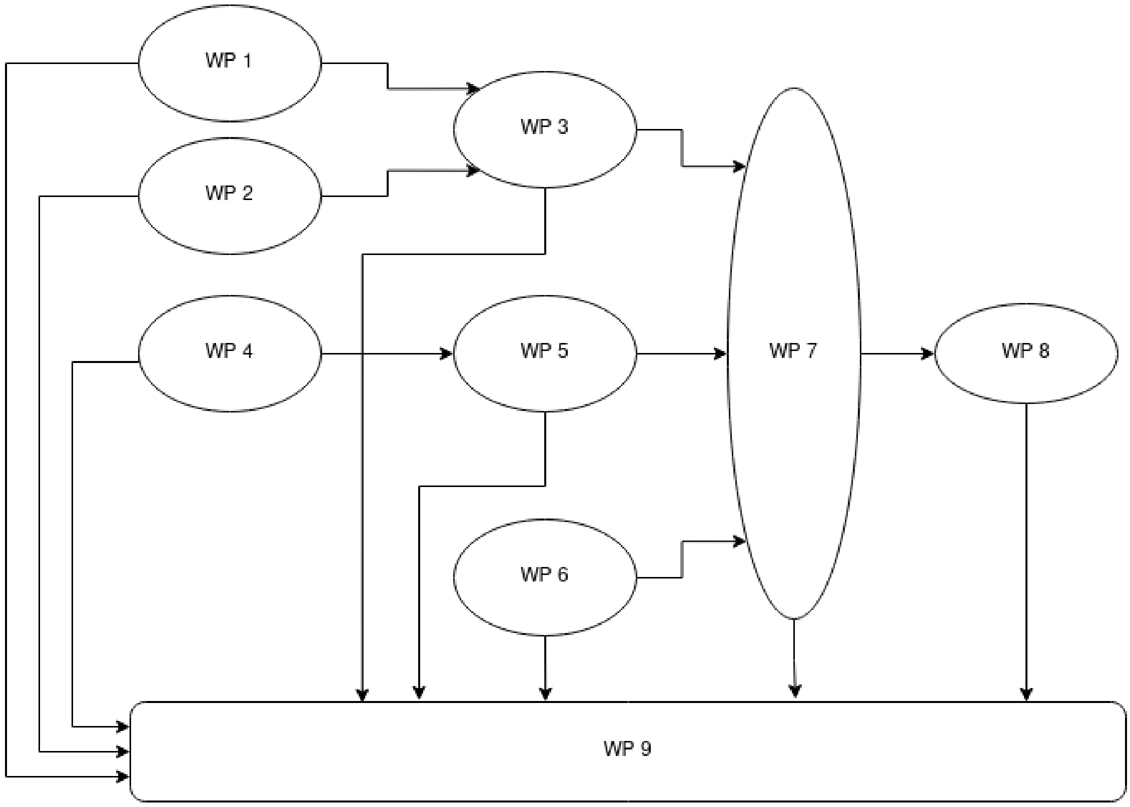


Figure 2.1: Work packages dependency.

## 2.3. Schedule

Table 2.1 shows how the elaboration of the thesis has been organized over a period of approximately 8 months. Green scales indicate the months in which the WPs were performed, where the darkest green is the month in which a milestone was planned for that WP. We started researching all the crucial data and understanding the problematic and the proposed solution. During the second month, we had to do the WP3 and start the WP4 and WP7. WP3 caused us some problems, which made it impossible to follow the scheduled route. This led us to finish the WP3 at the end of April. Parallel to the previous issues, we started to learn about the satellite orbits and to implement them. Despite the delays, we managed to catch up at the beginning of May. The main in May was to unite the emulation part with the orbits part and to create a system to automate the configuration.

At the beginning of June the emulator was functional and could connect several VMs together with representative orbits, however, the system has been refined during the rest of the months until its delivery to make it more robust and faster. June was focused on the implementation of the GUI, combining the first weeks of learning about the software, with a later implementation on VSNeS. Since July the project has followed three pillars: the development of the memory, the testing of the program with a realistic scenario and the improvement of the limitations that were detected. The completion of all WPs were linked to milestones. These consisted of demos of the software or the presentation of data collected from both tests and research of previous work.

Project Schedule

| | M1 | M2 | M3 | M4 | M5 | M6 | M7 | M8 |
|---|---|---|---|---|---|---|---|---|
| | 02/22 | 03/22 | 04/22 | 05/22 | 06/22 | 07/22 | 08/22 | 09/22 |
| WP1 ”Review the network emulators” | 🟩 | | | | | | | |
| WP2 ”Understand virtualization tools” | 🟩 | | | | | | | |
| WP3 ”Design and implement emulated networks” | | 🟩 | | | | | | |
| WP4 ”Review satellite dynamics” | | ▨ | 🟩 | | | | | |
| WP5 ”Design and implement orbit propagator” | | | ▨ | 🟩 | | | | |
| WP6 ”Design and implement 3D virtual globe GUI” | | | | | 🟩 | | | |
| WP7 ”Design and integration of the Software” | | ▨ | ▨ | ▨ | ▨ | 🟩 | | |
| WP8 ”Tool verification” | | | | | | ▨ | 🟩 | |
| WP9 ”Report preparation” | | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | 🟩 |

Table 2.1: Project Schedule: Gantt diagram followed during the elaboration of the thesis.

# CHAPTER 3. EMULATOR DEFINITION

This chapter introduces the first software concept that has been developed. It defines the general idea followed by a set of fundamental requirements for the final model. Once the requirements are established, we compare them with the technologies mentioned in Chapter 1.3.2.. This allows us to discard technologies and find starting points that will help our development.

## 3.1. Introduction

Currently, the deployment of satellite networks has a clear trend towards large constellations of LEO satellites. A clear example is the Starlink constellation [43], from the SpaceX company, which tries to provide Internet access to most of the Earth surface. These new projects offer a very broad field of research in the operation of satellite networks.

To develop this field of research thoroughly, there is a necessity for an emulation software with which to test novel developments. This software is capable of emulating an end-to-end connection in real time through a constellation of satellites considering the delay, the temporality of the contact caused by the relative motion and the possibility of ISL.

This thesis elaborates an open-source software that meets these needs for research. To accomplish this, we have determined a set of key requirements that are considered essential for this task and a new engine, called **Virtual Satellite Network Simulator (VSNeS)**, has been proposed. All these keys are compared with existing technologies, that allow us to know if we can start from an existing software that supplies an important part of our requirements or if it is necessary to start from scratch. If this is the case, despite not being able to proceed directly from a project, its analysis can provide us with a broad view of the concepts that may have been overlooked or give us support in parts of the application.

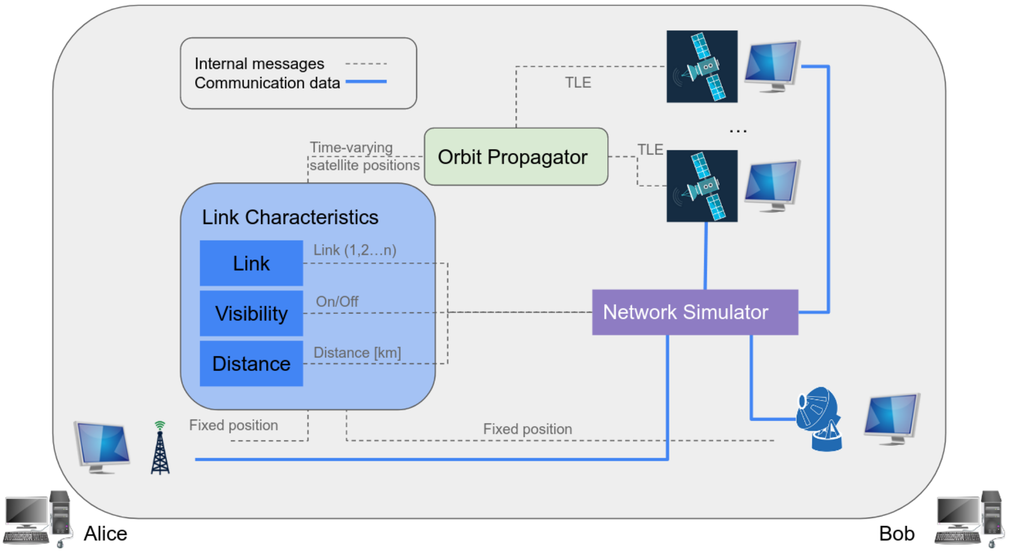


Figure 3.1: First conceptual sketch of the software to be deployed.

A concept scheme of the software is shown in Figure 3.1. It illustrates a simple configuration consisting of a constellation of satellites, a GW, and an ST. Also, thus depicts how each element is linked to a virtual machine that interacts with different modules. The modules consist of an orbit propagator, which determines the position of the satellites as a function of the time instant and the link characteristics. They define the existence of LoS of each link and the distance that forms them. Finally, the network simulator module applies the characteristics of the links to the transmitted data. In addition, real machines are placed next to the GW and the ST with which, through the software, the communication between external machines (Alice and Bob) shall be allowed.

## 3.2. Requirements

A set of requirements has been arranged to ensure that the VSNeS has the essential functionalities. These requisites go from essential concepts of a satellite communications network emulator to more specific concepts and needs that have been detected in parallel with other projects [42]. The requirements are the presented further along with an identifier Rn, where n is a number.

**(R1)** The software shall define a scenario consisting of at least Gateway (GW), Satellite Terminal (ST), and Satellite (SAT) type elements. **(R2)** All these elements shall be explained within our environment by Virtual Machines (VM). The use of VM is of great importance in order to be able to replicate the connections in the most realistic way possible, to be able to define the software and hardware of the elements of the real system. The emulation shall not be a closed environment **(R3)**, it shall have external ports that shall allow connection to GW or ST, allowing connection between two or more machines through the program.

The main element of the emulator shall be the simulation of the links, of which we shall not focus on the physical effects such as attenuation or fading. We shall focus on **(R4)** the introduction of representative delays based on the propagation time, which shall not be constant, **(R5.1)** the possibility of connecting and disconnecting of the different connections depending on the vision of the equipment, which is an essential requirement to be able to replicate a mobile environment of satellites.

Moving forward with the concept of representing an environment of non-GEO orbits, it shall be necessary to **(R5.2)** establish the ground stations at specific points on the Earth's surface and **(R5.3)** define any type of orbit that specifies the position of the satellites as a function of time. With this data, the program shall **(R5.4)** establish the existence of LoS, taking into account the Earth. All the requirements included in **R5** are intended to emulate in a representative way the dynamics of the satellites in different orbits.

This project tries to offer a very versatile system that offers facilities in the deployment of new characteristics of the network. For this reason, it is essential **(R6)** to be able to externally modify the standards and protocols used in communications, giving the greatest possible flexibility to the research. Moreover, it is important to offer verification or understanding facilities, so **(R7)** it must be possible to carry out measurements, such as reading packets, in any element of the connection and not just at the ends.

One of the most costly or repetitive points for the user is usually the configuration and deployment of the scenario. For this reason, key points in the development of the emulator are **(R8)** the introduction of data through a configuration file, in which all the ground stations and satellites shall be defined, each with their pertinent data (name, type, position, port, etc.). Furthermore, we are faced with the simulation of satellite constellations, which encompasses numerous devices, therefore **(R9)** the deployment and configuration of the VMs shall be automated. This tool aims at reducing user time, as well as minimizing errors during configuration.

Finally, a Graphical User Interface (GUI) will be introduced so that the user shall be able to verify the operation of her/his configuration. The GUI focuses on **(R10)** offering a view of the scenario, where the current position of the GB and the satellites is shown. This deployment is planned to be carried out through Cesium.

| ID | Description | Check |
|---|---|---|
| R1 | **Architecture elements:** The software shall be able to define a scenario consisting of at least GW, ST, and SAT type elements. | R |
| R2 | **Virtual Machines:** The nodes of the scenario are represented by VM. | R |
| R3 | **External access:** ST and GW must offer access to external machines to communicate across the emulated network. | T |
| R4 | **Representative Delay:** The messages suffer representative variable delays depending on the propagation time caused by the distances between the devices. | T |
| R5 | **Orbit simulation** | - |
| R5.1 | **Discontinuity in the links:** The links between the nodes shall not always be operative. This state change in function of the visibility between the equipment. | T |
| R5.2 | **Ground Station position:** the ground stations (ST, GW) are situated at specific points on the Earth's surface with specific coordinates. | T |
| R5.3 | **Orbit propagator:** The satellite position change as a function of time and it is defined by a Two Line Elements (TLE). | T |
| R5.4 | **Visibility detector:** The software has to establish the existence of LoS between devices, taking into account the Earth. | T |
| R6 | **Flexibility in communication:** The communication protocols and standards can be integrated by external libraries. | T |
| R7 | **Measurement points:** it must be possible to carry out measurements, such as reading packets, in any node of the connection and not just at the ends. | R |
| R8 | **Configure file:** it is necessary to be able to introduce the scenarios through files instead of manually, facilitating the simulation of more complex scenarios. The file shall have information about all the ground stations and satellites (Id, type, position, port, etc.). | R |
| R9 | **Automatic deployment:** The deployment and configuration of the VMs, links, and scenarios shall be automated. This tool aims at reducing user time, permitting complex scenarios, and minimizing the errors during configuration. | R |
| R10 | **Graphical User Interface (GUI):** The GUI offers a view of the scenario, where the current position of the GS and the satellites are shown in a 3D geospatial. This deployment is planned to be done through Cesium. | R |

Table 3.1: Definition of requirements.

Table 3.1 groups all the requirements and sets out the verification method required for each. R implies verification by review of the design and T by test.

## 3.3. Comparison of the Requirements and the Existing Technologies

This section is focused on the qualities mentioned the software in relation to our requirements. Although numerous projects have been mentioned, we only analyze those that fulfill our requirements and offer enough information to compare the different points.

First, we encounter **OpenSAND** [24], which perfectly meets the requirements R1 and R2. Its architecture consists of different elements (ST, GW, and SAT) each one on a machine (either physical or virtual) which can be connected between them or to external interfaces, fulfilling R3. Like all satellite network simulators, it focuses on the effect caused by the delay, which can be modified to establish simulations at different altitude, thus fulfilling R4. Despite being able to modify the distance parameter between devices, it does not offer the possibility of emulating the temporal contacts, so it does not meet any of the R5 requirements.

Besides, it does not provide flexibility in terms of protocols as it only offers communications via DVB-RCS2 - DVB-S, not complying with R6. If we dive into the flexibility of the measurements, we find that they can be made on the ends of the network, but no measurements of the satellite reception or emission can be obtained. This is because the simulation takes place in this terminal and data can only be obtained from the beginning or the end of the simulation, so it does not comply with R7. A transition from version 5.2 to version 6 is currently underway. This new version includes a web interface, as well as the use of XML configuration files, fulfilling the R8 requirement. However, R9 is not included because it does not have the possibility to deploy the scenario automatically. This is due to the fact that it requires the manual deployment of VMs or the connection of physical equipment.

In addition to the web interface, OpenSAND has a GUI that provides a schematic of the topology with indications of the available links, as well as the possibility to change the configuration. However, it does not offer a GUI with the functionalities required by R10.

The next technology to our interest, which has not yet found information on the final deployment, is **SCNE** [26]. This offers a deployment of different nodes (SAT, GW, and UT), which validates requirement R1. However, these elements are not formed by VMs, but are simulations of that model the devices. Despite not fulfilling requirement R2 directly, it is possible to interact externally with ns-3 by means of VMs or real equipment, being able to connect the simulated network with the physical network (verifying requirement R3).

STK and ns-3 offer two very powerful tools to perform a feasibility study between devices over time of fixed and mobile equipment (satellites) by determining the channel characteristics from STK and then simulating the effects of these characteristics on the network with ns-3. The combination of these two technologies provides sufficient resources to meet R4 and R5 requirements.

Ns-3 accepts external libraries, which make it possible to incorporate protocols through scripts, providing flexibility in the simulation, which validates requirement R6. Regarding R7, the SCNE documents do not mention any data on how the measurements are taken, nor has it been possible to identify what type of data ns-3 offers at each level. Moreover, there is no mention of the introduction of user-level configuration files to SCNE. However, STK is able to save and open existing scenarios, while SCNE orchestrator can generate files from STK for the network simulator. This combination ensures an automatic deployment of the saved scenario validating the R8 and R9 requirements, although R9 would only be fulfilled by the automatic deployment of the connections and scenario, while the VM deployment shall be done externally.

STK offers a GUI with both 3D and 2D geospatial representations, similar to Cesium, validating R10. However, despite having free versions, it is not an open-source like Cesium.

The next technology that offers some differential characteristics is the **Scalable Cloud-Base LEO Satellite Constellation Simulator** [31]. In this case, only two types of elements are defined, satellites and ground stations. Both simulated by means of VM as an extra element, the Earth, where the Netfilter hook is developed among other resources. Although it only has one type of GS, we could validate requirement R1 as well as R2. In the paper, there is no reference about sending data through external elements. However, it should be possible to communicate externally with the machines located in the cloud.

By combining three modules (i.e. SGP4, Line-of-sight Daemon, and Nerfilter kernel Module), the system is able to simulate the location of GS and satellite orbits to determine the view between devices and their distance for subsequent network simulation. In spite of taking into consideration the delay during the latency verification performed in the paper [31], the delay is not representative for the simulated scenario. Therefore, requirement R4 is not fulfilled although R5 is matched.

Only the support of Internet Protocol (IP) is mentioned in this technology, without mentioning the possibility of introducing external protocols or standards, which leads to the assumption that R6 is not fulfilled. Despite this lack of flexibility, the system does support representative measurements thanks to the centralization of the channel simulation, competing with R7.

The R8 requirement, based on the use of configuration files, is not directly fulfilled, although a data storage technology, MySQL database, is included, which allows the storage of scenarios and their subsequent deployment. Finally, we have noticed that it implements the GUI using Cesium as previously proposed in this project, so it complies with R10.

As mentioned above, **CORE** [33] is not a system developed implicitly in the simulation of satellite networks. As a result, a larger number of elements can be defined. However, the possibility of establishing ST, GW or SAT is excluded. Among the elements that can be defined are: Router, Host, PC, MDR, PRouter, Edit, Hub, Switch, Wireless LAN, RJ45, and Tunnel. Despite not working with VMs directly and focusing on containers, several of the elements mentioned above such as RJ45, PRouter, and Tunnel allow access to external machines, so that VMs can be introduced to the emulation environment, as well as access from external machines to the simulation, either directly from the program or from one of the VMs. These events do not directly validate R1 and R2 requirements, but offer flexibility to adapt processes in addition to meeting R3.

CORE allows the configuration of the channel both in its creation and execution time. The configurable parameters are bandwidth (bps), delay ($\mu$s), jitter (us), and loss (%). It also includes the possibility of introducing scripts to define the scenario, such as the mobility of the nodes or their position by coordinates. Although CORE only implements the disconnection between devices depending on the distance, it is possible to implement EMANE on this software, which allows the simulation on MAC and PHY layer. In other words, R4 and R5 requirements are not met directly, but tools are provided to meet them.

CORE is a very flexible emulator that offers the possibility of introducing protocols, as in the example mentioned above through ION, validating R6. In addition, it offers a great measurement capacity by being able to incorporate external software such as Wireshark, which allows analyzing the packets on all nodes, thus fulfilling the R7 requirement.

As far as scenario deployment is concerned, the emulator is capable of saving configuration files to automatically deploy the network without the need for reconfiguration. However, since VMs are not integrated, only the software access ports can be automatically enabled and not the complete configuration with VMs. This validates requirement R8, but rules out R9.

Finally, in terms of GUI (R10), this software includes by default a canvas. However, it can be extended to a 3D model of the globe through a predefined script, providing the required tool for the simulation.

All these features offer a lot of useful qualities for the emulation of any network, which makes CORE a good starting point.

Table 3.2 summarizes all the above information. In red are marked the fields that are not fulfilled. While in yellow are marked fields that are not fulfilled directly, these emulators offer a way to implement the requirement or give an extra to supplement it.

| **Requirements** | **OpenSAND** | **SCNE** | **SCBLSCS** | **CORE** |
|---|---|---|---|---|
| R1 | 🟩 | 🟩 | 🟩 | 🟩 |
| R2 | 🟩 | 🟥 | 🟩 | 🟨 |
| R3 | 🟩 | 🟩 | 🟩 | 🟩 |
| R4 | 🟩 | 🟩 | 🟥 | 🟩 |
| R5 | 🟥 | 🟩 | 🟩 | 🟨 |
| R6 | 🟥 | 🟩 | 🟩 | 🟩 |
| R7 | 🟥 | 🟩 | 🟩 | 🟩 |
| R8 | 🟩 | 🟩 | 🟨 | 🟩 |
| R9 | 🟥 | 🟩 | 🟩 | 🟨 |
| R10 | 🟥 | 🟩 | 🟩 | 🟩 |
| Accessible | 🟩 | 🟥 | 🟥 | 🟩 |

Table 3.2: Advisement of requirements with existing technologies.

# CHAPTER 4. PRELIMINARY DESIGN DISCUSSION

This chapter focuses on the technologies implemented in the final release. For this purpose, it has been divided into two sections. The first one explains the technologies that had been initially proposed and which have been finally implemented. In the second one, a more exhaustive analysis of the selection process of some of the software is made, showing how they have been tested in order to verify that they satisfying the requirements.

## 4.1. Software selection

When developing the thesis, we have changed the approach to find the best way to reach the requirements. This produces a methodology of short steps to validate the technologies and avoids working in one way if it involves more disadvantages than benefits.

**Network Emulator**

Upon studying the different existing emulation technologies and comparing them with our requirements, it has been observed that there are concepts similar to our proposal. However, none of them satisfies all the requirements. Despite these differences, it has been concluded that the most efficient way to work is to start from one of the existing programs and implement improvements to meet our requirements.

Due to its flexibility and accessibility to the software, CORE has been selected as a starting point. This software offers tools to implement the desired scenarios with great convenience in implementing new protocols and performing measurements for verification. CORE shall work as the base of the simulator, offering both the routing and effects of the network, as well as a GUI, where the scenarios and connections available at any time will be shown. Starting from here, we will have to implement some concepts to it.

Firstly, we have to add the VM deployment. CORE includes connection of the VMs to the system, but we will have to automate their deployments and connections. The second point is the channel characterization. We will have to make the changes of the links features automatic according to the movement of the nodes. In particular, delays and channel interruptions. In addition, we will have to generate a script capable of generating orbits from a TLE. Finally, it is necessary to identify the nodes. As mentioned previously, CORE is not characterized between ST, GW or SAT. The software shall characterize the VMs to work according to their functionality. Additionally, it modifies the GUI's symbology with representative node types.

As it shall be mentioned onwards, CORE could not be used in this project despite its great potential. This is due to the limitations of the 6.5.1 version, where it is not possible to characterize a Wireless Local Area Network (WLAN) with different delays for each pair of nodes. This led to require complex topologies and not to use the true potential of the program, that relies on the use of the CORE nodes and not of virtual machines.

We detected that CORE offered an emulation and a network management part that is more than what we need and it is counterproductive. For this reason, we focused on the emulation part. As mentioned above, CORE uses NetEm, which makes the emulation part. In contrast to CORE, NetEm does not generate network topologies. It characterizes a channel by modifying the output of the packets on each interface, which permits to define different rules for traffic that across the interface.

### Hypervisor

There are many hypervisors that work similarly. However, it is important to analyze the performance. For instance, if it is open source, as well as its flexibility. Hyper-V, for example, does not have the required flexibility. It only works on Windows 10, while we are working on Linux. Another hypervisor that we discarded is VMware's, because it is private. It has a free license, but it has restrictions like the number of CPUs.

The first option considered was VirtualBox [39]. It is an open source and it can work on several OS. Before the final definition of the software, we studied where to put the emulator. It could be in the host, a guest VM or a server. If the emulator works out the host, the host OS does not care. It made it interesting to use VirtualBox.

Despite being one of the most common virtualization software at the user level, it had to be discarded as it introduced problems with its operation regarding network design. The network ports of the host, which connected to networks next to the guests virtual machines, did not work promiscuously. Therefore, they did not allow the forward of messages of interest. For this reason, the KVM [40] program was analyzed as a second and definitive option.

KVM works over the Linux kernel, which limits the flexibility of our software, but it fulfills one of our requirements: working with Linux.

### Orbit Propagation

An important point about the emulator is the study of the movement of the satellites. However, we do not require high position accuracy. The software is intended to be able to emulate the effect of handover and delay variations, not to study positioning as in GNSS systems. However, a scalable design has been realized in which specific propagators can be introduced according to the accuracy requirements.

To start with, a Two-Body orbit propagator was implemented. The propagator was developed using functions that implemented different algorithms for position definition and reference system changes. Subsequently, libraries which allowed an easy implementation of the orbit propagator were found in particular of an SGP4 propagator, thus they were implemented. This provides us with a more accurate calculation of the position from an optimized module and with a verification that the results are correct.

Although Two-Body and SGP4 are implemented in parallel, SGP4 has been considered as the default execution configuration.

### Software Development

Currently, there are many programming languages such as Java, Python, C and others. This opens a wide range of possibilities on how to develop software. In our case, we started to program in Python3 because our initial decision was to work with CORE which contains a Python API that allows the creation of scenarios and their management from scripts. This led us to start programming in Python to adapt to the selected software. In addition, Python was a comfortable language to implement and offers many facilities for our needs.

Although the emulator finally worked on NetEm, software development had begun, so a transition to other languages was not considered. Even so, Python offers object-based programming with numerous libraries that implement many useful functions. This makes it a tidy and easy-to-implement language.

## 4.2. Software Validation

The validation process of the software used for virtualization is illustrated below. The combination of the different programs must be capable of creating VMs and interconnecting them, as well as being able to characterize the channel, especially including delays and disconnections. For this purpose, a series of tests were carried out with increased difficulty. Starting with the connection of two VMs through an emulator, including the channel characteristics further along. Once at that point, a connection is made between a higher number of VMs, in which all of them need to have a connection between all of them through the emulator. In the same way, then the characteristics are added to the channel, first general and then specific to each pair of VMs.

As already mentioned, the starting point was CORE and VirtualBox. Throughout the following points the limitations they supose will be observed, and how KVM and NetEm replace them.

### 4.2.1. Connection Between Two VMs Test with CORE

The first step of the project was to connect two VMs with an emulator platform. We define a VirtualBox the same way as a hypervisor, whereas CORE is like a Network emulator. In Figure 4.1, we can noticebthe scheme of the first test. It is a simple architecture with two VMs connected to the host by different interfaces which are connected between them with CORE. The two VMs had an IP address in the same network, but they would not be in the same “physical” network if CORE was not running.

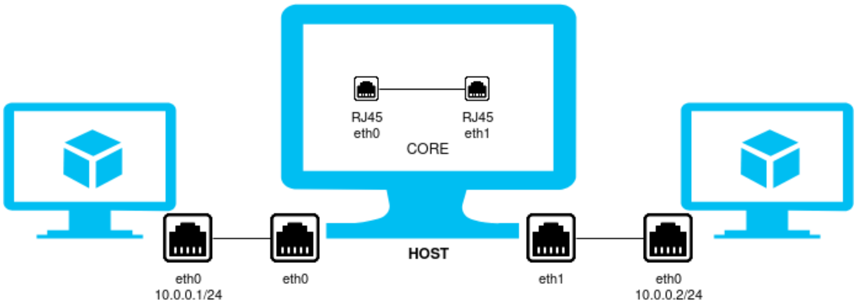


Figure 4.1: Schematic of the first test design; connection between two VMs across CORE.

The first attempt to run the scenario was not successful, as the packets sent by the VMs did not cross the host’s interface and they never passed through CORE. This failure led us to design a new scenario to understand how the system worked. The new scenario was composed of only one VM connected to a router of CORE. We implemented two different cases of this scenario: (1) with IP addresses in different networks and (2) in the same network.

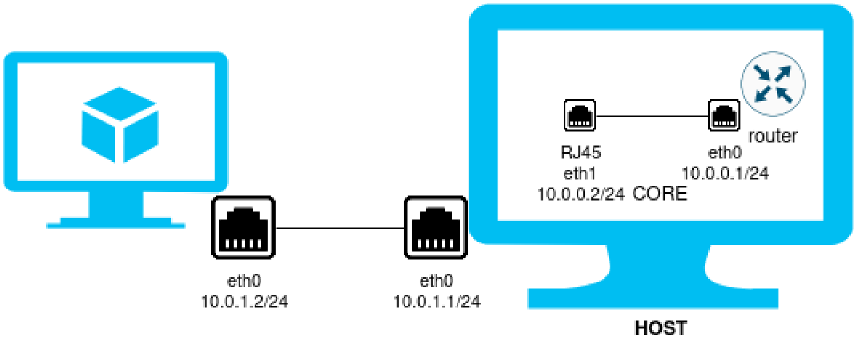


Figure 4.2: Schematic of the first test design: connection between a VM and a CORE router that works in different networks.

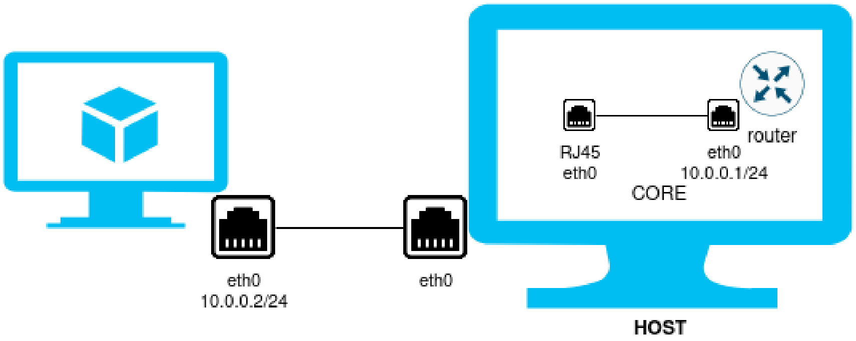


Figure 4.3: Schematic of the first test design: connection between a VM and a CORE router that works in the same networks.

The new scenario with different IPs was successful. The VM could send a ping to the router. However, this was not an efficient solution because it required a lot of configuration. As we can see in Figure 4.2, the scenario needed to define two interfaces in the host (one to connect the VM and the other one to CORE) instead of only one. Thus, it was necessary to define the IPs. Furthermore, the router did not have information about networks outside of CORE and it produced a necessary configuration of the routes. This test allows us to know that the CORE RJ45 tool worked well.

The proof of the scenario with only one network (as shown in Figure 4.3) did not work as we expected. The packets only traveled in one direction, from the CORE's router to the VM. The message from VM arrived at the host, but it did not cross the interface. To find the problem, we replicated the scenario, changing the VM by a computer. These allowed us to determine where the problem was: the hypervisor, the host or CORE.

With that test, we noticed that the problem came from the hypervisor, because the test worked well. Specifically, the problem was in the definition of the interfaces that is done by the hypervisor.

At this point, we replicated the scenario of Figure 4.3 with a new hypervisor: QEMU/KVM. The test fulfilled the requirement, thus we turned back to the first test and validated it. We configured CORE for the test in the first instance with the GUI, which offers an intuitive and simple method of configuration for simple scenarios. Once the test was validated, we replicated it by implementing the emulator configuration with a Python API, which makes it easier to automate the definition of scenarios.

### 4.2.2. Channel Characterization Test with CORE

The definition of the channel characteristics into the emulator is a key point in our software. Particularly, the delay, as well as the existence of the link. In order to know if it works and how, we created a loop in Python, which increases the delay every second with an arbitrary decision. Moreover, it disconnects the link when the delay is bigger than a threshold. This development was done as a simple and quick-test.

In the Python API of CORE there is a command to change the channel characteristics. It made it easy to adapt it to the previous test with delays. However, the configuration of the availability of the link was hard to manage if we deleted the channel. For this reason, we changed the configuration where we create and deleted the channel in relation to the LoS to change only the channel characteristics. To emulate the channel disconnection, we modified the loss packet rate: if there is a channel, its loss rate would be 0% and if not it would be 100%.

### 4.2.3. Multi-Connection Between VMs with CORE

The following step was to connect, at the same time, more than two VMs with different delays. The first idea was to replicate the before configuration, but with an extra VM. Unfortunately, it did not work. The reason being that CORE creates a bridge for every connection and it adds the interfaces to them. An Ethernet interface can only be linked with one bridge, making it a limitation for our system.

The second attempt was, instead of making a channel for every pair of VMs, connect all of them to a CORE WLAN. When CORE works with emulating a WLAN, it creates a bridge that adds all the nodes. This solution allows for a connection between all the VMs, but the version of CORE which we have worked on cannot define a specific delay for every couple. The channel characteristics were global for all the links.

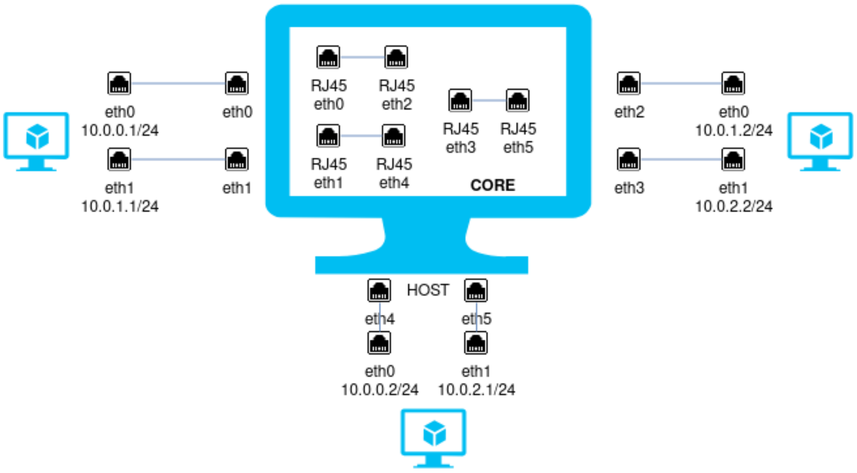


Figure 4.4: Schematic to connect multiple VMs using CORE. Every VM have one interface per node which it wants to communicate.

If we wanted to continue using CORE, there were two options. First, the VMs would have one interface per node in the scenario. With that solution, CORE can create a link to every pair of VMs and define the correct configuration for each one. In Figure 4.4, we can observe a schedule of the design and how the host would be able to create a number of interfaces following Equation 4.1, which shows that the number of interfaces increases as the square of the number of VMs, making it impractical. Although it would work, it would not be a realistic representation because the machines do not have many network cards. Also, the pre-configuration of the host and the guest are high.

$$N_{interfaces} = N_{VMs} * (N_{VMs} - 1) \tag{4.1}$$

The second option is based on creating a topology that connects all the nodes with different paths, which makes it possible to modify the channels. We put forward two topologies with CORE routers, all-to-all and star topology. The all-to-all topology (as shown in Figure 4.5) assigns one CORE router to every VM and interconnects all of them. It requires defining iptables rules to define only one path to connect the VMs. This topology works well, but it has a lot of downsides. The main disadvantages are: the large number of networks and addresses, the extra configuration of iptables, the unrealistic connection and the packets making two extra steps.

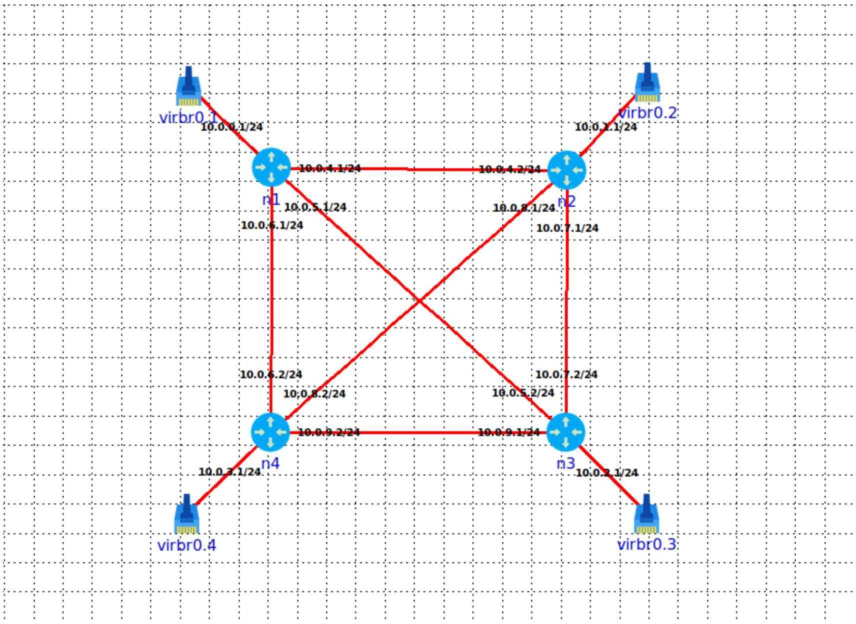


Figure 4.5: Graphical configuration of the all-to-all topology with CORE.

The star topology (as shown in Figure 4.6) connects all the VMs to the same CORE router. It is more simple than all-to-all, but it adds a difficulty: we cannot use the CORE tool for defining the channels. The CORE router has to do it. We found that it is possible to define delays and losses in the router with NetEm. However, this topology does not make sense because CORE only gives us a router and the host can work as it.

### 4.2.4. Connection Between VMs with NetEm

Once CORE has been discarded, we need to repeat the previous test. At the beginning, we thought about using the host as a router, but we finally took CORE's way of working. CORE joins the nodes with bridges, which allows all the nodes to be in the same network.

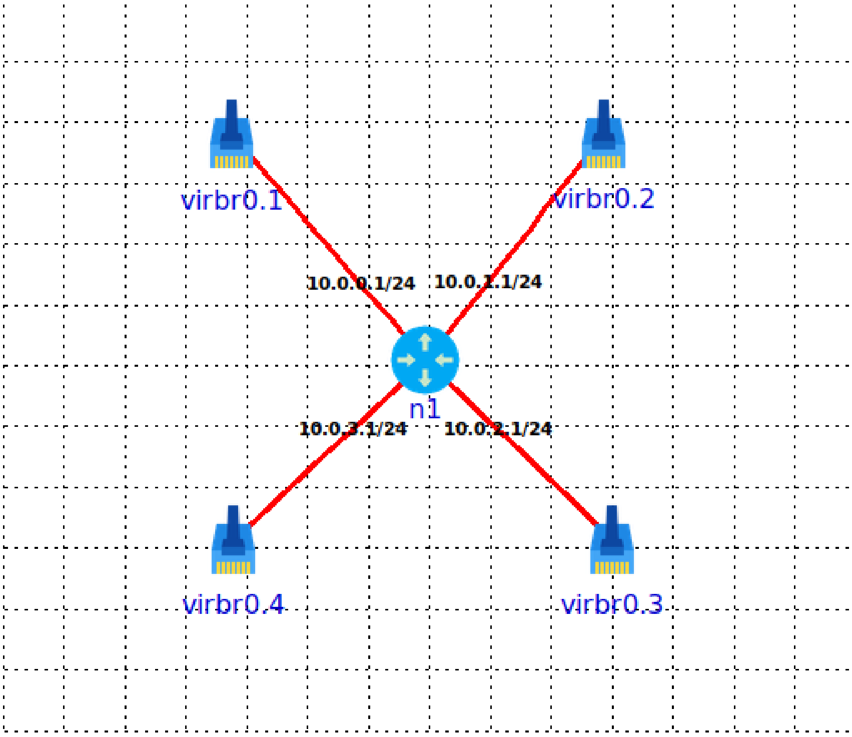


Figure 4.6: Graphical configuration of the start topology with CORE.

First of all, we made a ping between two VMs with IP addresses of the same network, but in different ‘physical’ networks that were joined with a bridge. Next, we repeated the test by putting the VMs in the same ‘physical’ network, but we assigned them different VLAN tags. That made the connection impossible without the bridge, even the physical connection. To generate this connection, the next configuration was applied in the host:

```
#Create a bridge
sudo brctl addbr brSATEMU
sudo ip link set dev brSATEMU up

#Create a VLAN interface at vnet0 with id 1 and add it to the bridge
sudo ip link add link vnet0 name vnet0.1 type vlan id 1
sudo ip link set dev vnet0.1 up
sudo brctl addif brSATEMU vnet0.1

sudo ip link add link vnet1 name vnet1.2 type vlan id 2
sudo ip link set dev vnet1.2 up
sudo brctl addif brSATEMU vnet1.2

sudo ip link add link vnet2 name vnet2.3 type vlan id 3
sudo ip link set dev vnet2.3 up
sudo brctl addif brSATEMU vnet2.3
```

This command lines create a bridge with the name brSATEMU, in which the interfaces of every VLAN are added. In this case, we created three new interfaces related with the VLANs 1, 2, and 3. For this first test we only need two, but we have defined an extra one to use it in the next test.

### 4.2.5. Channel Characterization Test with NetEm

When we obtained a connection, we started to work with NetEm to add delays and disconnections. NetEm works over the output packets of a specific interface. In our configuration, there is one interface for every VM, so if we want to define a specific delay between two VMs, we have to define twice (one in every interface). In order to test the scenario, we applied on the host the following line commands, which defined a delay for the packets sent over the bridge:

```
tc qdisc add dev vnet0.1 root netem delay 10ms
tc qdisc add dev vnet1.2 root netem delay 20ms
tc qdisc add dev vnet2.3 root netem delay 30ms
```

These three lines configure one delay for every machine. Specifically: the packets to the VM1 have a delay of 10 ms, whereas the VM2's have 20 ms and 30 ms for the packets that go to the VM in the VLAN with the id 3.

At that moment, we could make pings with a specific delay, but that delay was the same for every connection. To change this, we had to apply filters in relation to the source IP address or other rules. We achieved this by deleting the previous configuration and executing the next one:

```
sudo tc qdisc add dev vnet0.1 root handle 1: htb
sudo tc class add dev vnet0.1 parent 1: classid 1:2 htb rate 100mbit
sudo tc qdisc add dev vnet0.1 parent 1:2 handle 12: netem
//delay 10.0ms
sudo tc filter add dev vnet0.1 protocol ip parent 1:0 prio 1 u32
//match ip src 10.0.0.2/32 flowid 1:2
sudo tc class add dev vnet0.1 parent 1: classid 1:3 htb rate 100mbit
sudo tc qdisc add dev vnet0.1 parent 1:3 handle 13: netem loss 100%
sudo tc filter add dev vnet0.1 protocol ip parent 1:0 prio 1 u32
//match ip src 10.0.0.3/32 flowid 1:3

sudo tc qdisc add dev vnet1.2 root handle 2: htb
sudo tc class add dev vnet1.2 parent 2: classid 2:1 htb rate 100mbit
sudo tc qdisc add dev vnet1.2 parent 2:1 handle 21: netem
//delay 10.0ms
sudo tc filter add dev vnet1.2 protocol ip parent 2:0 prio 1 u32
//match ip src 10.0.0.1/32 flowid 2:1
sudo tc class add dev vnet1.2 parent 2: classid 2:3 htb rate 100mbit
sudo tc qdisc add dev vnet1.2 parent 2:3 handle 23: netem
//delay 20.0ms
sudo tc filter add dev vnet1.2 protocol ip parent 2:0 prio 1 u32
//ip src 10.0.0.3/32 flowid 2:3

sudo tc qdisc add dev vnet2.3 root handle 3: htb
sudo tc class add dev vnet2.3 parent 3: classid 3:1 htb rate 100mbit
sudo tc qdisc add dev vnet2.3 parent 3:1 handle 31: netem loss 100%
sudo tc filter add dev vnet2.3 protocol ip parent 3:0 prio 1 u32
//match ip src 10.0.0.1/32 flowid 3:1
sudo tc class add dev vnet2.3 parent 3: classid 3:2 htb rate 100mbit
sudo tc qdisc add dev vnet2.3 parent 3:2 handle 32: netem
//delay 20.0ms
sudo tc filter add dev vnet2.3 protocol ip parent 3:0 prio 1 u32
//match ip src 10.0.0.2/32 flowid 3:2
```

The commands define a scenario where: the channel of the VM1 (IP 10.0.0.1/24) with the VM2 (IP 10.0.0.2/24) has 10 ms of delay, the channel between VM2 and VM3 (IP 10.0.0.3/24) has 20 ms and there is no connection between VM1 and VM3.

All these tests were successful, so we could validate the technology for our case. The next step was to prepare the manager software to automatically make the before configurations for specific scenarios and modify the channel when the code was running.

# CHAPTER 5. VSNES DESIGN

This chapter illustrates all the information related to the final model of our emulator, named Virtual Satellite Network Simulator. Foremost, an overview of the program, as well as the structure in modules that work independently managed by a central module, are described. Then each module is analyzed in detail. These segments are focused on explaining how the different programs used have been implemented. In addition to the modules, we deeply explain concepts such as automatic configuration or orbit propagation, which despite not being independent modules, they are a very important part of the operation of the engine.

## 5.1. Introduction

For the development of the emulator, Virtual Satellite Network Simulator, four independent blocks will be combined: a hypervisor, a network emulator, a 3D globe viewer, and a manager module. The management module is where the software development will be performed. Figure 5.1 shows a scheme with the distribution of the different modules:

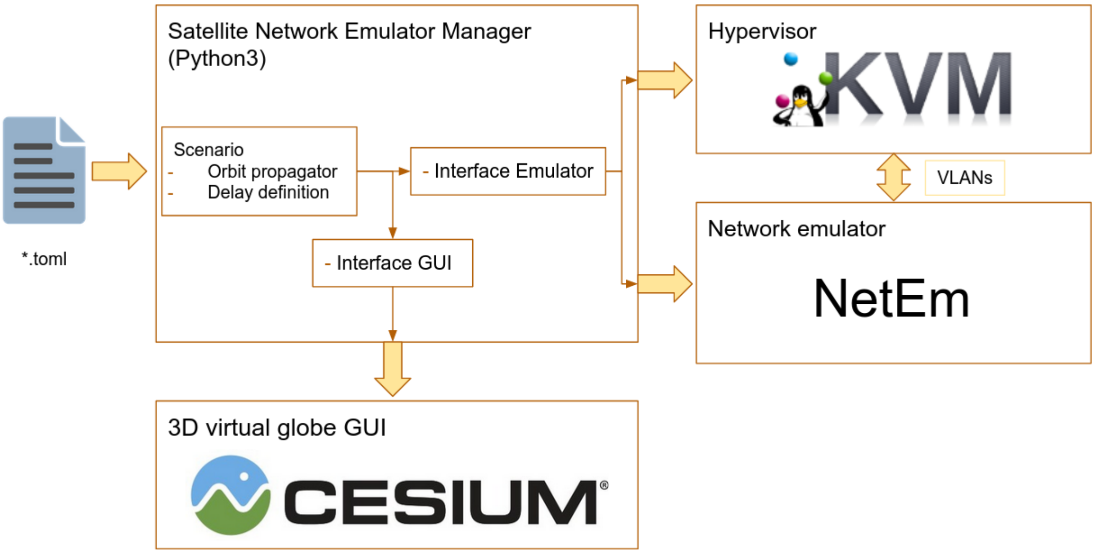


Figure 5.1: Representative diagram of the software structure.

As shown in the diagram, the software starts by reading a configuration file from which the manager defines a scenario. This scenario consists of a set of nodes which have a given position or orbit. From this data, the manager characterizes the links between the different nodes for specific instants of time.

The position and channel information are processed in order to be implemented in two ways: network emulation and data visualization. First, as many VMs as nodes are created and interconnected through the network emulator, which is updated with the delays of each channel.

In the other direction, we find Cesium, the 3D viewer, in charge of offering a graphical help to understand the topology at each instant of time. The management module generates an initial 3D image that is subsequently updated to match the graphic information with the emulated network topology.

The user has access to three different concepts: the main program (manager), the virtual machines (VMs), and the 3D viewer. The main program (manager module) is the only point where the user can interact to run the program and make modifications to the simulation, as it works through commands. The user has access to the VMs that are generated in order to perform the appropriate emulations. And, finally, in 3D viewer, which has only an informative function and it is not able to interact with the emulation.

## 5.2. VMs and Network Emulator

The hypervisor and the network emulator modules make the virtualization of the scenario. The combination of them allows representing real machines interconnected by a channel with representative characteristics.

In both modules, we have experienced a process of changes to find the optimal option to reach our goals. As we can see in Figure 5.1, we use QEMU/KVM as our hypervisor and NetEm for the network emulator.

### 5.2.1. QEMU/KVM

KVM [40] [41] is an open source virtualization technology built into Linux, i.e. it is part of Linux. Thus, versions 2.6.20 or higher include it. Specifically, KVM turns Linux into a type 1 hypervisor. This integration in the kernel makes it much more efficient than other hypervisors. In order to use KVM, other tools that work together are required. Firstly, Quick EMUlator (QEMU), which emulates many hardware resources such as disk, network, and USB. Secondly, libvirt, which provides a consistent API for provisioning and managing VMs. Libvirt manages three tools: virt-manager, virsh, and virt-install, which can perform all the management.

Virt-manager is a GUI to manage QEMU/KVM. It offers direct graphical access to VMs. Virsh is a command-line tool that manages the VMs. Finally, virt-install creates new VM guests. The manager module uses the command-line tools to interact with the VMs.

In addition to these commands, the manager software needs to configure parameters such as the IP address of the guest machines. These changes are made by SSH. This type of configuration puts a limitation or an extra work over the software. That is because every OS has some particularities on their command lines. It makes us program the requite commands to all OS.

Currently, we only have implemented the configuration to Linux, so the user has to use VMs with this OS. In our tests, we usually have worked with Ubuntu 20.04 [44] or Alpine 3.16.1 [45].

### 5.2.2. NetEm

NetEm [34] has been selected as a replacement for CORE. Netem is an enhancement of the Linux traffic control facilities that provides network emulation functionality for testing protocols such as add delay, packet loss, duplication, and other characteristics to packets outgoing from a selected network interface. Netem is controlled by the command line tool *tc*, which is part of the *iproute2* package of tools.

NetEm emulates a channel by modifying the output of the packets on each interface. However, it allows defining different rules of work for traffic that across the same interface. To create the topology of our scenario, the software has to create one interface for every node and join all of them to a hub. In this way, all the VMs are in the same network, but the host can modify the channels. To create the bridge, the software uses the brctl tool.

To work with NetEm it is important to understand the concepts of qdisc, class, and filter inside *tc*. The first item, Queueing Discipline (qdisc), is the set of rules under which an interface processes output traffic. One example can be the following, where we add a delay of 100 ms for all the outbound traffic of the interface 'eth0':

```
tc qdisc add dev eth0 root netem delay 100ms
```

Once we know how to modify the traffic, the next step is to add different rules in the same interfaces. For that, *tc* works with classful qdisc, a qdisc which contains classes. A class has a qdisc as a parent and they can have their own qdisc to add specific rules. That allows defining more than one qdisc in an interface, for we define a root qdisc which we add classes with their rules. Now we have an example of one interface with two rules, first a delay of 100 ms and other with a delay of 150 ms and losses of 5%:

```
tc qdisc add dev eth0 root handle 1: htb
tc class add dev eth0 parent 1: classid 1:1 htb rate 100mbit
tc qdisc add dev eth0 parent 1:1 handle 11: netem delay 100.0ms
tc class add dev eth0 parent 1: classid 1:2 htb rate 100mbit
tc qdisc add dev eth0 parent 1:2 handle 12: netem delay 150ms loss 5%
```

The last tool that we need it is filter. It catches some type of traffic and sends it to a qdisc. *tc filter* can match with source/destination address, source/destination port, IP protocol (TCP, UDP, ICMP, etc.), and it can mark packets by *iptables*. Following with the example above, to apply the delay of 100ms to the packets coming from a specific VM, you need to use two commands. The first one to mark the traffic with iptables and the second one to define the filter.

```
sudo iptables -A PREROUTING -t mangle -m physdev --physdev-in eth1
//-j MARK --set-mark 1
sudo tc filter add dev vnet0.1 protocol ip parent 1:0 prio 1 handle 1
//fw flowid 1:1
```

With this configuration, we are marking the traffic coming from the eth1 interface. This implies that the messages that go out through the eth0 interface and are marked, shall be detected by the filter and shall be sent to the 1:1 qdisc, where a delay of 100 ms is applied.

### 5.2.3. Connection Between VMs and Network Emulator

The emulator adds the channel characteristics to the output of the interfaces. This is why each VM is required to be connected by a different one. To achieve a simple and easy to manage configuration, it has been decided to connect all VMs and the host to a single network, in which a VLAN is generated for each VM. All VMs will be configured on the same network (e.g. 10.0.0.0/24), but with a different VLAN tag, avoiding direct connection between them. The configuration that QEMU/KVM performs is by creating a bridge between interfaces associated to the VMs. The approach is to replicate this concept with VLANs. The host will have an interface in each VLAN without any assigned IP address, which will be connected to a hub.

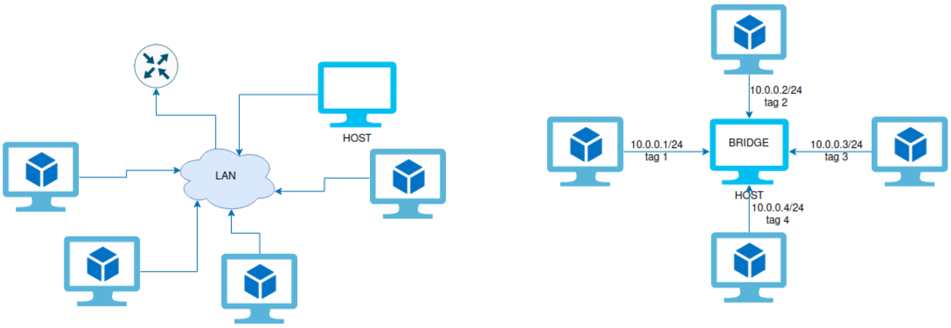


Figure 5.2: Network configuration: (1) control network and (2) emulated network (VLANs).

With this configuration, we have a first network (the default one) to control the VMs and to connect external machines to the emulator. Afterwards, a second network (the emulated one) which only allows connection between VMs.

The VLAN generation has been done using the command:

```
sudo ip link add link [IntName] name [NewIntName] type vlan id [tag]
```

This command generates a new interface that assumes the traffic tagged with the VLAN tag arriving at the parent interface. Although a bridge is created, it must work as a hub. In this way, all the machines that have a channel between them receive all the messages sent and not only the packets destined to them. This is achieved by setting the storage time of the routing tables to zero, in which the destination Media Access Control (MAC) is related to the outgoing interface. This is achieved by applying the following configuration:

```
sudo brctl addbr [brName]
sudo ip link set dev [brName] up
sudo brctl stp [brName] off
sudo brctl setageing [brName] 0
sudo brctl setfd [brName] 0
```

Regarding the configuration of the nodes, two different deployments have been implemented. The standard model, where the node has only one interface assigned to the emulation and the Relay mode, where the node has an incoming and an outgoing interface, connected to different VLANs.

The standard configuration is shown in Figure 5.2 where all machines are connected to the emulation network with a single port. With this one, the VMs only have connection when there are LoS between the nodes.

The relay mode tries to replicate the satellite communication model, in which messages are simply replicated. For this configuration, two network interfaces are generated in different VLANs and joined to a bridge, as shown in Figure 5.3. Even if there are two interfaces, only one IP address is assigned, directly associated with the bridge. This configuration has to include a set of rules using iptables and tc qdisc to limit traffic in a single direction. This is very important in order not to generate loops within the transmissions or duplicate messages, which would collapse the network. This mode requires the creation of an extra interface on the host also.

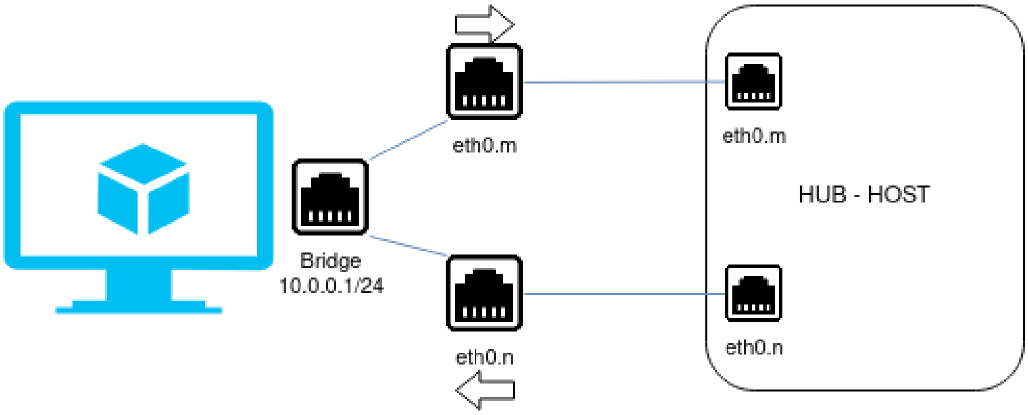


Figure 5.3: Relay configuration: Concept diagram of host and guest machine configuration in relay mode.

With this configuration, the terrestrial nodes will receive again their own messages, forwarded by the satellite. Not all protocols accept this effect, as is the case with ARP. ARP is the protocol used to determine the MAC address of a particular IP address within a network. When a node receives its own ARP message, it detects a node with the same MAC and IP, and it therefore it could generate a collapses in the network by repeating the request. To avoid this effect of network saturation, ARP tables are defined beforehand, avoiding the use of ARP request.

## 5.3. 3D Virtual Globe GUI

The 3D virtual globe GUI will serve as a graphical support for the simulation, representing the location of the nodes at the instant of the simulation and showing the links available at each instant. This requires an application capable of representing the globe and superimposing images on it. Among the existing applications there are several which we can perform these representations such as Cesium [32], SDT3D [46], and STK [27].

Among the three options mentioned above, STK is the most complete application. Since it does not only include graphical representations, but it is not open source, it requires a license. Besides, SDT3D is open source and has the possibility of interacting directly with CORE, a factor that is not important when we are intended to use them separately and CORE finally will not be used. CesiumJS is also open source and offers a more updated design than SDT3D and a great possibility of representation.

Finally, CesiumJS has been chosen, which is one of the packages offered by Cesium, that is the foundational open platform for creating powerful 3D geospatial applications. CesiumJS is a JavaScript library used to create 3D globes and 2D maps in a web browser without the necessity for a plugin.

The implementation of CesiumJS has two steps. First, we need to create a server to load an HTTP web base on a Cesium script [47]. The web is programmed to load a configuration time and if is necessary reload it with a period of time. On the other way, the software have to write the configuration file to define the virtualization scenario. This is a CZML file that permits to use a generic Cesium viewer to display a particular scene without the need for any custom code.

CZML [48] is a JSON format created for describing a time-dynamic graphical scene, mainly for display in a web browser running Cesium. It describes geometric shapes, figures, labels between others and specifies how they change with time over Earth's globe. The CZML format is a list of packets with a specific characteristics like name, position, color, etc. We have implemented three types of packets: main packet, node packet, and channel packet.

The **main packet** defines the generic information of the scene like version and clock parametres. Below there are listed all the packet items:

1. **ID:** is a string which define identifier which must be unique. The main packet ID is document.
2. **Version:** is a number that follows the next sequence [1.0, 1.1 1.2 ...]. It defines if there are changes in the file and the server need to reload the scenario.
3. **Availability:** is defined by an interval of two date times. It is the time when the file data is operative.
4. **Clock:** Cesium class that is defined by the following items. It groups all the information about the simulation time.
    (a) **CurrentTime:** it is a date time which defines the instance when the visualization starts.
    (b) **Multiplier:** it is a float which defines how much time the clock advances in the case of a clock tick.
    (c) **Interval:** is defined by an interval of two date times. It defines the start and the end of the time scroll bar.
    (d) **Range:** it determines how the clock should behave when the end time of the simulation is reached. It can be UNBOUNDED, CLAMPED or LOOP_STOP. The simulation clock will run when the range is UNBOUNDED. In the second case, CLAMPED, the clock will stop. And when the range is LOOP_STOP, the simulation clock will come back to the start of the visualization. Our software use the LOOP_STOP option.
    (e) **Step:** determines how the multiplier item is to be used. It can be defined like as TICK_DEPENDENT, SYSTEM_CLOCK_MULTIPLIER or SYSTEM_CLOCK. The first option uses the multiplier like sum of seconds. The second option, SYSTEM_CLOCK_MULTIPLIER, it is used to multiply its value by the time elapse between the clock ticks. The last option does not use the multiplier

item, it is like we would use the SYSTEM_CLOCK_MULTIPLIER with the multiplier equals to one. Our software uses the SYSTEM_CLOCK_MULTIPLIER option.

The second type of entity is the **node packets**. The file will have one packet per node with all their information. Currently, our software has two types of nodes: Satellites and Ground Stations. The first needs more information to define a friendly visualization, which means that we need to define more item. For this reason, the following packets do not apply to both nodes.

1. **ID:** it is a string which defines an identifier which must be unique. The node packet id is equal to the name of the node.
2. **Description:** it is a string with data for the user. In our case, the description has the IP address of the node and the TLE if it is a satellite and the position (latitude, longitude, height) if it is a ground station.
3. **Billboard:** it describes a two-dimensional icon
   (a) **Image:** it is a URI which defines the image/icon. Both type of nodes use icons that have used in example of visualizations in Cesium. They are represented in Figure 5.4.

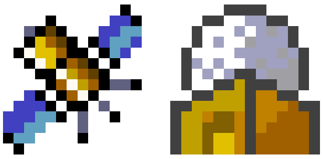


Figure 5.4: Satellite and Ground Station icons.

   (b) **Color:** It can be two things, a vector with rgtb components or a float between 0.0 and 1.0. If it uses the rgtb vector, it multiplies the image to that. In the other case, the float represents the translucent property. A value of 0.0 makes the billboard transparent and 1.0 makes the billboard opaque. The software always defines that as a 1.0.
   (c) **Scale:** it is a float that specifies the uniform scale to apply to the image. A scale greater than 1.0 enlarges the billboard, while a scale less than 1.0 shrinks it. Currently, we use a value of 1.5 to define it.
4. **Position:** it is the object which defines a world location as a Cartesian.
   (a) **Cartesian:** it is a list that defines the position in ECEF or ECI (default ECEF) and the date time of the position. In the case of the ground station, we only need to indicate the position in ECEF without date time. The correct format is: $[time_1, x_1, y_1, z_1, time_2, ..., z_n]$.
   (b) **InterpolationAlgorithm:** [only for satellite nodes] it defines the type of interpolation used to add more points between the indicated positions. The possible algorithm is Lineal, Lagrange, and Hermite. The software always indicates the method of Lagrange.

   (c) **InterpolationDegree:** [only for satellite nodes] it is float that defines in degrees the distance between the extra points. We define that as 5 degrees.

   (d) **ReferenceFrame:** [only for satellite nodes] it defines the used coordinates. The possible options for this item are FIXED or INERTIAL. FIXED is equal to ECEF coordinates and INERTIAL is equal to use ECI. The FIXED method is the default, for this reason we only indicate this item for satellites that the positions are in ECI.

5. **Label:** it describes a two-dimensional label.

   (a) **Text:** string which specifies the text of the label. The text for the nodes is its name and its IP address. In addition, the satellite label has the NORAD ID.

   (b) **HorizontalOrigin:** it indicates the horizontal position of the text in relation with the Billboard object. The item can be defined as CENTER, LEFT or RIGHT. It is CENTER for our case.

   (c) **Scale:** it is a float that specifies the uniform scale to apply to the image. A scale greater than 1.0 enlarges the billboard while a scale less than 1.0 shrinks it. Currently, we use a value of 0.5 to define it.

   (d) **PixelOffset:** it is a 2D position which specifies the label's pixel offset in screen space from the origin of Billboard object. It is used to avoid that the image and the label from being on top of each other.

6. **Path:** [only for satellite nodes] it describes a polyline defined as the positions of the object over the time. It does not apply to ground station because they only can be static objects in this first version of the software.

   (a) **Width:** it is a float which specifies the width in pixels. It is defined as 1.0.

   (b) **LeadTime:** it is a float which specifies the number of seconds in front of the object to show. It is defined equal to one period of the satellite.

   (c) **TrailTime:** it is a float which specifies the number of seconds behind the object to show. The path is done with only the leadTime and it is defined as 0.0.

   (d) **Resolution:** it is a float which specifies the maximum number of seconds to step when sampling the position. The default value is 60 but we use 120.

   (e) **Material:** object which specifies the material used to draw the path. In other words, it describes the color used to draw it. The default value is white, but we use yellow.

Finally, we create the **channel packets** to print a line between the nodes in the instants where there is a link between them. Intuition would lead one to believe that there is one such packet for each pair of nodes. However, we only create this packet in case there are any contact between them during the simulation.

1. **ID:** it is a string which defines an identifier and it must be unique. The channel packet id is equal to a combination of the names of the nodes. The exact structure is: $[Node_1Name]-to-[Node_2Name]$

2. **Description:** it is a string with data for the user. In our case, the description describes a table where there are the hours which define when the contact link starts and when it ends.

3. **Polyline:** it is an object which defines a line. We use it to show the connectivity between nodes.

   (a) **Show:** it defines if the polyline is visible or not. That item defines a list where in every position there are an interval and a boolean.

      i. **Interval:** it is defined by an interval of two date times. It defines the start and the end of the boolean validation.

      ii. **Boolean:** it is bool (true or false). It says if the polyline is showed or not during the before interval.

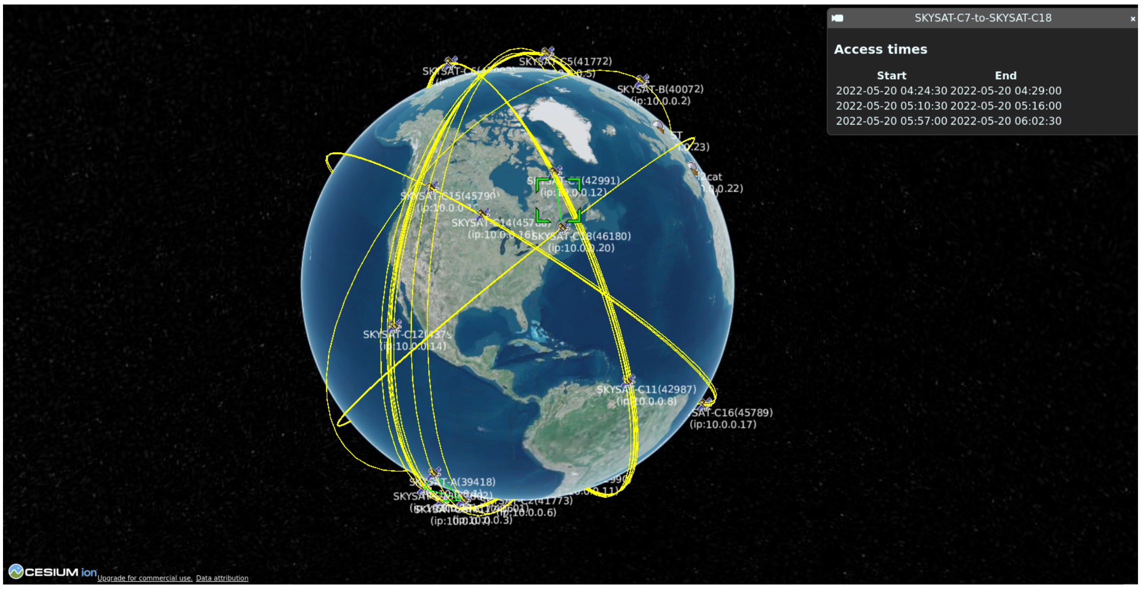
   (b) **FollowSurface:** it is a bool. If it is true, the line has the curvature of the Earth. If not, it is a straight line. The software defines that as $false$.

   (c) **Width:** it is a float specifying the width in pixels. It is defined as 1.0.

   (d) **Material:** it is an object which specifies the material used to draw the polyline. In other words, it describes the color used to draw it. The default value is white but we use green.

   (e) **positions:** defines the location of the line. A line needs two points to be drawn. In our case, it is defined with the exact position of the nodes. It has the follow structure: $[''[Node_1Name]\#position'','' [Node_2Name]\#position'']$

Figure 5.5 shows an example of Cesium running as the VSNeS GUI. In this case 21 Planet's satellites [49] are represented in addition to some GS. In the GUI, the different satellites are represented together with their future paths and the links available at the time of simulation. At the top right is an information box, in this case a table of contacts between two nodes is represented.



SKYSAT-C7-to-SKYSAT-C18

**Access times**

| Start | End |
|---|---|
| 2022-05-20 04:24:30 | 2022-05-20 04:29:00 |
| 2022-05-20 05:10:30 | 2022-05-20 05:16:00 |
| 2022-05-20 05:57:00 | 2022-05-20 06:02:30 |

Figure 5.5: Representation of SKYSAT constellation in the VSNeS GUI.

## 5.4. Orbit and Satellite Contact Models

The process of calculating orbits and contacting models is performed on the management module, Chapter 5.6.. This part of the calculation is one of the most important elements of the emulator, as it is in charge of characterizing the links. Orbit calculation and contact modeling are two separate processes. When the program is started, a list of satellite positions over time is generated. This list is later used to determine the channels at each time instant. The use of orbit propagators has been made scalable in order to make it easy to insert new propagation models. The satellites loaded in the software shall be characterized by an orbit class. This class will include subclasses formed by the different propagators.

In the current version, two subclasses have been implemented, namely Two-Body and SGP4. These two implementations have been done in a very different way. Two-Body has been implemented by means of algorithms implemented by functions, while SGP4 has been developed through well-known libraries, for instance Skyfield [50]. Both propagators perform all their calculations from the information contained in the TLEs.

The functions that run the Two-Body propagator are based on an algorithm that obtains the ECEF position from the Kepler elements and the time elapsed since the Time of Applicability (ToA). The algorithm followed is detailed in Annex A. Equation 5.1 shows the rotation matrix (R), which is used to obtain the ECI coordinates from ECEF coordinates, where $\theta$ is the angular rotation that has occurred since epoch [51].

$$R = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.1}$$

For the correct operation of the program, the propagators must be able to calculate the positions from a time instant or a time vector. These positions must be expressed in both ECI and ECEF coordinates.

Once the nodes have been placed at each time instant, the existence of the channel and its delay are determined. Two contact models have been developed: one for satellite-to-satellite connections and the other one for the links between the terrestrial and space segments.

The existence of connection between satellites is determined through two concepts: LoS and threshold distance. The threshold is one of the parameters entered by the user. Any pair of satellites whose distance is greater than this value is disregarded and there is no channel between them. The distance between satellites is determined by the modulus of the vector joining them, formed by the difference of their positions.

To calculate the LoS, the Earth has been the only parameter taken into account as an obstacle, modeling it as a sphere. Moreover, no refraction on the wave propagation caused by the atmosphere has been considered. The propagation has been taken into consideration as a straight line. The calculation of the existence of LoS has been made by comparing angles with one of the satellites as a reference point. The angle formed between the junction of the satellites and the center of the Earth must be greater than the angle formed between the center of the Earth and the point on the Earth's surface where LoS of the reference satellite is tangent. In case the angle between the satellites is the smallest, the distance between the satellites is compared with the distance from the reference satellite to the tangent point, if the distance between the satellites is less than the distance formed by the tangent to the Earth.

The contact model between a GS and a SAT is determined by distance, as well as elevation. To obtain these parameters, two changes of variables are required. First, an intermediate change to the Local Tangent Plane (LTP) coordinates system is required, in particular to the North, East, Down (NED) [52] variant. This coordinate system places the origin of coordinates on the GS. Equation 5.2 describes the method to change to LTP system, its result is the vector $P_{NED}$ which is formed by $N$, $E$, and $D$. The equation uses $P_{xx\_ECEF}$, the position of the SAT or GS in ECEF coordinate system. The rotation matrix ($R_{NED}$) is defined in Equation 5.3, it requires knowing the position GS (the new reference point) in Geographic Coordinate System to kwon latitude (lat) and longitude(lon).

$$P_{NED} = R(P_{SAT\_ECEF} - P_{GS\_ECEF}) \tag{5.2}$$

$$R_{NED} = \begin{bmatrix} -sin(lat)cos(lon) & -sin(lat)sin(lon) & con(lat) \\ -sin(lon) & cos(lon) & 0 \\ -cos(lat)cos(lon) & -cos(lat)sin(lon) & -sin(lat) \end{bmatrix} \tag{5.3}$$

Subsequently, a new change is made to the Azimuth-Elevation Coordinate System, where we obtain the desired values. These are compared with some reference values. The distance has to be less than a threshold distance and the elevation angle greater than a minimum angle. Equations 5.4, 5.5, and 5.6 change the system coordinates from NED to the Horizontal coordinate system. The distance between nodes is $d$.

$$d = \sqrt{N^2 + E^2 + D^2} \tag{5.4}$$

$$azimuth = atan(E/N) \tag{5.5}$$

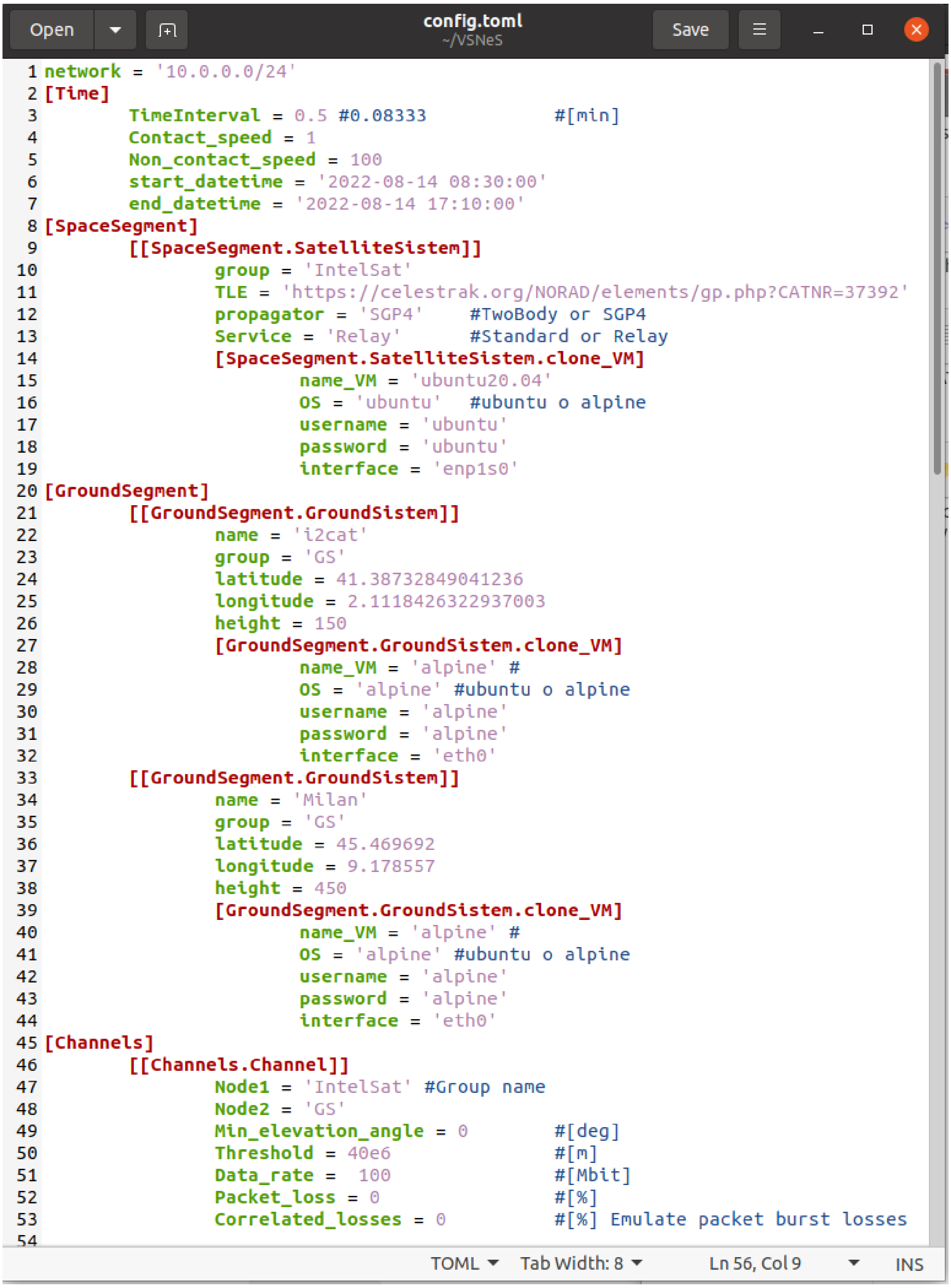
$$elevation = asin(-D/d) \tag{5.6}$$

The calculation of the delay is done by estimating the propagation time from the distance and the theoretical data of the speed of light in vacuum. This results in delays are somewhat lower than those that occur in a real environment, but this does not pose any limitation for the study of protocols. However, in the case of GNSS technology studies, this could be a very significant limitation by oversimplifying the scenario.

## 5.5. Configuration

The configuration is the basis for entering the scenarios and correctly characterizing each element. This is done by means of files that are read at startup and images to clone the machines. The configuration files consist of a main Tom's Obvious Minimal Language (TOML) file and one or more files composed of TLEs.

TOML [53] is a configuration language that tries to be easy to read and write, as well as effortless to parse by a wide variety of languages.TOML allows you to define individual variables and objects with their own variables, as well as vectors. VSNeS defines a set of objects and vectors to define the entire scenario. Figure 5.6 shows a simple scenario with a GS and a GEO satellite.



```
network = '10.0.0.0/24'
[Time]
        TimeInterval = 0.5 #0.08333             #[min]
        Contact_speed = 1
        Non_contact_speed = 100
        start_datetime = '2022-08-14 08:30:00'
        end_datetime = '2022-08-14 17:10:00'
[SpaceSegment]
        [[SpaceSegment.SatelliteSistem]]
                group = 'IntelSat'
                TLE = 'https://celestrak.org/NORAD/elements/gp.php?CATNR=37392'
                propagator = 'SGP4'     #TwoBody or SGP4
                Service = 'Relay'       #Standard or Relay
                [SpaceSegment.SatelliteSistem.clone_VM]
                        name_VM = 'ubuntu20.04'
                        OS = 'ubuntu'   #ubuntu o alpine
                        username = 'ubuntu'
                        password = 'ubuntu'
                        interface = 'enp1s0'
[GroundSegment]
        [[GroundSegment.GroundSistem]]
                name = 'i2cat'
                group = 'GS'
                latitude = 41.38732849041236
                longitude = 2.1118426322937003
                height = 150
                [GroundSegment.GroundSistem.clone_VM]
                        name_VM = 'alpine' #
                        OS = 'alpine' #ubuntu o alpine
                        username = 'alpine'
                        password = 'alpine'
                        interface = 'eth0'
        [[GroundSegment.GroundSistem]]
                name = 'Milan'
                group = 'GS'
                latitude = 45.469692
                longitude = 9.178557
                height = 450
                [GroundSegment.GroundSistem.clone_VM]
                        name_VM = 'alpine' #
                        OS = 'alpine' #ubuntu o alpine
                        username = 'alpine'
                        password = 'alpine'
                        interface = 'eth0'
[Channels]
        [[Channels.Channel]]
                Node1 = 'IntelSat' #Group name
                Node2 = 'GS'
                Min_elevation_angle = 0         #[deg]
                Threshold = 40e6                #[m]
                Data_rate =  100                #[Mbit]
                Packet_loss = 0                 #[%]
                Correlated_losses = 0           #[%] Emulate packet burst losses

```

Figure 5.6: TOML file: example of configuration of two GSs and a GEO satellite constellation.

The first class to be defined is Time. This object defines all the time parameters that interfere with the simulation: TimeInterval, Contact_speed, Non_contact_speed, start_datetime, and end_datetime.

Afterwards, there is the network variable, in which you enter a string with the IP address of the network that will be created in the VLAN, e.g. '10.0.0.0/24'.

The next step is to define the nodes. A class is created for each type, specifically SpaceSegment and GroundSegment. Both contain a vector called SatelliteSystem and GroundSegment respectively. The classes have some variables in common, which are group, Service, clone_VM, and Emulation_startup_script. The group variable defines a set of nodes with the same characteristics about the channels. Service defines the network configuration of the node. Currently, it can be the standard or the relay mode. Clone_VM is an object destined to the definition of the machine to be cloned. The variables of this object define the name, the username, the password, the interface name and the OS. Finally, we find the Emulation_startup_script, a class that defines the name of a script and the parameters to running. VSNeS run that script previous to run the emulator.

Every position of the SatelliteSystem vector contains constellations or sets of satellites with the same characteristics. In addition to the variables mentioned above, SatelliteSystem is made up of the variable TLE and the propagator ones. TLE indicates the name of the file containing the TLE of the satellites to be inserted. This variable accepts both local files and URL addresses. Propagator defines the type of orbit propagator. However, if it is not defined, the SGP4 propagator is used.

The other set of nodes, the GS, are defined by the GroundSegment object, which is formed by the GroundSystem vector. Each position of the vector defines a GS. Each station is defined by a name and a position which is reflected in latitude, longitude, and altitude.

Finally, we found a class to define the channel parameters. The channel class consists of a vector in which each channel is defined. The channels are defined by the groups of affected nodes: a distance limit, the maximum transmission rate, the packet losses, a correlated loss parameter (which allows us to simulate bursts of losses) and finally the definition of the minimum elevation angle. Nevertheless, this only applies to the communications between GS and SAT.

## 5.6. Manager Module

The last module, the satellite communications emulator manager, is responsible for the deployment and the execution of the scenario to be emulated. Specifically, this module has to be able to generate a scenario from a configuration file, which will contain the information of the nodes. Once the information has been analyzed and the nodes have been defined, the program will be able to locate them from a propagation of their orbit, in the case of satellites and subsequently determine the existing connections and the delays associated with the distances. Once the scenarios are defined, QEMU/KVM together with Netem and brctl will be deployed to emulate the network and Cesium will be deployed to represent the position and connections of each node at the simulation time.

For the development of this module, Python 3 has been used together with a programming oriented to objects, creating an interactive software with the user through predefined commands entered by the terminal.

### 5.6.1. Class Structure

During the execution of the program it is necessary to use a set of classes that allows the storage of data, the management of functions and methods. The classes that make up this software are: Scenario, Channel, Time_parameters, Node, Satellite, GroundStation, Orbit, TwoBody, and SPG4.

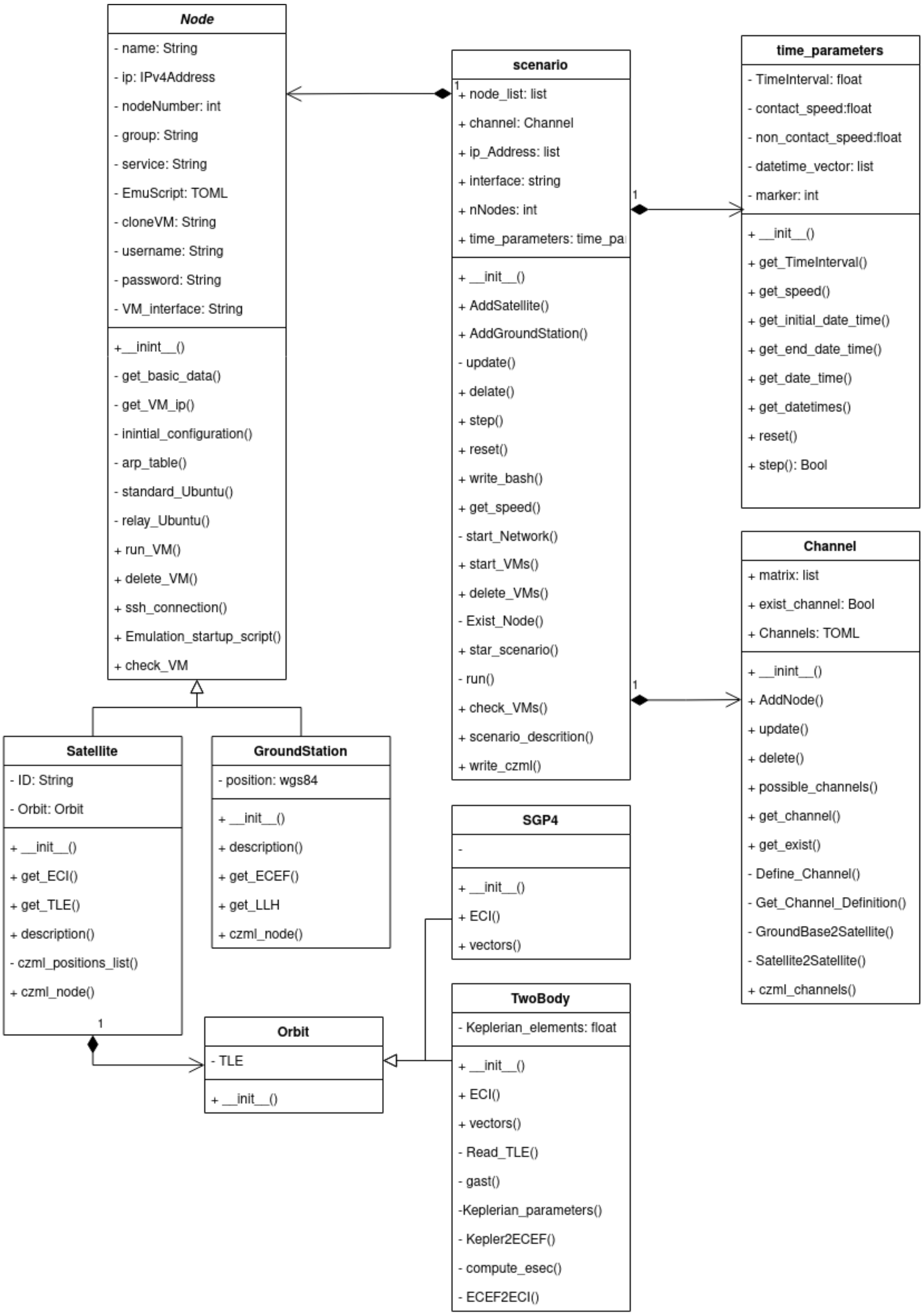


Figure 5.7: Classes diagram of the satellite emulator.

As can be appreciated in the class diagram (as shown in Figure 5.7), Scenario is the main class that manages all the information. Scenario contains a list of Node objects, time_parameters object, and a Channel object, which can be divided into two subclasses: Satellite and GroundStation. Simultaneously, Satellite contains a variable of the Orbit class type, which is divided into subclasses. Currently, only two exist: TwoBody and SPG4, which is the propagator used to define the SAT orbit.

#### *5.6.1.1. Scenario Class*

Scenario is the main class, in which all the data defining the execution is stored. During each execution, there is only one single scenario object that is created when reading the configuration file. The main program only accesses this class where all the other classes are stored, having access to all the necessary information.

**Variables**

The variables of scenario are composed by an object of the other class. This allows classes to easily interact with each other. The Scenario class consists of the following variables:

- *node_list*: is a vector formed by Satellite or GroundStation nodes. This vector contains all the nodes that interfere in the emulation.
- *nNodes*: it determines the number of nodes in the simulation at each instant. The maximum value of this is related by the network mask, e.g. for a /24 mask the maximum value of nodes would be 254.
- *channel*: an object of Channel class that defines all the possible links.
- *time_parameters*: a class object that defines all the values related with the simulation time.

**Methods**

The scenario is composed of the following methods. These allow methods to manage the data and activate the emulation.

- *__int__(self, TOMLfile)*: Constructor of the class, it initializes all the variables with the data of a configuration file written in TOMLT format.
- *AddSatellite(self, sat_tle, constellation)*: it adds a new satellite to the list of nodes and channels. For this purpose it receives the basic information in TOML format and the TLE. The method make a search with the name and the ID of the Satellite in the *node_list* if there is not, it adds it there and update the channel object.
- *AddGroundStation(self, TOML_GS)*: it receives toml data to create a new *GroundStation* object. The method make a search with the name of the Ground Station in the *node_list* if there is not, it adds it there and update the channel object.
- *step(self, EMU:Bool)*: it increases the time marker and updates the channels with the new position of the nodes. The function returns a boolean to determine if the emulation is finished. EMU is a boolean that determines if these updates are performed.

- *reset(self)*: it sets the time marker to the initial position and recalculates the channel.
- *write_bash(self)*: it writes two bash files. One with the configuration needed to run the host as a bridge with emulation in the VLAN interfaces and another to remove all the changes produced by the first file.
- *start_Network(self)*: it checks if the network where the hypervisor connects the VMs is running. And it turns it on if necessary.
- *start_scenario(self, EMU, CESIUM)*: it starts the necessary threads based on the input boleans. If EMU is true, it turns on the emulation part, with the *_run()* method. And if CESIUM is true, it starts a server and opens a web browser with the direction of the server.
- *_run(self, EMU, CESIUM, n_connections)*: it performs three tasks. Firstly, it prints the data regarding the channels. It shows the delay between the nodes. Secondly, it updates the CZML file if CESIUM is true. The update changes the version of the file, the current time and the multiplier clock. Finally, it manages the advance of the time and updates the channels. With the sum of these three tasks, we obtain the emulation process working in parallel with the visualiser.
- *check_VMs(self)*: it checks if all the VMs are on. If someone is off or does not exist, it asks if you want to activate all the VMs.
- *delete_VMs (self)*: it removes all virtual machines related to the scenario. This method is called at the end of the execution of VSNeS. Before deleting the VMs, it makes a check of the existence of these and if they exist, it makes a query to the user to see if he wants to delete them or not.
- *start_VMs(self)*: it executes the startup of all VMs. In case some or all of the machines do not exist, the function creates them. Subsequently, the machines are started and the initial configuration is applied.
- *Exist_Node(self, New_node)*: it checks that there are no repeated nodes. In the case of GS, it only compares by name. In the case of SATs, the name and NORAD ID are checked for matching. None of them can match. In case of coincidence, it returns as True.
- *scenario_description(self)*: it returns a string with a generic description about the scenario: time of execution, number of nodes and information of the nodes.
- *_Emulation_startup_script(self)*: it executes the configuration required for the scenario in each VM. This consists of writing the ARP tables and the script entered by the user.
- *write_czml(self)*: it writes a CZML file in 'Class/templates/ScenarioCZML.czml'. That file describes the scenario around the time and is compatible with Cesium. This method calls to methods of the channel and the nodes to obtain the different packets and writes it in the file.

#### *5.6.1.2. Node Class*

It is the parent class of Satellite and GroundStation. This class has the function of grouping common data between the two subclasses, such as identification data, IP address, and VM information.

**Variables**

The Node class consists of in variables that all the nodes have. All the nodes should have names to identify them and all the data related with the VM and the network.

- *name*: it is the Node name, entered with the configuration file. It identifies the node. It is unique for everyone.
- *nodeNumber*: the int that defines the position in the list of nodes. This count starts in 1 instead of 0.
- *ip*: the address of the VM in the VLAN. Netmask is not included. Every node has a different ip.
- *mask*: the netmask of the network address. It uses a format of 32-bit (e.g. 255.255.255.0).
- *group*: it is the string that defines the name of a group of nodes that have the same channel properties. The groups can be defined by only one node.
- *service*: the string that defines the network configuration used in the node's VM. Currently, this variable accepts the standard and relay modes previously explained.
- *EmuScript*: it is a TOML object with the name of a script and a vector with the inputs. This script is executed when the emulation mode is started.
- *clone_VM*: the string with the name of the VM that is used to create the VM associated with the node.
- *username*: a string with the username of the VM that will be cloned.
- *password*: it is the string with the password of the VM that will be cloned.
- *VM_interface*: the string with the name of the network interface of the VM that will be cloned.

**Methods**

The Node class is composed of methods with the objective of managing a VM. They can create, destroy or configure the guest machine, among other things.

- *get_basic_data(self)*: it returns a string with the name and the IP address of the node.
- *_get_VM_ip(self)*: it returns the IP address of VM that it has in the test LAN.
- *_initial_configuration(self)*: it sends commands to the VM using SSH protocol. By default, this method changes the username to the name of the node and configures the VLAN. Also, it applies configuration scrips of the user.

- *_arp_table(self, node_list)*: it configures the node's VM arp table. For this process, it looks up the MAC of each node and relates it to the IP. The configuration is done using the SSH protocol.

- *_standard_Ubuntu(self)*: it returns a string with the commands for VLAN configuration in standard mode on Ubuntu. This configuration is specific to Ubuntu when working with the sudo command.

- *_relay_Ubuntu(self, nNodes)*: it returns a string with the commands for VLAN configuration in relay mode on Ubuntu. This configuration is specific to Ubuntu when working with the sudo command.

- *run_VM(self)*: it checks the state of the VM and makes the necessary steps to initialize, as well as applying the initial configuration. For example, if there is not a VM with the name of the node, it creates a new one. If the VM with the node's name is off, it turns it on.

- *delete_VM(self)*: it deletes all the data related with the VM. To do it, it turns off the VM with the node's name and then destroys it and deletes its image.

- *ssh_connection(self)*: it opens a new terminal with an SSH connection. First, it tries to introduce the password by itself and if this fails, it makes the user to insert manually.

- *check_VM(self)*:it returns a boolean. It is true if there is a VM with the node's name and is false if not.

- *Emulation_startup_script(self, node_list)*: it executes the script defined in the specific variable via SSH. This function has as input the list of nodes in order to obtain information from each one of them. This is useful because it allows the user to request it as a variable the IP address of one of the unknown nodes. Currently, only IPs address are searched.

#### *5.6.1.3. GroundStation Class*

This class defines fixed nodes on the surface of the Earth. Therefore, it is defined with geographical positions.

**Variables**

The GroundStation class consists of the parent variables and an extra position. The static position of the Ground Stations is the principal feature of this type of node.

- *position*: Skyfield class to define the position. This type of object returns the position in ECEF or LLH with a simple command. It is a static node. For these reason, the position is always the same value.

**Methods**

The GroundStation class is composed of the following methods. These have the function of providing information about the node to the 3D globe visor and the terminal.

- *description(self)*: it returns a string that describes the node. It is composed by the node's name, its IP address and the position (latitude, longitude and height)
- *get_ECEF(self)*: it returns an array with the position of the node in ECEF coordinates.
- *get_LLH(self)*: it returns an array with the position of the node in LLH coordinates (latitude, longitude and height).
- *czml_node(self, datetime_vector)*: it returns a CZML packet which describe the node.

GroundStation also consists of the methods of the Node class.

#### *5.6.1.4. Satellite Class*

This is the class that defines the nodes that represent satellites, so information about their orbit and number of satellites is added to the Node type class.

**Variables**

The Satellite class consists of two variables of its own. One adds information about how to identify the node and the other defines its move.

- *id*: it contains the satellite number information, namely the NORAD Catalog Number. This value is extracted from the TLE entered by the configuration file.
- *orbit*: it is an object of the Orbit class.

Satellite is also formed by the variables of the Node class.

**Methods**

The Satellite class is composed of methods to describe the node and locate it:

- *get_TLE(self)*: it returns a Skyfield's object class that has all the information about the orbit of the satellite.
- *get_ECI(self, datetiem)*: it returns an array with the position at an instant of time in ECI (Earth-centered inertial coordinate).
- *description(self)*: it returns a string that describes the node. It is composed by the node's name, its ip, its NORAD ID and the TLE.
- *czml_position* it returns a CZML data about the position of the satellite in every moment. The calculate position extends from the initial time until the end time plus one period of the satellite. It allows to represent the follow step of the satellite during all visualization.
- *czml_node(self, datetime_vector, results, index)*: it returns a CZML packet which describes the node.

Satellite also consists of the methods of the Node class.

#### *5.6.1.5. Orbit Class*

This class is the parent of the used orbit propagators. Currently, only two propagators are implemented, but the structure of the project would accept the incorporation of new ones as long as they have some common methods.

**Variables**

The Orbit class is formed by only one variable, *TLE*. It is a class of skyfield that knows all the information about the TLE and it can propagate the orbit with the facility. Skyfield allows the propagation of orbits with an SGP4 propagator, so we would only have to apply the functions that this library contains to our variable.

**Methods**

All the subclasses of the Orbit must consist of the following classes:

- *_ECI(self, datetime)*: it returns an array with the position at an instant of time in ECI coordinates.

- *_vectors(self, datetime_vector)*: it returns two vectors, one in ECEF coordinates and the other in ECI. Each position of the vectors corresponds to the location of the satellite at different instants defined by the input vector.

#### *5.6.1.6. SGP4 Class*

This subclass defines a SPG4 propagator with the help of Skyfield module. It gives the necessary tools to define the position and to compare it.

This class takes advantage of the functions incorporated in the TLE object to obtain the location of the satellites. This class acts as a manager of Skyflied's own classes to obtain the necessary information for VSNeS.

#### *5.6.1.7. TwoBody Class*

This subclass defines a Two-Body propagator. In addition to the default methods and variables of the orbit class, this subclass requires more elements to work. It extracts the information from the TLE variable to define its variables and then uses a set of methods to obtain the information required for the general methods of the orbits.

**Variables**

The TwoBody class is formed mainly by the Keplerian elements plus other important data to define the orbit. The different variables that make up these class are listed below:

- *ToA*: a float that defines the time of applicability. It defines the time to which all the time-varying fields in the element set are referenced. This parameter is in days with respect to the beginning of the year.

- *ToAyear*: an int that defines the year over which the ToA is defined. It defines the last two digits.

- *a*: a float that defines the orbit major semi-axis. This parameter is in meters.

- *i0*: a float that defines the Orbit inclination. This parameter is in radians.
- *e*: a float that defines the eccentricity of the orbit.
- *Omega*: a float that defines the length of the ascending node at ToA. This parameter is in radians.
- *w*: a float that defines the argument of the perigee in the ToA. This parameter is in radians.
- *M0*: a float that defines the mean anomaly at ToA. This parameter is in radians.
- *n*: a flaot that defines the satellite mean motion. This parameter is in radians per second.
- *GAST*: a float that defines the right ascension of the Greenwich meridian at the ToA. This parameter is in radians.

**Methods**

The TwoBody class is defined not only by the basic methods of an orbit class, but also by the following methods that allow the calculation of the position.

- *_Read_TLE(self, Line1, Line2)*: it returns the elements of interest for the propagation of an orbit defined in a TLE (ToA, i0, Omega0, e, w, M0, n, ToAyear). These data is returned to the same format as in the TLE.
- *_gast(self, ToA, ToAyear)*: it returns the GAST in radians from the entries of the application time in days with respect to the beginning of the year and the last two digits of the year in which it is applied.
- *_Keplerian_parameters(self, i0, Omega0, e, w, M0, n, GAST)*: it returns the variables needed to calculate the position of a satellite with the Two-Body propagator in a format adapted for direct calculation (meters and radians).
- *_Kepler2ECEF(self, date_time)*: it returns the position in ECEF coordinates at a precise instant of time following the algorithm specified in Annex A.
- *_compute_esec (self, date_time)*: it returns the time elapsed between the ToA in days and the instant at which the object's position is to be calculated.
- *_ECEF2ECI(self, ECEF, date_time)*: it returns the position in ECI coordinates from an ECEF position and the time instant. This is achieved by applying the rotation matrix of Equation 5.1.

### *5.6.1.8. Channel Class*

Channel object has one of the most important tasks in the emulator. It is responsible for determining the LoS between nodes and their distance to create a matrix with the information of the channels that the emulator has to create.

**Variables**

The Channel class consists of the following variables:

- *delay_matrix*: a square matrix of $N_{Nodes} * N_{Nodes}$ defining the delays between nodes in ms. It relates the nodes according to the position they occupy in the node_list vector. It is noteworthy that the main diagonal is formed by 0 and the positions of the nodes that do not have LoS between them are defined with a -1. In addition, if the channels are not possible between the nodes, the position is defined by a -2.

- *exist_channel*: it is a boolean which is true if there are some pairs of nodes with a possible link.

- *Channels*: an object of type TOML that stores the information of the possible channels defined in the configuration file. This variable works as a vector that defines a channel in each of its positions. The channel data consists of the groups that are connected, the distance threshold and the maximum transmission rate or the losses, among others.

### Methods

The channel class is composed of the following methods:

- *AddNode(self, node_list, nNodes, date_time)*: it adds a new node to the *delay_matrix* calculates the possible channels between the new node and the rest of nodes and writes it in the matrix. If *exist_channel* is false and there are some new position outside the diagonal different of -1 or -2, its state will change to true. If not, it will remain unchanged.

- *update(self, node_list, nNodes, marker, EMU)*: it reloads the variables for a new instant. To make that process, firstly, it copies the *delay_matrix* and it defines the *exist_channel*as false. Then, it makes a loop inside others to compute the delay between all the nodes. Afterwards, it computes the delay of a pair of nodes. It will save in the *delay_matrix*. Also, it will change the configuration of NetEm for the link. It will only pass if the EMU boolean is true and if the old value is different from the new one. If one value of the new matrix without the diagonal is different from -1 or -2, it changes the *exist_channel* to true.

- *get_channel(self,node1, node2)*: it returns the *delay_matrix*. It can give a specific position, line or row or the complete matrix.

- *get_exist(self)*: it returns the *exist_channel* boolean.

- *_Define_Channel(self, node1, node2, marker)*: it returns the delay between node1 and node2 in *date_time* instant. Its work is to call the other methods that compute the delay. To call the correct methods, it checks the type object that is every node and decides.

- *_Get_Channel_Definition(self,node1,node2)*: it returns the channel information linking the two nodes entered. If the connection is not possible, it returns none. It compares the groups of the nodes with the groups that interact in each channel by means of a loop and in the case of coincidence it returns all the data of that specific channel with a variable of type TOML.

- *_ECEF2NED(self, pseudoDistance, LLH)*: it returns an array in NED coordinates of the position of a satellite with respect to a GS. To do so, it applies the rotation matrix shown in Equation 5.3 with the position in LLH of the GS and the vector that forms the union between the SAT and the GS.

- *_NED2AzimuthElevationDistance(self, NED)*: it returns an array in the Azimuth Elevation Coordinate System from an array in the NED system. It applies Equations 5.5, 5.6 and 5.4.

- *_GroundBase2Satellite(self, SAT, GS, MinAngle, threshold, date_time)*: it returns the delay between a satellite and a GS in a specific moment. Firstly, it computes the existence of direct link. The existence of the link depends on elevation angle and the distance. Elevation angle has to be above a minimum value, while the distance has to be below the threshold. If this is fulfilled, the delay is computed. If not, the method returns a -1.

- *_Satellite2Satellite(self,ECI1,ECI2,threshold)*: it returns the delay between two satellites. It computes the existence of direct link comparing angles. That angles are formed by the line between the Earth's center and one of the satellites in relation with the Earth's surface and the other with the second satellite. If the second angle is greater than the first one, it will be LoS. It also takes into account the threshold. If all this is fulfilled, the delay is computed. If not, the method returns a -1.

- *czml_channels(self, datetime_vector,node1,node2,results, index)*: it returns a CZML packet which describe the channel between two nodes during the entire simulation time. If during the simulation time there is no connection, the channel CZML would be None.

### *5.6.1.9. Time Parameters Class*

The *time_parameters* class has all the data related with the simulation time and methods to manage it. It is important because most of the calculations needs to know the date.

**Variables**

The time parameters class consists of the following variables:

- *TimeInterval*: is a timedelta object which defines the elapse time between the steps of the emulation. It is defined with the configuration file with a value of minutes. It only takes values greater than 0.

- *contact_speed*: is a float which defines the rate of execution when there is an operative channel. The execution is repeated following Equation 5.7. It only takes values greater than 0. The typical value will be 1, that represent a realistic move of the scenario.

- *non_contact_speed*: is a float which defines the rate of execution when there is not any operative channel. The execution is repeated following Equation 5.7. It only takes values greater than 0. And it typically will take values greater than *contact_speed* value. This item has the objective to accelerate the emulation in the instants when there isn't representative situations and makes faster the test.

$$elapse\_time = TimeInterval[seconds]/Speed \tag{5.7}$$

- *datetime_vector*: is an array of datetime objects. The software defines it with an initial and end times and the *TimeInterval*. It defines all the instants in which the manager make its operations.

- *marker*: is an int that determines the position of the *datetime_vector* where the emulation is.

### Methods

The time parameters class is composed of the following methods:

- *get_speed (self, channel)*: Returns the *contact_speed* or the *contact_speed* depending on the incoming bolean. For example, if *channel* is true, it will return *contact_speed*.

- *get_datetimes(self)*: Returns *datetime_vector*.

- *get_initial_date_time(self)*: Returns the first position of *datetime_vector*.

- *get_date_time(self)*: Returns the position of *datetime_vector* that *marker* says.

- *get_end_date_time(self)*: Returns the last position of *datetime_vector*.

- *get_TimeInterval (self)*: Returns a float which represent the value of *TimeInterval* in seconds.

- *reset(self)*: overwrite the marker with the initial position, 0. The software uses this method when the emulation arrive to the final or the user stop it.

- *step(self)*: increases the *marker* in one unit. If the marker get a value greater than the length of *datetime_vector*, it returns true boolean, in the other case it returns false.

## 5.6.2. Python Modules

One of the advantages of using programming languages such as Python is the amount of available libraries. We found in them functionalities for all we can think. Those libraries have two principal benefits, we avoid time spent in a work that has been done already by someone else and usually the code is optimized.

The software manager take profit of some modules that are integrated in Python for default and some other that have to be installed. Between the first type, it implements: datetime, math, multiprocessing, paramiko, subprocess, sys, threading, time, webbrowser.

The **datetime** [54] module supplies classes for manipulating dates and times. It save dates and times and permit to make operation between them and express them with different formats. The manager give the *datetime* class to represent dates and *timedelta* class to add instants of time to the dates.

**Math** [55] module provides access to mathematical functions with real numbers. It allows us to perform trigonometric operations (cosines, sines, etc.) and square roots.

**Multiprocessing and threading** [56] [57] modules have the goal to manage threading processes. They are similar but offer some custom qualities. Multiprocessing is useful because it allows to shout down a thread. In contrast, threading give facilities to know the number of thread that are open and obtain an object when the process is over.

The **paramiko** module [58] implements the SSH protocol, providing both client and server functionality. We use this module to configure the VMs once they are open.

The next module, **subprocess** [59], allows us to spawn new processes, connect to their input/output/error pipes and obtain their return codes. We use it to send bash commands to the host (e.i. configure the network interfaces or use the virsh commands).

System-specific (**sys**) [60] is a module which provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. We integrate it to an aesthetic purpose. In concrete, to overwrite lines in the terminal. It permits a clean vision of the data without a chaotic flux of numbers.

The **time** [61] module provides various time-related functions. The manager software takes advantage of two of these functions. First, the software use the module to sleep the program. It is necessary to control how advance the time during the emulation. The other use is measuring the execution time, that allows us to adjust the sleep time. This setting is made by subtracting the execution time to the desired elapse time.

The last integrated module is **webbrowser** [62]. It provides a high-level interface to allow displaying web-based documents to users. In our case, it opens us a web browser with the direction of our server to visualize the 3D virtual globe GUI.

The Python libraries that must be installed to run are skyfield, toml, czml, flask, julian, and astropy. Each one provides different functionalities that guarantee the operation of the program.

**Skyfield** [50] provides tools to make operations with the position of the starts, planets, and satellites in orbit around the Earth. In our case, we only apply a small part of the library. We use the tool for three functionalities: TLE file reading, coordinate system conversion, and SPG4 propagator implementation.

To read files, the *load* class is used together with its *tle_file()* method. It reads a local or online file and returns a list of objects of the *EarthSatellite* class. This method is in charge of discarding wrong formats, avoiding the introduction of incorrect data that may alter the program operation.

Once we have the EarthSatellite classes defined by the TLE, we only have to match the class with time instants, obtaining the position for that precise instant. The library works with its own time variable, so from the objects defined with the datetime class, new objects of the timescale type are created.

```
ts = load.timescale().from_datetime(datetime)
```

Once the time variable is defined as ts and the EarthSatellite class as satellite, the position in ECI, ECEF and LLH is obtained from the following commands.

```
geocentric = satellite.at(ts)
#Print ECI coordinates
```

```
print(geocentric.m)
p = wgs84.geographic_position_of(geocentric)
#Print ECEF coordinates
print(p.itrs_xyz.m)
#Print LLH coordinates
print(p.latitude.degrees, p.longitude.degrees, p.elevation.m)
```

These positions are obtained through the propagation of the SGP4 orbit, from the SGP4 Python library that is incorporated into Skyfield.

Finally, it is used to perform coordinate system changes in ground stations with a wgs84 ground model. These changes were previously introduced with functions, but the addition of the library makes the saving of variables more efficient and allows a change of geoid from a few modifications.

The **TOML** [63] library allows parsing and creating TOML files. This allows the use of these configuration files by making it very easy to implement changes to the format, unlike creating your own configuration file layout.

In the current version, only the library is used for reading the data. The creation of these files is no longer used. This has not been included as it is not possible to change parameters from the software, but all changes are applied directly to the configuration file.

The use of this library is simple, first a TOML object is loaded from the configuration file:

```
#Open file
fo = open('config.toml', "r")
TOML = toml.load(fo, _dict=dict)
```

This object contains all the information stored in the file. In order to analyze its content, the object is called by specifying between square brackets the name of the desired variable. The following shows how to call an object or a variable within an object:

```
NetworkTOML = TOML['Network']
Network = TOMLfile['Network']['network']
# or
Network = NetworkTOML['network']
```

TOML allows a fast and efficient reading regardless of the number of configuration data. Avoiding the creation of reading codes that parse the various possibilities in the configuration file and become complex as new configurable data is included.

In addition to the ease of use of the toml files, the use of such libraries for configuration makes the software easier to extend by adding new functions or data to be considered.

The **CZML** module [64] is an open source Python library to write and read CZML files. It works with a class-based information storage system, which is then translated into CZML format. This library allows a lot of the correct format adaptation and speeds up the incorporation of new data in the file or the modification.

The libraries **julian** [65] and **astropy** [66] are used to run the Two-Body propagator. Julian module change a date time in ISO format to Julian date and vice versa. And astropy compute the angular rotation since epoch of a specific date time.

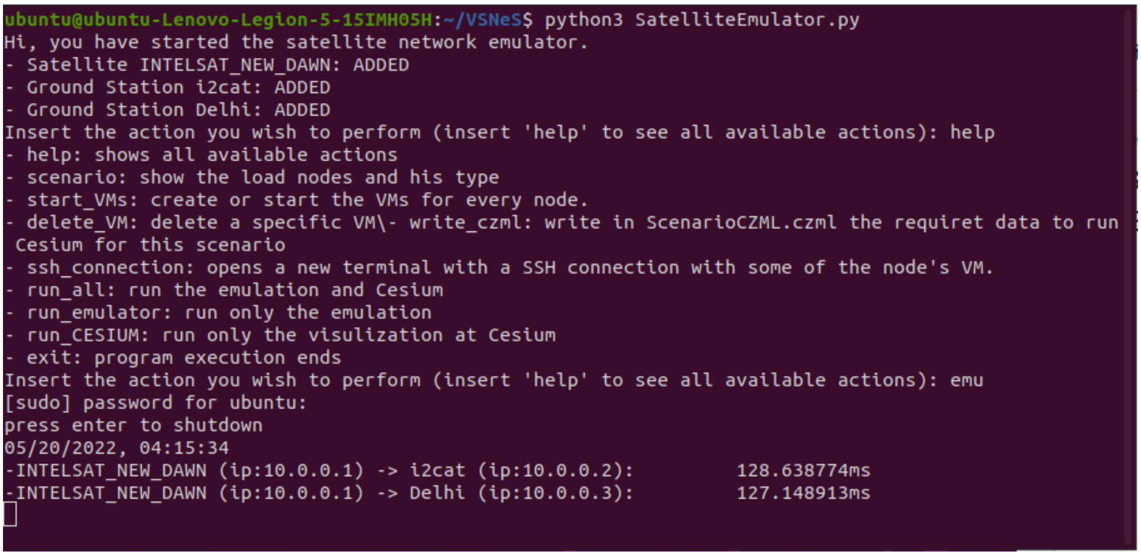
The last library that VSNeS uses is **Flask** [67]. It gives tools to create a web server in combination to HTML templates. Specifically, Flask generates a local server on the desired port and manages the call of the templates and the information entered into them.

### 5.6.3. Command-Line Interface CLI

VSNeS has a CLI, named *SatelliteEmulator.py*, consists of a Python program that works with the classes mentioned above. The program offers to the user a simple and intuitive interaction through terminal commands, with which the user can activate the different modules. It calls the principal public methods of the Scenario class, which make the management of the emulator. The CLI is the only access to the emulator for the users.

When the user runs the Command-line interface, it will read *config.toml*. In this file there are information about the time parameters, the satellites, and the ground station that the user would complete before of running the program. With the data of the file, the program configures all the parameters of the class and wait new orders. Before starting the emulation it is advisable to create the VMs of the specific configuration and write the CZML file, which we need to the visual representation. This two processes have a heavy computational cost and a reasonable amount of time is needed to complete them. For this reason, the users can decide if they need some of them.

The program have three types of execution modes(*run_emulator, run_cesium, run_all*). The first mode provides connection between the nodes, if applicable. Also, the terminal shows the hour that is represented and the delays between all the nodes. The next mode give only information, it opens a server with an interactive representation of the scenario. The user can control the time and see the connectivity of the nodes in every instant. The final mode is the most complete one, it combines the previous ones. In this case the user can not control the time of the GUI, it is our software who manages it.



```
ubuntu@ubuntu-Lenovo-Legion-5-15IMH05H:~/VSNeS$ python3 SatelliteEmulator.py
Hi, you have started the satellite network emulator.
- Satellite INTELSAT_NEW_DAWN: ADDED
- Ground Station i2cat: ADDED
- Ground Station Delhi: ADDED
Insert the action you wish to perform (insert 'help' to see all available actions): help
- help: shows all available actions
- scenario: show the load nodes and his type
- start_VMs: create or start the VMs for every node.
- delete_VM: delete a specific VM\- write_czml: write in ScenarioCZML.czml the requiret data to run
 Cesium for this scenario
- ssh_connection: opens a new terminal with a SSH connection with some of the node's VM.
- run_all: run the emulation and Cesium
- run_emulator: run only the emulation
- run_CESIUM: run only the visulization at Cesium
- exit: program execution ends
Insert the action you wish to perform (insert 'help' to see all available actions): emu
[sudo] password for ubuntu:
press enter to shutdown
05/20/2022, 04:15:34
-INTELSAT_NEW_DAWN (ip:10.0.0.1) -> i2cat (ip:10.0.0.2):         128.638774ms
-INTELSAT_NEW_DAWN (ip:10.0.0.1) -> Delhi (ip:10.0.0.3):         127.148913ms
```

Figure 5.8: CLI demo: example of a scenario with two GSs and a GEO satellite.

When the program is closed, it gives the option to delete the used VMs. It aims to avoid maintaining VMs that require a lot of storage and in the same time give the opportunity of maintaining VM that will be used. Every VM needs a several amount of time to be created, so if they exist, the start-up time of the emulator is really reduced.

If the users want to apply some changes in the configuration, they will have to close the program and modify the files. That may seem a problem, but it is more intuitive how to make the configuration because there is only one way. And if the modifications do not include add/remove nodes, the emulation can be run instantly.

Figure 5.8 shows the VSNeS' CLI with the help and run_emulator command running. The following list shows the commands that the user can interact:

- *help*: shows a brief description about all the commands that the user have available.
- *scenario*: prints in the terminal the main information of the loaded scenario. It shows the interval of emulation, the number of nodes and the nodes with information about their IPs address and position. The command have the function of validate that the loaded data is correct.
- *start_VMs*: makes all nodes have their VM operational. It creates the nodes' VMs that have not existed yet and turn on the rest if necessary. After that, the software configures the VMs.
- *delete_VM*: when the user selects this command, the program ask him what VM he wants to delete. The user can answer with the name of one of the nodes, with all, or with exit. If the answer is some name, the software will delete only the VM with that name. But if the user wants to delete all, the answer have to be all. In the case that the answer is exit, any VMs will be deleted.
- *ssh_connection*: opens terminal which have SSH connection with the nodes. Before, it asks which VM to connect to. The users have the options of open a specific one, all or anyone. The first time that this connection will do it, the user will have to set the password manually, but the rest of the times the software will have the enough credentials to the connection.
- *run_emulator*: runs the emulator. The host creates a bridge and new interfaces to connect the VMs between them. The software change the channel characteristics with NetEm every time that the clock of the emulation is ticking and there are changes. The terminal is updated in real time with the delay of the links. The process finishes when the user press 'Enter' or the timer reaches the end.
- *run_cesium*: makes a server in a new terminal and opens a web browser with the direction of the local server. The server provides of a representation of the scenario in Cesium where the users can control the time. The program asks if the CZML needs to be written. In the current version, the user has to close manually the server with the new terminal.
- *run_all*: combines the emulator with the 3D visor. It manages the emulator like in the emulator mode and opens a server with a similar service than the Cesium mode. The difference is that in this case, the user can not control the emulated time. The emulator part and the Cesium part have to be stopped with different ways. The emulator is finished when the user press 'Enter' and the Cesium server when the user press 'Ctrl+c' in the new terminal.
- *exit*: exits the program. At this moment, the user has the option to delete the VMs or not.

## 5.7. Execution Process

To start using VSNeS it is necessary to follow a series of steps in order to have all the dependencies installed and the necessary files. Foremost, it is necessary to install the VSNeS directory, where all the required code is located and where the configuration file has to be placed. Once the download is done, we enter the directory:

```
cd VSNeS
```

To facilitate the installation process a script has been incorporated to solve all the dependencies. Specifically, it installs the hypervisor, the Python libraries, tools to manage the bridges and functions for SSH. To run the script, follow the command below:

```
./install.sh
```

Once everything is installed you have to generate virtual machines, so that VSNeS has an image to clone and replicate the features. First, it is necessary to download a Linux ISO from any distributor. Then proceed to deploy the VM and install the OS. This process can be done through commands or through the GUI of the hypervisor. The most intuitive method is through the GUI, *virt-manager*, clicking on the button to create a new virtual machine and entering the data that are requested.

The images to be used for cloning must have a set of programs installed. The standard execution mode requires the installation of the openssh-server and net-tools package. Openssh-sever allows the connection of VSNeS to the VM through the SSH protocol and net-tools adds tools for the configuration of network parameters. In case you want to implement the relay mode it is also necessary to install bridge-utils, in order to create a hub. The following is an example of the commands needed to configure an Ubuntu image:

```
sudo apt install openssh-server
sudo apt install net-tools
sudo apt install bridge-utils
```

At this point, the only thing left to do is to modify the config.toml file. Here you specify the scenario to be represented. The configuration may depend on other files such as TLEs or scripts. It is important that these are inside the folder or their path is specified when entering their name.

Once the above steps have been completed, the system is ready to run the program. Currently, only a CLI version is available, which is started by:

```
python3 SatelliteEmulator.py
```

Once the software is started, all the commands listed in the CLI section can be executed, but it is recommended to follow a specific methodology.

Firstly, it is interesting to launch the command run_cesium with which we visualize the scenario and allows us to see quickly if the scenario has been loaded well and if the time interval collects the data of interest.

The next step is to start all the virtual machines, start_VMs. This generates and activates the VMs which is a costly process but will not have to be repeated for every run. Besides activating them it is interesting to connect via SSH having an efficient connection without wasting resources for graphical interfaces. The command is SSH_connection and allows the connection with one or all VMs.

At this moment the software is ready to run the emulation and make the desired measurements.

# CHAPTER 6. PRELIMINARY TEST RESULTS

During the process of the development of the software, we have performed four tests to determine if VSNeS works correctly and assess the limitations of the software. These tests were performed as the software development progressed and revisions were made to already tested items when significant changes were made.

Prior to the tests, a review tool has been implemented. It consists of a test script based in unittest [68]. This script runs all the functions of the program and verify if they do what is expected. This tool is useful to search bugs when part of the code is changed.

## 6.1. Test 1: Node position

The nodes' position is one of the key points in the emulator because we use this data to compute the link characteristics. It is important that when we use the location, all of them make reference to the same instant and in the requite coordinate system.

During the development of the project, we have implemented two orbit propagators. So both systems would have to pass verification. Despite this concept, the SGP4 propagator has been introduced through a library with its own validation.

Having a validated tool makes it easier to check our implementation. It is only necessary to compare the results and the error introduced in them. In addition to the comparison between the two propagators used, an extra library, ephem, has been used. Ephem [69] is a Python library that implements a SGP4 propagator.

Then we compared the results of the different libraries during different time instants. All of them give us similar values of ground track with a small deviation of 2 degrees in our code, something that is normal because we implement a Two-Body propagator and not a SGP4. Also, we compare the ground track (lat, long) with the visualizer of CelesTrak. For a deeper study of the differences between the libraries and the Two-Body propagator, the absolute error of the position in different coordinate systems has been analyzed. Figure 6.1 shows the absolute error of the different axes in ECEF and ECI. As can be seen, the absolute error between the propagators in ECEF is practically null, offering values close to 0 at all times. In contrast, in the case of the ECI system there are large errors, but they occur at specific times and only on one of the axes. Since the error only occurs on one of the axes, it does not represent a large deviation from the position determined by the SGP4 propagator. It should be noted that the error in the z-axis is practically null, this is due to the fact that in the change from ECEF to ECI this axis is not modified.

The last validation of the position is with the visualizer, it is useful to a last check of the before points in addition to see if the GUI works well, but this test is done it in Test 3.

## 6.2. Test 2: Delay and Connectivity

The second test is focused on the validation of the channel definition. The test checks the calculations and how they are reflected in the emulator.

To check the definition of the channel availability between nodes, we have to verify the

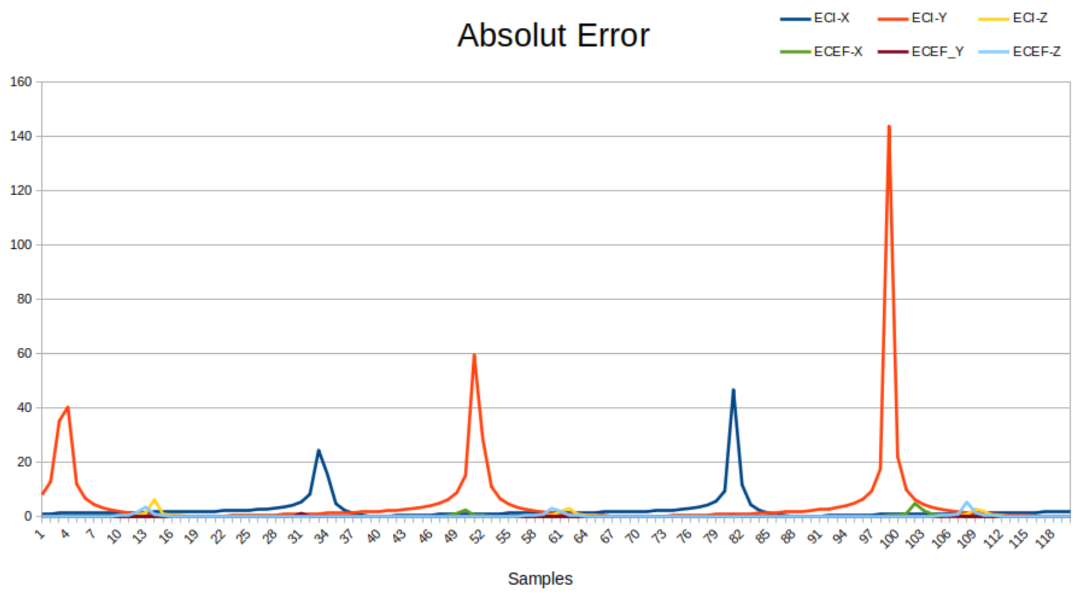


Figure 6.1: Absolute error between the position values of a satellite obtained by our Two-Body propagator and Skyfield's SGP4 propagator.

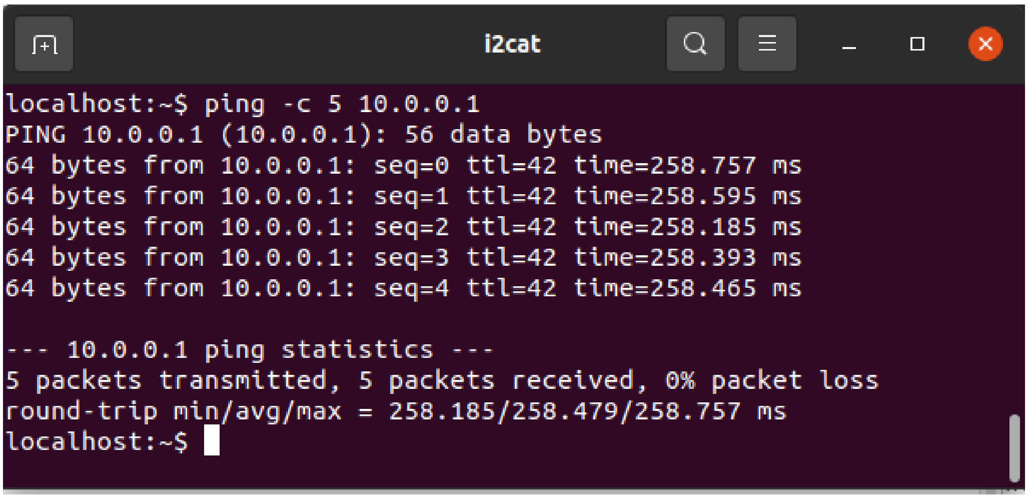
distance and LoS between the nodes. A first iteration of tests was carried out in which we could predict the outcome of the input date. For example tests with a configuration of a really small threshold, satellites on opposite sides of the Earth, a satellite over a ground station, etc. Similarly to Test 1, we compared the results of our code with the values provided by Skyfield, verifying that we obtained the same parameters.

Once we have made computational checks, we compared the results of the delay with the theoretical values of RTT. For this, two tests have been carried out. A first one with a GEO satellite transmitting to a LEO satellite and a GS, and a second test comparing the connection between a GS and a LEO orbiting satellite.

These tests have been performed both by comparing the calculated values and by running the simulation, verifying that the delays between the machines are correctly applied.

The first test allows to verify delay values and the system that determines the communication between satellites. For this test a GS was placed on the i2Cat Foundation headquarters in the Nexus I building in Campus Nord UPC in Barcelona (latitude = 41 ° 23' 15.2" N, longitude = 2 ° 06' 43.1" E, and height = 150 m (above sea level)) and two satellites were used: ONEWEB-0413 as LEO and INTELSAT NEW DAWN as GEO.

Pings have been performed from and to the GEO satellite to check if a connection is established and the delay values have been measured. Theoretical RTT values for GEO satellites exceed 240 ms, this value being an approximation of the delay with the minimum distance between the GEO satellite and the closest point on the Earth's surface (approx. 36 000 km). For ease of calculation, considering that a GEO satellite is able to cover half of the Earth's surface, this means that the maximum distance of its link will be $\sqrt{(36000+6371)^2+6371^2}=42847$ km, which means that the RTT cannot exceed 285.6 ms, if only propagation delays exist. During the test we obtained delay values between the GS and the satellite of 258 ms (as shown in Figure 6.2), which is in the range of theoretically, the distance between the Satellite and GS is over 38 580 km. On the other hand, the connection between satellites provides a similar delay that changes with time due to the relative movement, and ends up losing the connection.



```
localhost:~$ ping -c 5 10.0.0.1
PING 10.0.0.1 (10.0.0.1): 56 data bytes
64 bytes from 10.0.0.1: seq=0 ttl=42 time=258.757 ms
64 bytes from 10.0.0.1: seq=1 ttl=42 time=258.595 ms
64 bytes from 10.0.0.1: seq=2 ttl=42 time=258.185 ms
64 bytes from 10.0.0.1: seq=3 ttl=42 time=258.393 ms
64 bytes from 10.0.0.1: seq=4 ttl=42 time=258.465 ms

--- 10.0.0.1 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 258.185/258.479/258.757 ms
localhost:~$
```

Figure 6.2: Ping of i2Cat GS to INTELSAT NEW DAWN

For the second test, the same scenario has been selected, excluding the GEO satellite. In this scenario ONEWEB-0413 flies over i2Cat, so it is possible to check the maximum and minimum RTT, and also to check the contact model between a ground station and a satellite. The maximum and minimum RTT values are 12.1 ms and 4.69 ms, respectively. The maximum RTT is higher than the threshold establishes, but it is only 2 ms higher than can be produced by the processing time and the latency of the hypervisor connection, while the minimum value reflects a minimum distance of 703 km ($\frac{4.69*10^{-3}s}{2} * 3 * 10^8 m/s$) which is consistent with LEO satellite orbits.

## 6.3. Test 3: GUI

The test consists of checking that the information is displayed correctly plus that there is synchronization between the emulator and the viewer. As mentioned previously, with the GUI verification we can reinforce the validation of the node positioning. This was performed with a scenario composed with two satellites (ONEWEB-0413 and STARLINK-2438) and one ground station (i2Cat) on May 20, 2022, at 4:30 a.m. The validation consists in the comparison of our GUI with the CelesTrak GUI and with Google Maps. As we can see in Figure 6.3, our program draws with exactitude the ground stations. Figure 6.4 shows the comparative between our visor and the CelesTrak ones. It is important to validate that the TLE introduced is the current one, because it is the one CelesTrak uses.

The next test is the representation of one scenario to check if the delay and the disconnections between VMs corresponds with the showed data on the terminal and on the GUI. The scenario selected is the mentioned before, two satellites (ONEWEB-0413 and STARLINK-2438) and one ground station (i2Cat) on May 20, 2022, at 4:30 a.m since 5:30 a.m. The TLE used in these test are older than the previous ones. During the development we select this scenario to make all the checks, but we did not capture with the CelesTrak.

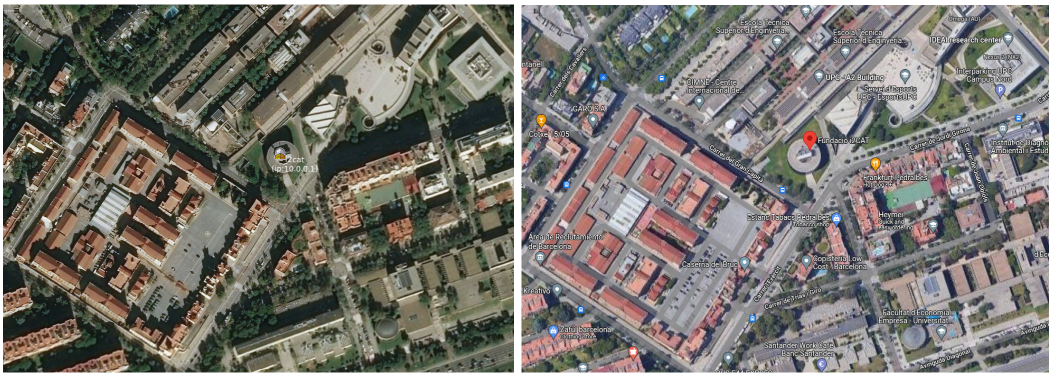


Figure 6.3: Comparative between our visor against Google Maps locating the i2Cat offices.

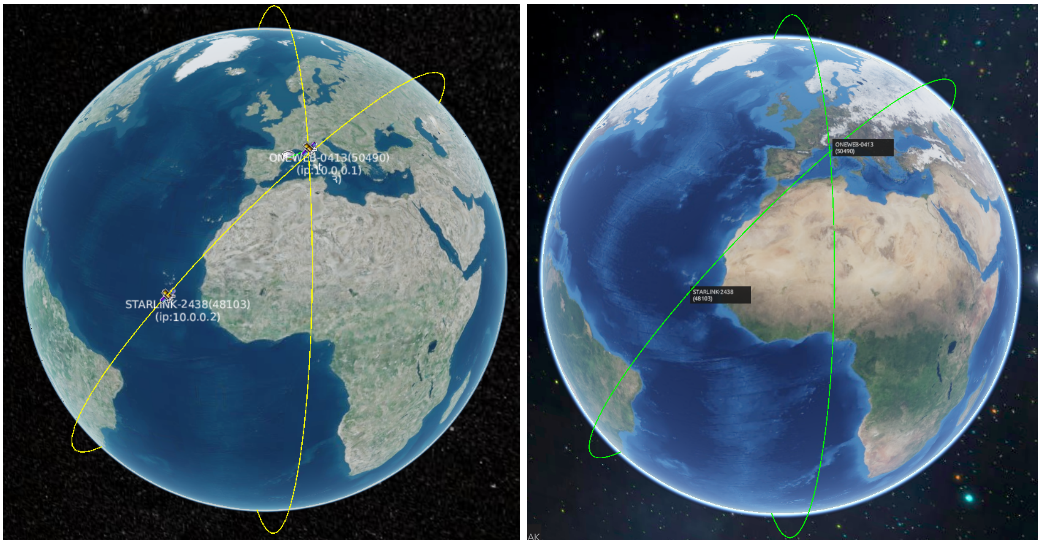


Figure 6.4: Comparative of the representation of ONEWEB-0413 and STARLINK-2438 orbits in our visor and in the CelesTrak ones using the same TLEs.

```
ONEWEB-0413
1 50490U 21132X   22139.91667824  .00039947  00000+0  38757-2 0  9990
2 50490  87.2156 305.4021 0004148 137.5916 203.0744 14.92471037  1305
STARLINK-2438
1 48103U 21027M   22139.92055824 -.00010023  00000+0 -66961-3 0  9998
2 48103  53.0584 270.5355 0001327  58.7906 301.3212 15.05585520 63229
```

We make pings from the different node terminals to the rest of the nodes. With these pings we should observe that there is no connection when there is no green link between the nodes in the GUI and that the delay of the packets is double that indicated in the terminal as expected, since the terminal indicates the delay in one direction and when pings are performed we obtain the RTT.

After the test, the software and its correct operation could be validated. Despite this, sometimes the emulator works with a short delay with respect to the GUI, which is negligible for our purposes. This is because Cesium stores all the channel information and simply displays it, while the emulator calculates the channel matrix in real time, so it may take a few ms for the new channel configuration in the emulator.

## 6.4. Test 4: Performance

Following the set of operational tests designed to ensure that the program is working properly, a performance test has been designed. This test is responsible for analyzing the effect of the execution on the host computer and its limitations.

For this test, it is important to contextualize that the data has been extracted from a single computer, so the data should serve as indicative information. The computer in question is a Lenovo Legion 5 5IMH05H. The laptop has 15.5 GiB of RAM, an Intel® Core™ i7-10750H CPU @ 2.60GHz × 12, a NV166 / Mesa Intel® UHD Graphics (CML GT2) graphics card, and a hard disk capacity of 512.1 GB. It has worked on a partition of about 170 GB with a 64-bit Ubuntu 20.04.0 LTS operating system.

The software has been tested using VMs with two different OS. We used Ubuntu 20.04 in its desktop version and the Alpine OS, which only includes a CLI version. In this way, it is possible to analyze the capabilities of the program using very heavy machines versus light machines.

Each of the OS has been subjected to two data collections. The first scenario is the simplest one and consists of 3 VMs: two GS and a satellite. In the second case, we tried to push the program to its limits by generating the maximum number of VMs that the partition was capable of storing.

As mentioned above, the partition has 170 GB of storage. After the installation of the operating system and various programs as well as the creation of VMs for cloning, the available storage is 132 GB. This value and the size of the machines will determine the maximum number of VMs that can be created.

The Ubuntu machines, heavier VMs, consisted of 2 CPUs, a RAM memory of 4 GiB, and a storage size of 10 GiB. This means that the maximum number of VMs that the test computer is capable of storing is 13. On the other hand, the Alpine machines are light VMs. In our case they consist of a CPU, 768 MiB, and a storage of 4 GiB. This allows a maximum number of VMs of 33.

For data collection, we have used the Python3 script, *sdrPerformance.py*. This script extracts only information about the RAM usage and the workload of each of the 12 CPUs. The script allows storing performance data with an adjustable periodicity in a csv file. A period of 1 second was used for data collection.

The operation of the script is based on the collection of data from two main commands: *top -n 1 -b -1* and *free*. The first command provides information about the CPU usage, while the second one shows how the RAM memory is being used.

Once the scenarios for which measurements are to be obtained and their method of obtaining them have been established, it is necessary to define some rules to ensure that the tests are comparable with each other. Firstly, a configuration has been established in which the emulator defines the channel characteristics once every second. Then we defined the initial characteristics, where no VM related to the scenario was created and no other VM was turned on. In addition, to maximize resources we have closed all external programs to VSNeS or to the data collection, such as web browsers. Finally, a set of steps to be performed during the simulation has been defined:

1. **Execution of VSNeS:** The program is executed using Python3, at this moment the

configuration file is loaded and the position of the nodes is calculated for all the time instants of the simulation.

2. **Virtual machines' startup:** The command to activate the virtual machines is executed. As no virtual machine has been created, the software performs the creation and start-up one by one.

3. **SSH connection:** The command is used to open terminals with SSH connection to the VMs. This allows later pings between the machines.

4. **Running the emulator:** Only the emulator is started to measure how the system is loaded. During the execution, pings are made to verify if it causes any effect.

5. **Full execution:** This option requires a first write process. This allows us to first determine the consumption in writing the CZML file for the visualization of Cesium. Once the file has been generated, we can check the effect of the creation of a server and its visualization. As in the previous case, pings are made to check if they have a relevant effect.

6. **Shutdown and delete:** Once the previous tasks have been performed, VSNeS execution is finished. At this point, VSNeS gives the option to delete the generated VMs, a process that we also perform. This way you can check the shutdown time and if the initial values are restored.

The graphs of the four data collections clearly show the times in which each of the listed steps was performed. Without going into detail in any of the scenarios, we can observe that steps 3 and 4 have practically no effect, so it can be defined that the emulation process does not overload the system, leaving the CPU free. On the contrary, this does not happen at load moments, such as system startup or writing the CZML file, where the CPU resources used are extremely high. On the other hand, the use of RAM memory is affected by the use of VMs and the creation of the server where Cesium is running. As expected, once the process is finished, you can verify that all the parameters return to their initial state.

If we go into detail in the different scenarios, we can obtain very interesting data. First, if we start by analyzing the graph of Figure 6.5, which corresponds to the RAM in a case with 3 light VMs. We observe that the free RAM and the used RAM change their value in a correlated way as VMs are created. This does not occur at the time of server generation, where more space is reserved than later used.

If we continue in the same scenario, but we analyze the CPU, we can see the abovementioned effects of load, although in this case due to the reduced size of the scenario we have not captured a maximum during the creation of the file for Cesium, as shown in Figure 6.6. On the other hand it is interesting to note the performance of the CPU load with respect to the use of VMs, in this case unlike the RAM does not use constant resources instead it is releasing them as it uses them.

Another important performance factor is the execution time. Comparing the two graphs, we can determine on a scale of seconds the execution time of each process. If analyzed chronologically, the VSNeS startup process is in first place, taking less than 3 seconds. Then 3 VMs are created and started up, which takes approximately 45 seconds, about 15

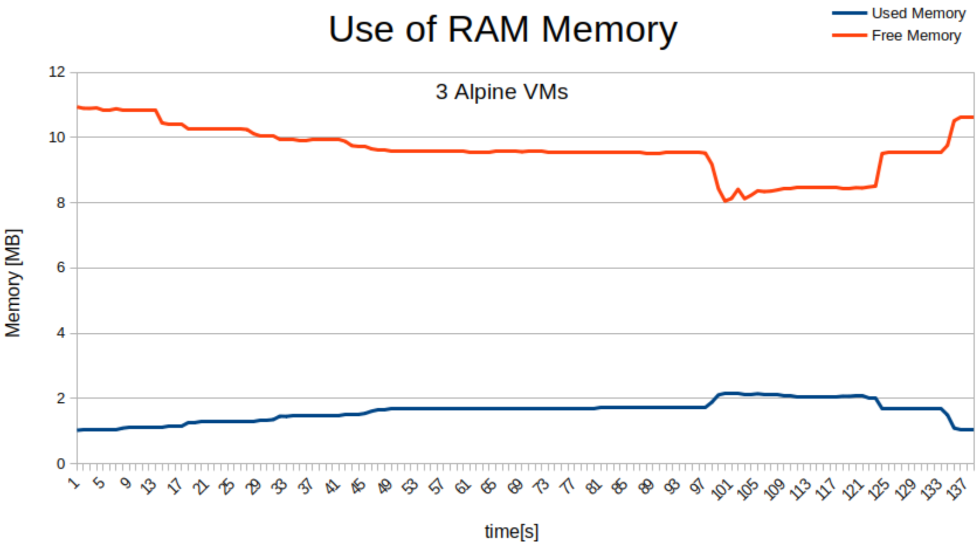


Figure 6.5: Performance test: CPU resources consumption graph during the execution of 3 Alpine virtual machines with VSNeS.

seconds per machine. Finally, the other relevant data, such as the data formation for the GUI, is not clear enough, due to the measurement precision, to draw conclusions.

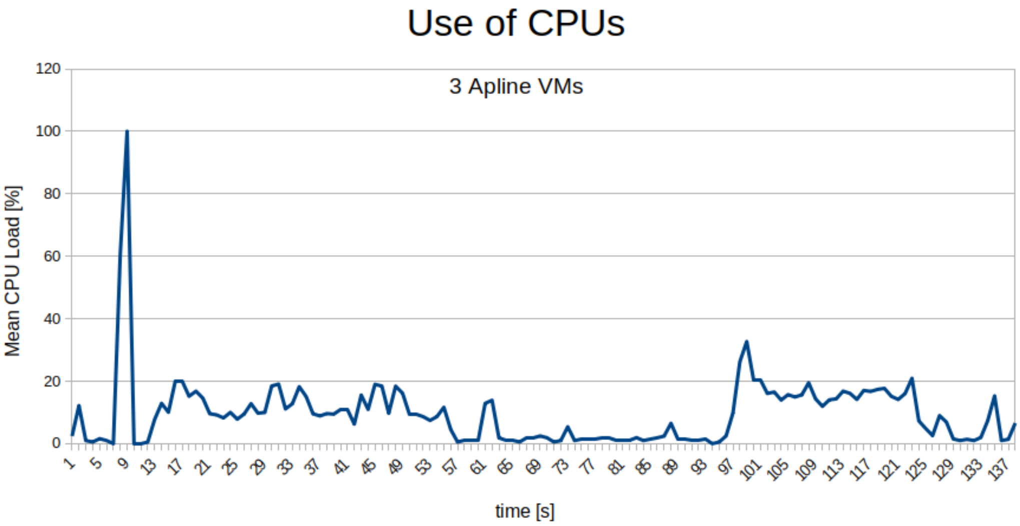


Figure 6.6: Performance test: RAM memory consumption graph during the execution of 33 lpine virtual machines with VSNeS.

Continuing with the same 3-node configuration as before, we now analyze the effect of heavier VMs. We start by studying the RAM, that is showed in Figure 6.7. As expected, the resources consumed by these machines were going to be much higher, this is observed in the growth of the memory used as each one is created. In the case of Figure 6.5, the steps were soon appreciable. But the most differential factor is found in the reservation of RAM memory, eliminating all free RAM. Despite this reservation of resources, when the host requires these resources for the formation of the server, they are released.

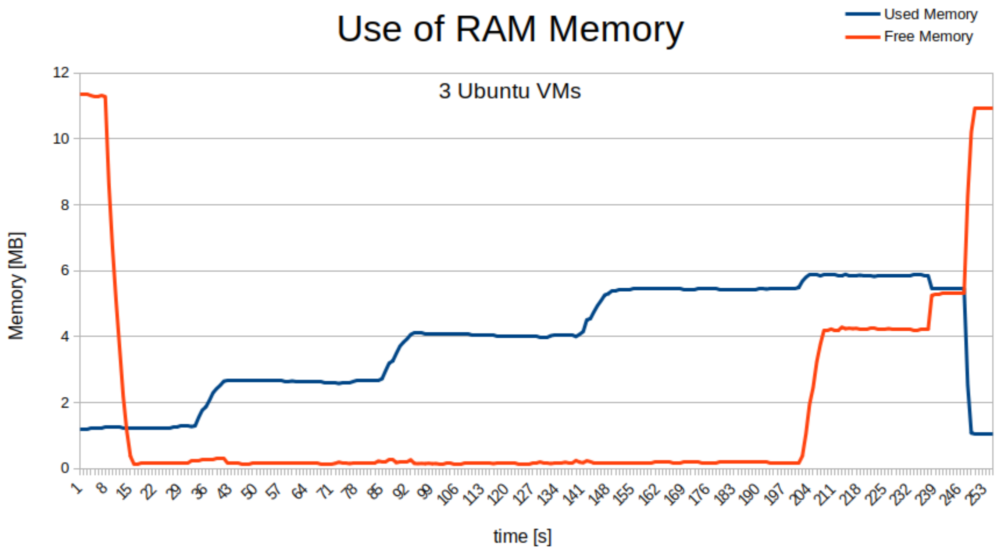


Figure 6.7: Performance test: RAM memory consumption graph during the execution of 3 Ubuntu virtual machines with VSNeS.

Another notable difference between light and heavy machines is the use of the host CPU, going from a 20% maximum during the creation of the machines to maximum values of 70%. In Figure 6.8, in addition to the comparison of scales, it can be seen in detail how the time of creation and startup of the machine are critical, and then the resources are released.

In this second scenario, time becomes a more decisive element, as step 2 is greatly increased. Approximately, the time of this process takes about 140 seconds, i.e. a little less than two and a half minutes. This value is three times that obtained with the Alpine machines. Finally, it is interesting to note that this graph does capture the maximum during the loading of the CZML file, which is not an excessive amount of time.

Having studied the simplest cases, it is time to analyze how VSNeS behaves when the host machine is pushed to the limit. If we start with the Alpine machines, we observe that the use of 33 VMs does not collapse the system, despite this being the limit due to the available storage. In Figure 6.9, we can see how the launching of 33 nodes uses about 6 MB of RAM and leaves 1.3 GiB free. This is a clear indication of the possibility of launching much larger scenarios, especially if we do not focus on the free space, but on the maximum capacity of the laptop, which is 15.5 GiB. These resources will allow us to launch a scenario of up to the size of this one.

In addition to the effect of numerous machines, it is important to analyze how the complexity of the scenario increases the computation of the system. If we analyze Figure 6.10, we observe the same peaks during VSNeS startup and data collection for the viewer, which are now spread out over a longer period of time. On the other hand, the CPU load during the emulation process has been slightly elevated, but at no time is it close to high values that would imply an overload of the system.

The most significant value left by this data collection is the execution time. The most significant point is at system startup. At this moment the software stores in vectors the position of the nodes at each instant, so it is important to take into consideration that

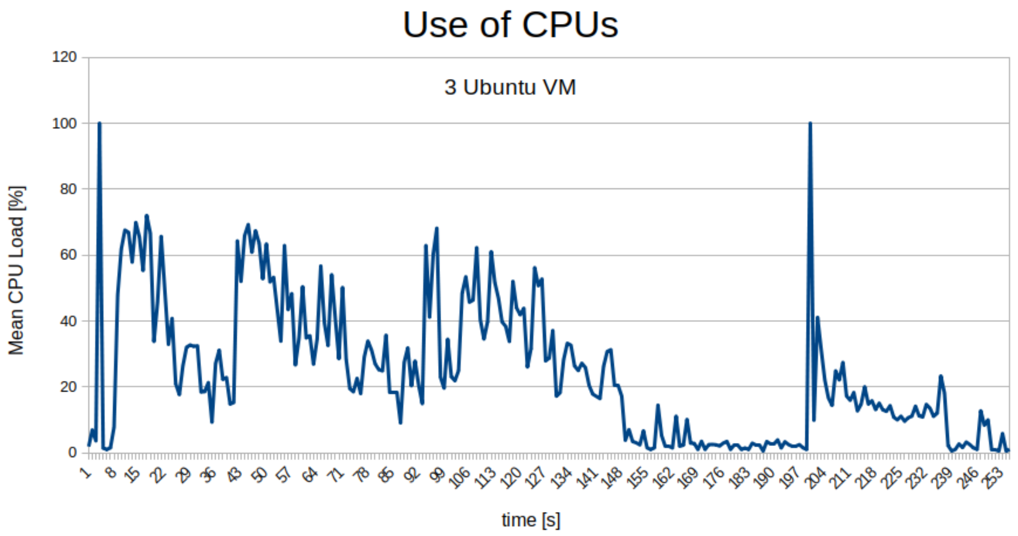


Figure 6.8: Performance test: CPU resources consumption graph during the execution of 3 Ubuntu virtual machines with VSNeS.

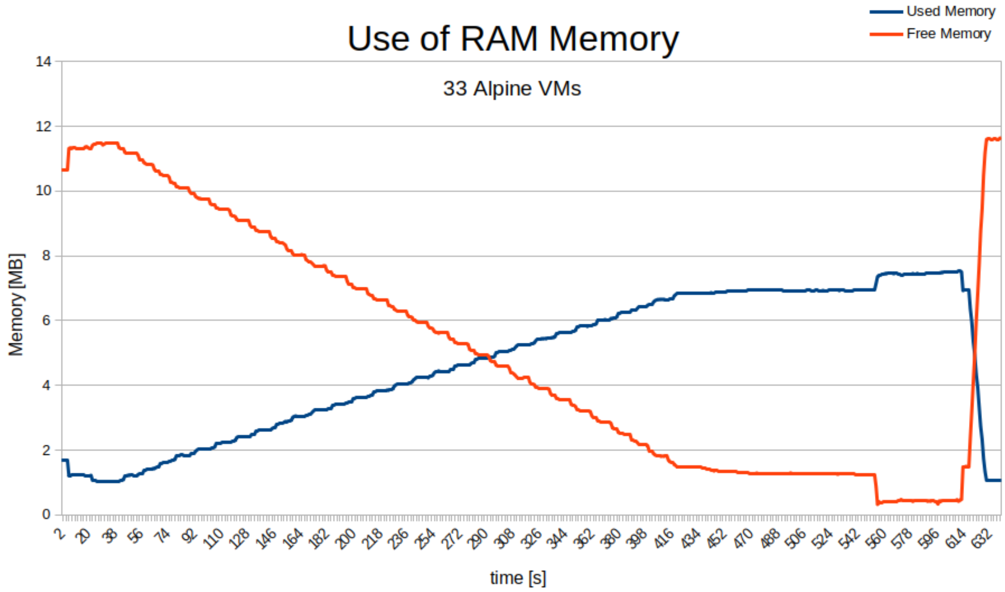


Figure 6.9: Performance test: RAM memory consumption graph during the execution of 3 Alpine virtual machines with VSNeS.

the scenario that has been loaded is formed by 240 time instants and that 31 nodes are satellites. This resulted in a start-up process of half a minute, this time is produced by the limitations of the orbit propagator that requires approximately 4 ms for the calculation of the different positions. Another time related issue is to check if the creation time of each machine is constant or increases as the number of VMs created increases. From the graphs it is obtained that the creation time of VMs is approximately 410 s which implies about 13 s to create each one, a value very similar to the one obtained in the first sampling. The rest of the data collection phases do not provide relevant information on execution times.

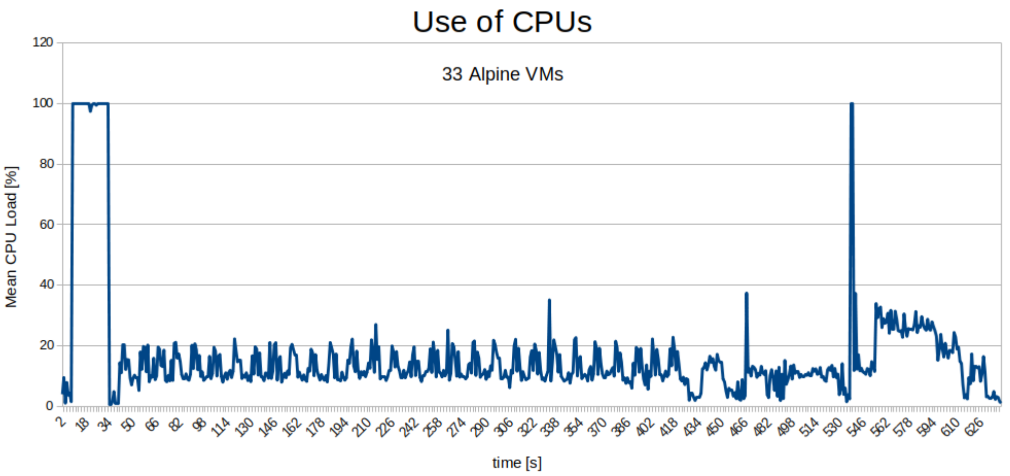


Figure 6.10: Performance test: CPU resources consumption graph during the execution of 33 Alpine virtual machines with VSNeS.

Due to the huge RAM margin found with 33 VMs, a second test was performed with 65 VMs. To avoid storage problems, the disk capacity of the new VMs was reduced. As you can see in Figure 6.11, the system works even though it is very limited, slowing down the host machine. Still, it results in a very big scenario considering that we are working with VMs.

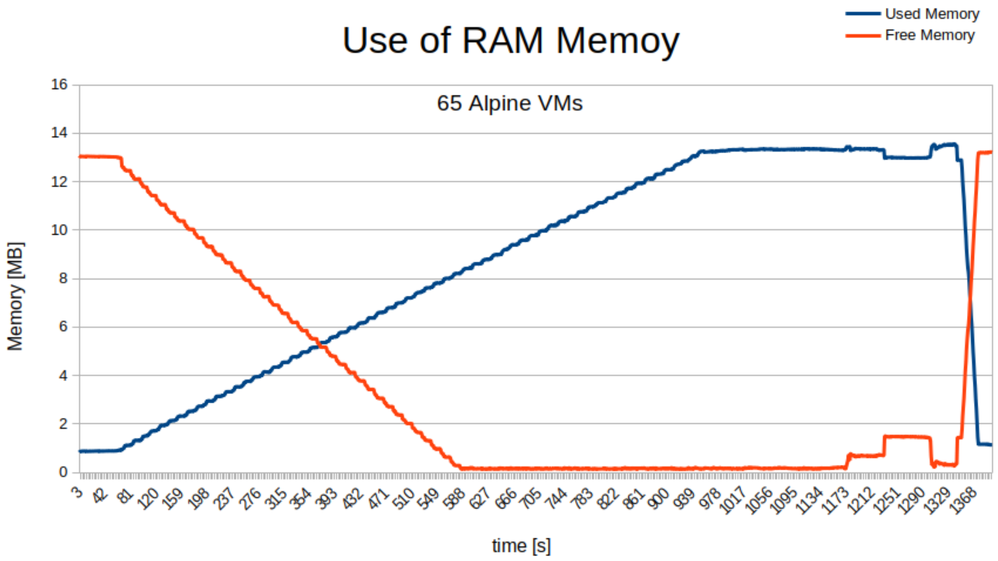


Figure 6.11: Performance test: CPU resources consumption graph during the execution of 65 Alpine virtual machines with VSNeS.

Finally, we found a case where the system collapses. With the use of the Ubuntu machines, it is not possible to launch the complete scenario that was intended. The program is able to create the 13 machines launched but not to keep all 13 running simultaneously, being limited to 8 machines running. This effect is clearly seen in Figure 6.12, with the creation of the first 8 machines with which 100% of the RAM memory usage is reached. Once at this point, the RAM memory drops, this is the moment when the host system shuts down one of the VMs, having the capacity to create new ones. Something similar happens when starting the Cesium server, another of the machines has to be shut down to be able to execute this process. So the RAM memory in this case is a limiting factor for the creation of the scenario.

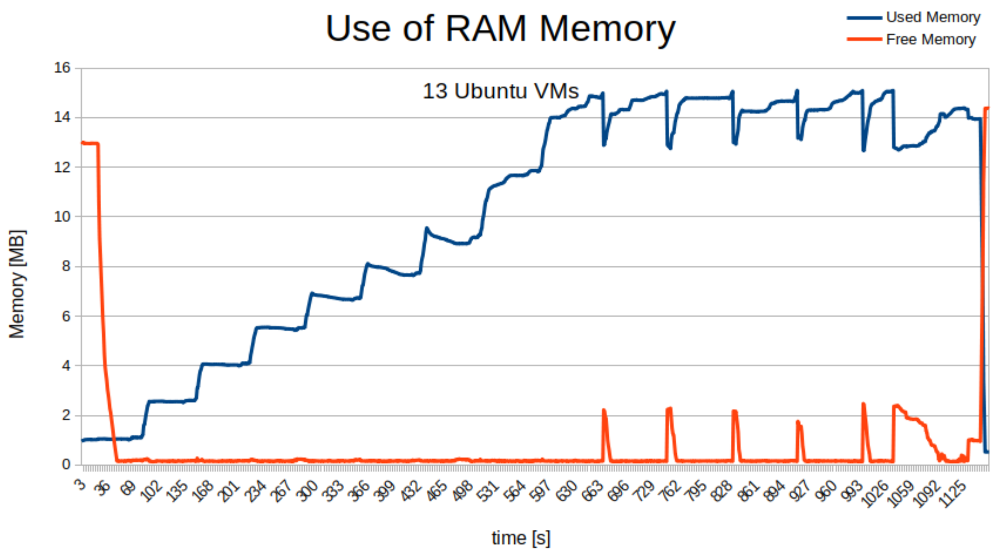


Figure 6.12: Performance test: RAM memory consumption graph during the execution of 13 Ubuntu virtual machines with VSNeS.

If we focus on the CPU, as shown in Figure 6.13. At all times, there is a high consumption, which is not surprising when compared to the first test with Ubuntu. This representation does not provide any relevant data.

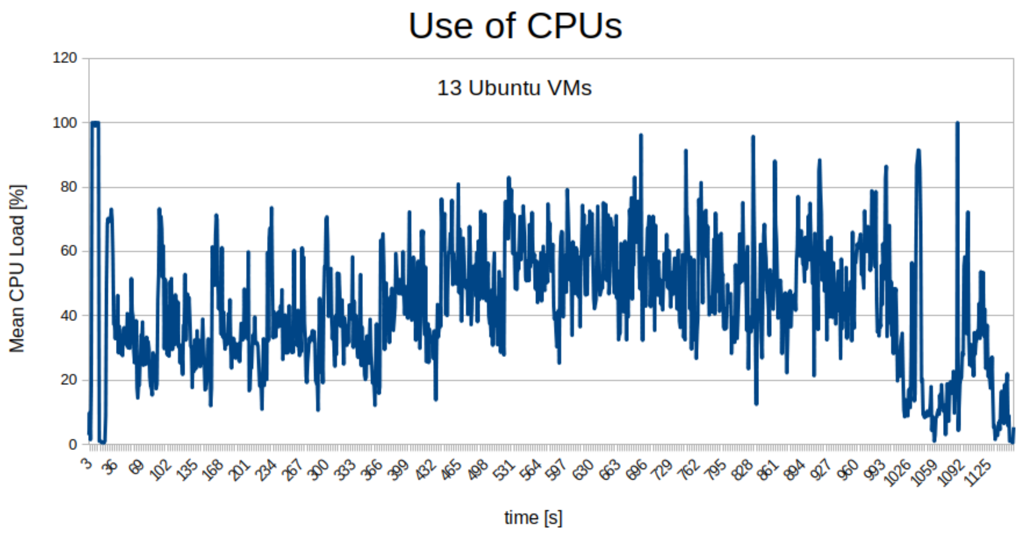


Figure 6.13: Performance test: CPU resources consumption graph during the execution of 13 Ubuntu virtual machines with VSNeS.

If we focus on the runtime, the most critical point is the VM creation, which occupies practically all the data collection. This process took about 1000 seconds, i.e. almost 17 minutes. This means an increase in the creation time of each VM with respect to the first scenario with Ubuntu machines, this is due to the limited resources and the time implemented in stopping the machines.

As it has been observed with the different measures, the program accepts many nodes as long as they do not require an excess of RAM memory. This makes it more interesting to launch machines without graphics and reducing their RAM.

# CHAPTER 7. SATELLITE SCENARIO

The following chapter groups the design and implementation of three scenarios on VSNeS. The data obtained is shown and analyzed with different protocols. These three scenarios use the relay function of the simulator on the satellites in order to interconnect two GSs. Prior to the different scenarios, the bases of these scenarios are defined and which are the different tests that have been performed on each scenario.

## 7.1. Scenario Definition

Once the VSNeS was validated, different scenarios were designed to use the software in cases closer to reality, thus demonstrating the viability of the engine for future developments. In this case, we have focused on the connection between two stations via satellite.

To achieve this connection, we have been creating scenarios with increasing level of complexity. Starting with a scenario formed by a single GEO satellite that gives a permanent view to the GS and ending with a LEO constellation in which the satellites that offer connection are varied.

The objective of these satellites is to generate a relay between the GSs, so that despite the absence of direct vision, they can communicate with each other. In these scenarios, latency, packet delivery rate and throughput parameters have been obtained. In each scenario, mesurements have been made with different losses in order to check how the different protocols interact. The tools used for the mesurements were ping and iperf3.

The ping tool uses Internet Control Message Protocol (ICMP) to send a request and waits listening for the response. With the received packets, the host can determine the RTT and the packet losses. In the follow scenarios, we use the tool in one of the GS with the follow command line:

```
ping -c [N_pings] [IP_dst] | sed 's/ /,/g;s/=/,/g'>[NameFile.csv]
```

The command line writes a file in csv format of a specific number of pings to the indicated IP address. The sed tool changes characters to other characters, in this case we change the spaces and the equals sign to commas. This is done to make easy work with the data in an Excel spreadsheet. This command uses the default sending period, which is equivalent to one packet per second.

Iperf3 is a tool to generate data flows with a certain bitrate that are mesured to obtain the network throughput rates. It tests TCP, UDP, and SCTP throughput. To do it, it needs to establish a server and a client. In the scenarios, TCP and UDP measurements have been performed using the following commands:

```
#Client TCP
iperf3 -c [IP_dst] -p [Nport] -t [time] > [NameFile.csv]
#Client UDP
iperf3 -c [IP_dst] -p [Nport] -u -t [time] > [NameFile.csv]
#Server
iperf3 -s -p [Nport]> [NameFile.csv]
```

The metrics that we can obtain from the mesurement with iperf3 in TCP mode are formed by the amount of data sent and the bit rate. On the other hand, in UDP mode we obtain data transferred, bit rate, loss probability and jitter. In this case, jitter is defined as the variation of one-way delay in a packet flow, ignoring lost packets. We work with different ports for each of the protocols in order to be able to collect samples from both simultaneously. This is done to reduce data collection time. Due to the commingling of flows during data collection, the data may be altered as the full capacity of the channel is not reserved for the test of each protocol. However, it was decided to perform the tests in this way as it was possible to obtain sufficient information to understand the interaction of the protocols and to reduce the test time to one third. To better understand how the different protocols interact with the flow junction it is important to consider the configuration of the protocols. Iperf3 by default transmits with an unlimited bitrate for TCP, while for UDP it uses 128 KBytes/s by default.

All commands have been executed in the invited VMs. To facilitate the data processing, the different .csv files have been transmitted to the host. For this purpose, the scp tool has been used. The following shows how it is used.

```
scp [NameFile.csv] ubuntu@192.168.122.1:/home/ubuntu/
```

The command sends the specified file to a specific address on another machine. In this case, it is displayed with the username and the IP address of the computer used for the different scenarios.

In the three scenarios presented, the above measurements have been performed under three different situations. Measurements were made by modifying the packet losses of each link with VSNeS (0%, 1%, and 10%). It is important to consider that the configured losses are for each link, so the packet loss is cumulative, this means that the losses of the end-to-end link are higher than those configured by having an intermediate node (the satellite in relay mode). The probability of system losses follows the relation of the 7.1 equation, where $P_C$ is equals the probability of receiving the message correctly in the total system, which is defined in Equation 7.2. $P_{L1}$ is the probability of losses in a link and $n$ is the number of links.

$$P_L = 1 - P_C \tag{7.1}$$

$$P_C = (1 - P_{L1})^n \tag{7.2}$$

During the scenarios, packet loss measurements are performed at two different points, at the receiver and at the transmitter, which receives a response to its transmission. In the case of the receiver, when taking measurements with UDP mode, we obtain losses caused by two links. This causes the established loss values to increase according to the 7.3 equation, resulting in theoretical values of 0%, 2%, and 19%. In the case of measurements at the transmitter, i.e. using the ping tool, the reflected losses will coincide with those of 4 links, following the 7.4 equation. The theoretical values for this test increase to 0%, 3.9%, and 34.4%. In the case of TCP transmission, since the protocol is able to detect them and perform retransmissions, the measurements are not reflected in the loss rate, rather in the effective throughput, which is reduced.

$$P_L(n=2) = 2 * P_{L1} - P_{L1}^2 \tag{7.3}$$

$$P_L(n=4) = 4 * P_{L1} - 6 * P_{L1}^2 + 4 * P_{L1}^3 - P_{L1}^4 \tag{7.4}$$

As the data collection is done simultaneously, a limitation of the channel transmission has been set well above the one expected to be achieved. This allows us to see how the different protocols work in terms of latency and packet loss.

## 7.2. Scenario 1: GEO Satellite

The first scenario has been established with a satellite of the INTELSAT company. The INTELSAT network consists of geostationary satellites that provide global coverage. The Intelsat New Dawn satellite (NORAD ID 37392), also known as Intelsat 28, has been selected. The satellite is selected because it is located in a stationary orbit at 33 degrees east over the equator, this allows the existence of LoS with the i2Cat GS (which has been used throughout the tests) giving it a long transmission range. The real function of the SAT is to provide television and broadband service over Africa, but in this case we will only take its data as a reference of its orbit.

The satellite will provide a relay service between a GS located in Spain and another one in India, as shown in Figure 7.1 and Figure 7.2. These nodes are located one on the i2CAT's roof (Spain) at 41 ° 23' 15.2" N, 2 ° 06' 43.1" E, and 150 m of height and the other is located in New Delhi (India) at 28 ° 37' 13.6" N, 77 ° 12' 27.4" E, and 150 m of height.

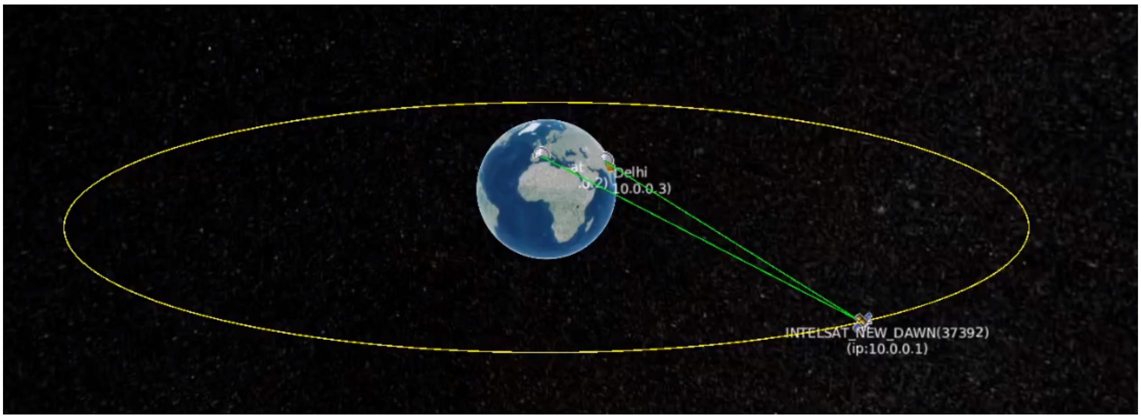


Figure 7.1: GEO Scenario: Representation in Cesium of the relay between i2Cat and New Delhi.

Channel availability between nodes has been defined with a fixed value of distance threshold. The height of a GEO satellite is around the 35,000 km over sea level and the position of the GSs are not in the equator on in the longitude 33 E, meaning that the minimum threshold to establish communications is much higher than the satellite's height, 40,000 km threshold has been defined in order to have enough margin to secure the connection. We ran the TCP and UDP tests for 13 min and launched 850 pings. We obtained RTT, jitter, packet error and throughput data.

If we begin by analyzing the data obtained with the TCP test, we can clearly see how the throughput varies as a function of the losses. The operation of TCP is based on the confirmation of received messages with an ACK. This causes the packets to be resent in case of not receiving the confirmation. In addition, the delay of the scenario causes a

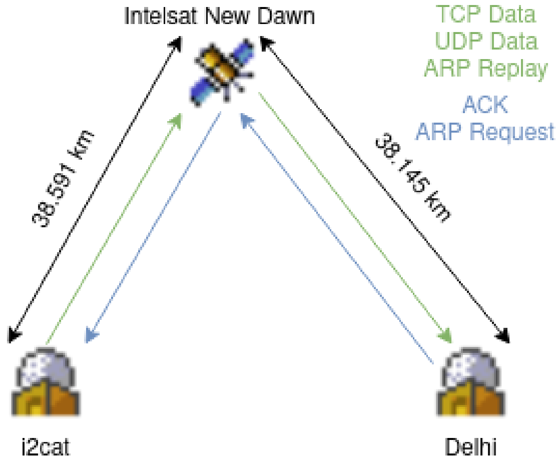


Figure 7.2: GEO Scenario: Scheme of the relay between i2Cat and New Delhi.

limitation in the sending of new packets when waiting for the ACK of the previous ones. The obtained metrics without losses correspond to a total throughput of 2.1077 Mbytes/s. With only 1% loss the transmitted data rate is reduced to 42.2 Kbytes/s. In the worst case, 10% loss, throughput is 1.8 Kbytes/sec. This simple test provides a clear reflection of the limitations of this protocol in the face of high delays and losses.

In the case of UDP, a protocol in which no data is resent and no acknowledgment of receipt is made, the test offers a good tool for estimating losses. With the transmission of datagrams, we have obtained results of losses that agree with the theoretical ones (0%, 1.99%, and 19.14%). The transmission rate that the receiver perceives is directly related to the losses. In the best case, the throughput is 128 kbytes/s and is reduced depending on the losses to 125.4 kbytes/s, and 103.5 kbytes/s (1% and 10% respectively). In this case we cannot draw clear conclusions about the protocol, but we can verify that the software and the scenario work as expected. Table 7.1 lists all the data for Scenario 1: throughput with TCP and UDP, Packet Loss Ratio (PLR) in UDP test and Packet Delivery Ratio (PDR) in UDP too.

| GEO Scenario | | | | |
|---|---|---|---|---|
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 2,158.285 | 128.0398 | 00.00 | 100.0 |
| **1%** | 42.199 | 125.3779 | 01.99 | 98.01 |
| **10%** | 1.796 | 103.477 | 19.10 | 80.90 |

Table 7.1: Transmission parameters in Scenario 1.

Finally, we conducted a study of RTT and losses during pings. The values obtained correspond to 513 ms of RTT, which is consistent with the established connections, where the nodes are located at a distance greater than 38 000 km, as we can see with the latency of the links in Figure 5.8 (38 591 km with i2Cat and 38 145 km with Delhi) . Figure 7.3 shows a box plot with the reception of the different packets in the 3 tests, from which the jitter can also be estimated with the standard deviation. In this case the packet loss values do not coincide in their totality, even though the values are close and with longer test duration would tend towards them. The theoretical values correspond to 0 %, 3.9 %, and 34.4 % while those obtained are 0 %, 3.6 %, and 36.7 %.

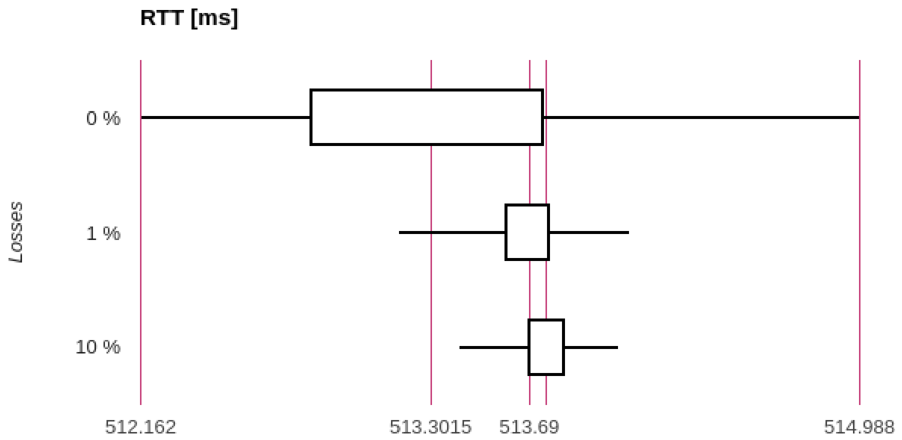


Figure 7.3: Box Plot: RTT of 850 pings between i2Cat and New Delhi (ICMP).

## 7.3. Scenario 2: LEO Satellite

The LEO orbiting satellite scenario has followed the same principles as in GEO and consists on two GSs with simultaneous contact with the satellite that relays the packets. For this purpose, the ground segment had to be relocated. It has been decided to keep the location of Spain, the i2CAT office and add a new station in Italy. This new GS is located in Milan at 45° 28' 10.89" N, 9° 10' 42.81" E, and 400 m of height.

For the selection of the satellite, a survey was performed on August 14, 2022, at 10 a.m. of the position of the Starlink constellation satellites. Starlink [43] is a macro constellation developed by Spacex that offers internet service via satellite. During this study, we searched for a satellite with a ground path that overflies both locations. The satellite that best matched our requirements at that time was STARLINK-1104 (NORAD 44922). Figure 7.4 and Figure 7.5 show the orbit of the selected satellite and the connection that it establishes with the GSs.

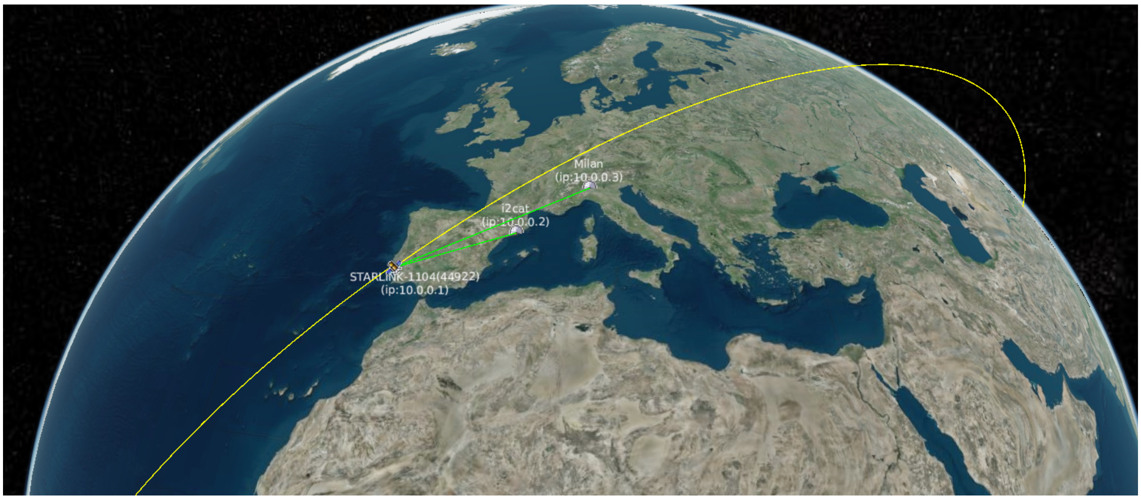


Figure 7.4: LEO Scenario: Representation in Cesium of the relay between i2Cat and Milan.

In this scenario, very similar parameters have been used for the channel definition: same maximum throughput and same probability of packet loss. However, the threshold distance has been reduced to 2 000 km. This is because the distance between nodes is much smaller, so the power capabilities of the satellites are reduced. This value has been chosen arbitrarily, taking into account the range of altitudes that comprise the LEO satellite region.

During the course of a day, a LEO satellite can provide coverage to an area of the Earth multiple times, but the contacts are not characterized in the same pattern. For this reason, we have studied the contacts that can occur during a day and the different effects. Table 7.2 shows that during the selected day, the satellite establishes a connection during 6 times. This configuration allows a contact time between the two GSs in the range of 4.5 and 8 minutes.

| **First Contact** | | | |
|---|---|---|---|
| **Connection** | **Start of contact** | **End of contact** | **Contact time [min]** |
| i2CAT to STARLINK-1104 | 08:32:00 | 08:37:30 | 5.5 |
| Milan to STARLINK-1104 | 08:33:00 | 08:39:30 | 6.5 |
| i2CAT to Milan | 08:33:00 | 08:37:30 | **4.5** |

| **Second Contact** | | | |
|---|---|---|---|
| **Connection** | **Start of contact** | **End of contact** | **Contact time [min]** |
| i2CAT to STARLINK-1104 | 10:09:00 | 10:18:00 | 9 |
| Milan to STARLINK-1104 | 10:10:30 | 10:20:00 | 9.5 |
| i2CAT to Milan | 10:10:30 | 10:18:00 | **7.5** |

| **Third Contact** | | | |
|---|---|---|---|
| **Connection** | **Start of contact** | **End of contact** | **Contact time [min]** |
| i2CAT to STARLINK-1104 | 11:49:30 | 11:57:30 | 8 |
| Milan to STARLINK-1104 | 11:51:00 | 11:59:30 | 8.5 |
| i2CAT to Milan | 11:51:00 | 11:57:30 | **6.5** |

| **Fourth Contact** | | | |
|---|---|---|---|
| **Connection** | **Start of contact** | **End of contact** | **Contact time [min]** |
| i2CAT to STARLINK-1104 | 13:31:00 | 13:37:30 | 6.5 |
| Milan to STARLINK-1104 | 13:31:30 | 13:39:30 | 8 |
| i2CAT to Milan | 13:31:30 | 13:37:30 | **6** |

| **Fifth Contact** | | | |
|---|---|---|---|
| **Connection** | **Start of contact** | **End of contact** | **Contact time [min]** |
| i2CAT to STARLINK-1104 | 15:10:30 | 15:19:00 | 8.5 |
| Milan to STARLINK-1104 | 15:11:00 | 15:20:00 | 9 |
| i2CAT to Milan | 15:11:00 | 15:19:00 | **8** |

| **Sixth Contact** | | | |
|---|---|---|---|
| **Connection** | **Start of contact** | **End of contact** | **Contact time [min]** |
| i2CAT to STARLINK-1104 | 16:50:00 | 16:59:00 | 9 |
| Milan to STARLINK-1104 | 16:51:00 | 16:59:00 | 8 |
| i2CAT to Milan | 16:51:00 | 16:59:00 | **8** |

Table 7.2: Contact times in Scenario 2.

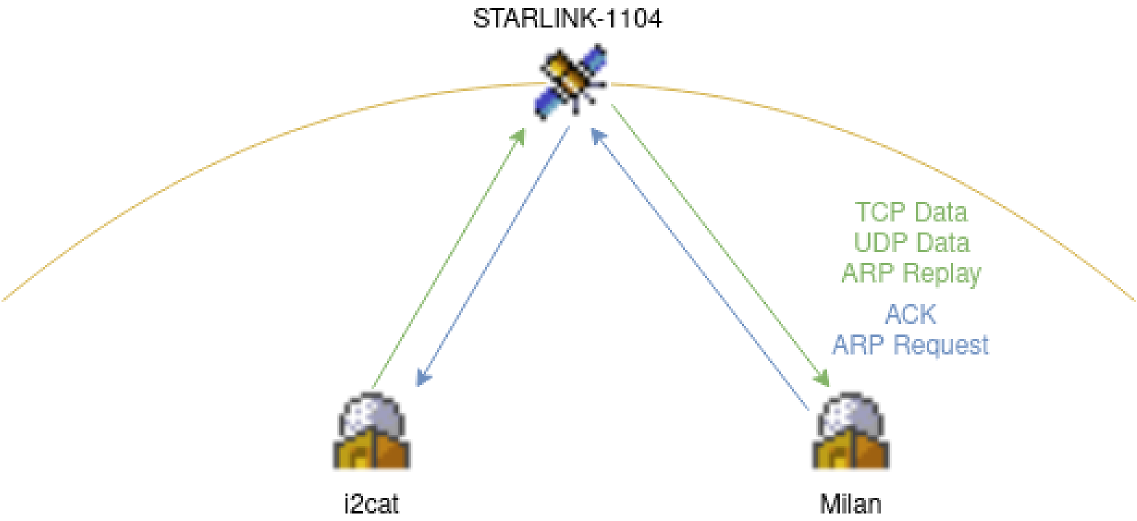


Figure 7.5: LEO Scenario: Scheme of the relay between i2Cat and Milan.

As can be seen, in this scenario the most limiting concept is the contact time, forcing all the information to be sent at a precise instant. If we follow the same analysis pattern as in the previous scenario, we observe a clear improvement in the amount of data sent via TCP. In the first test, without losses, we obtain mean throughput of 126 Mbytes/s. As in the first scenario, TCP drastically reduces the data throughput by adding losses. The metrics obtained with 1% loss correspond to a mean transmission rate of 651.6 Kbytes/s. The most radical change is found in the last scenario, where the mean throughput is reduced to 7.89 Kbytes/s. As can be seen, the latency caused by LEO orbits does not have any negative effect on this protocol, but it is severely affected by high loss rates.

The analysis of the UDP transmission data does not provide any differential parameters with respect to the GEO scenario. We obtain again losses that correspond to the theoretically expected values and similar throughput as we can see in Table 7.1 and Table 7.3. Although in TCP latency has a very important influence on the effective throughput, in transmissions over as UDP that do not require confirmations, latency only generates a delay in the signal, but it has no effect on throughput.

Finally, we have collected data from the use of ping. The test was performed during the tested day. To reduce the execution time, a multiplier of x300 over the execution time has been configured for non-contact moments, which means that the time in the emulation is accelerated during the moments when communication is not possible. With this data collection, we can compare how the RTT varies in the different contacts. As we can see in Figure 7.6, the RTT fluctuates between 27 and 10 ms, always following a very similar pattern, the RTT variation during each contact can be analyzed as a parabola. It starts the contact at the instant with the longest delay and as the transmission takes place it reduces and then increases. This is provoked by the distance of the satellite between the GSs at each instant. Red circles indicate the moments in which there are gaps between the samples, caused by the moments in which there are no contacts. Although apparently the parabolas are united, this is not the case because of these gaps. We have also analyzed the packet losses of some of the contacts, obtaining data that corresponds to the expected values, conforming to the theoretically calculated values (values obtained in the second contact: 0 %, 3.9 %, and 40 %).

| **LEO Scenario** | | | | |
|---|---|---|---|---|
| **First Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 110,606.336 | 127.640 | 00.00 | 100.0 |
| bf 1% | 499.712 | 125.360 | 01.97 | 98.03 |
| **10%** | 10.757 | 103.455 | 19.10 | 80.90 |
| **Second Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 132,894.720 | 127.890 | 00.00 | 100.0 |
| **1%** | 808.960 | 125.230 | 01.96 | 98.04 |
| **10%** | 10.650 | 103.290 | 19.20 | 80.80 |
| **Third Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 135760.896 | 127.924 | 00.00 | 100.0 |
| **1%** | 631.398 | 125.215 | 01.95 | 98.05 |
| **10%** | 9.400 | 103.270 | 19.20 | 80.80 |
| **Fourth Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 123,012.096 | 127.980 | 00.00 | 100.0 |
| **1%** | 580.608 | 125.093 | 01.95 | 98.05 |
| **10%** | 4.962 | 103.887 | 18.70 | 81.30 |
| **Fifth Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 136,775.680 | 127.899 | 00.00 | 100.0 |
| **1%** | 696.104 | 125.293 | 02.00 | 98.00 |
| **10%** | 8.676 | 103.482 | 19.00 | 81.00 |
| **Sixth Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 135,596.032 | 127.924 | 00.00 | 100.0 |
| **1%** | 692.627 | 125.377 | 01.95 | 98.05 |
| **10%** | 2.883 | 103.440 | 19.00 | 81.00 |

Table 7.3: Transmission parameters in Scenario 2.

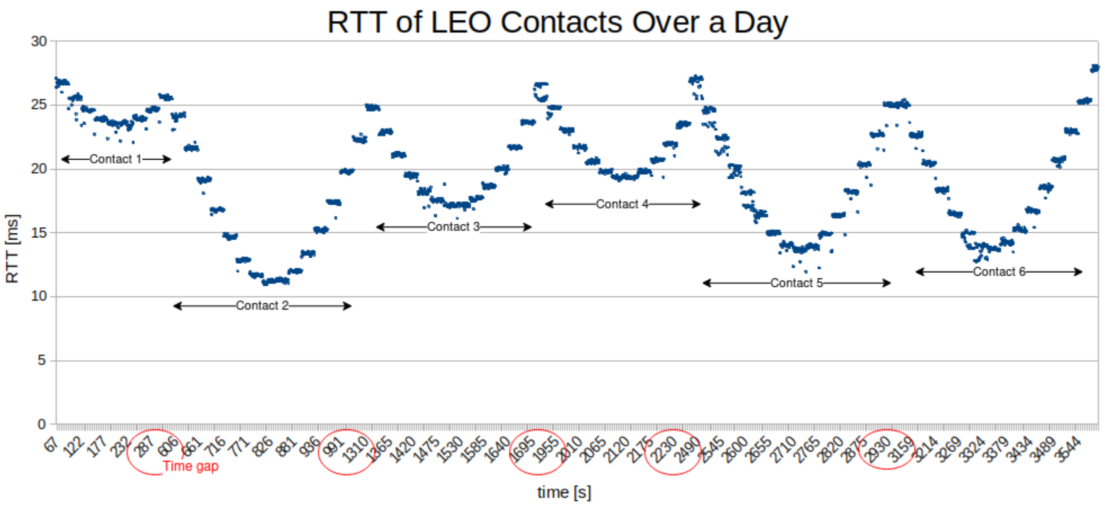


Figure 7.6: RTT of pings between i2Cat and Milan along a day around STARLINK-1104.

## 7.4. Scenario 3: LEO Constellation

Scenario 3 is an extension of the LEO satellite scenario. In this case, two satellites are added that travel over the same orbit as STARLINK-1104 (NORAD 44922). With the combination of these three satellites, we are able to extend the connection window between the GSs as well as include moments with redundancy. Redundancy occurs when more than one satellite has contact with the two stations at the same time, which doubles the traffic. The satellites added for this scenario are STARLINK-1144 (NORAD ID 44933) and STARLINK-2596 (NORAD ID 48409). Figure 7.7 and Figure 7.8 illustrate the scenario.

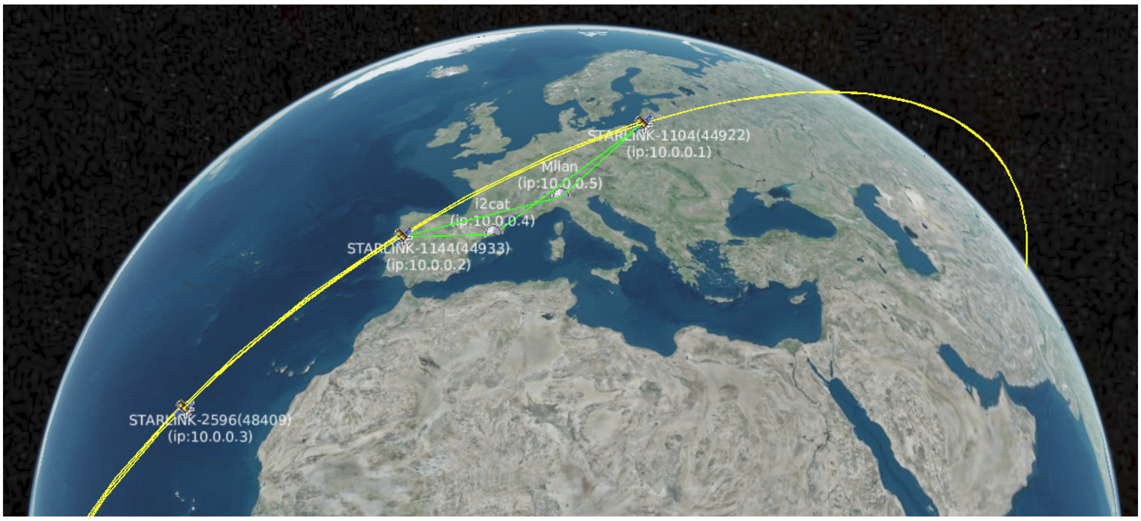


Figure 7.7: LEO Constellation Scenario: Representation in Cesium of the relay between i2Cat and Milan.

| First Contact | | | |
|---|---|---|---|
| **Satellite Relay** | **Start of contact** | **End of contact** | **Contact time [min]** |
| STARLINK-1104 | 08:33:00 | 08:37:30 | 4.5 |
| STARLINK-1144 | 08:38:30 | 08:43:00 | 4.5 |
| STARLINK-2596 | 08:43:30 | 08:49:00 | 5.5 |
| **Total** | - | - | - |

| Second Contact | | | |
|---|---|---|---|
| **Satellite Relay** | **Start of contact** | **End of contact** | **Contact time [min]** |
| STARLINK-1104 | 10:10:30 | 10:18:00 | 7.5 |
| STARLINK-1144 | 10:16:30 | 10:23:30 | 7 |
| STARLINK-2596 | 10:22:00 | 10:29:00 | 7 |
| **Total** | 10:10:30 | 10:29:00 | 18.5 |

| Third Contact | | | |
|---|---|---|---|
| **Satellite Relay** | **Start of contact** | **End of contact** | **Contact time [min]** |
| STARLINK-1104 | 11:51:00 | 11:57:30 | 6.5 |
| STARLINK-1144 | 11:56:30 | 12:03:00 | 6.5 |
| STARLINK-2596 | 12:02:30 | 12:08:30 | 6 |
| **Total** | 11:51:00 | 12:08:30 | 17.5 |

| Fourth Contact | | | |
|---|---|---|---|
| **Satellite Relay** | **Start of contact** | **End of contact** | **Contact time [min]** |
| STARLINK-1104 | 13:31:30 | 13:37:30 | 6 |
| STARLINK-1144 | 13:37:00 | 13:43:00 | 6 |
| STARLINK-2596 | 13:42:00 | 13:48:30 | 6.5 |
| **Total** | 13:31:30 | 13:48:30 | 17 |

| Fifth Contact | | | |
|---|---|---|---|
| **Satellite Relay** | **Start of contact** | **End of contact** | **Contact time [min]** |
| STARLINK-1104 | 15:11:00 | 15:19:00 | 8 |
| STARLINK-1144 | 15:16:30 | 15:24:30 | 8 |
| STARLINK-2596 | 15:22:00 | 15:30:00 | 8 |
| **Total** | 15:11:00 | 15:30:00 | 19 |

| Sixth Contact | | | |
|---|---|---|---|
| **Satellite Relay** | **Start of contact** | **End of contact** | **Contact time [min]** |
| STARLINK-1104 | 16:51:00 | 16:59:00 | 8 |
| STARLINK-1144 | 16:56:30 | 17:04:00 | 7.5 |
| STARLINK-2596 | 17:02:00 | 17:09:30 | 7.5 |
| **Total** | 16:51:00 | 17:09:30 | 18.5 |

Table 7.4: Contact times in Scenario 3.

Table 7.4 shows the 6 contacts that occurred during the day studied between the two GSs with the satellites used. The combination of these satellites on the same orbit allows us to extend contacts from about 6 minutes to contacts of up to 19 minutes, thus significantly improving the connection time. These contacts coincide in time so that the connection time is not the sum of the different contacts, rather small windows originate where the connection can be made through two satellites. These windows produce redundancies, since the messages sent by each GS arrive in duplicate to the other. However, the first

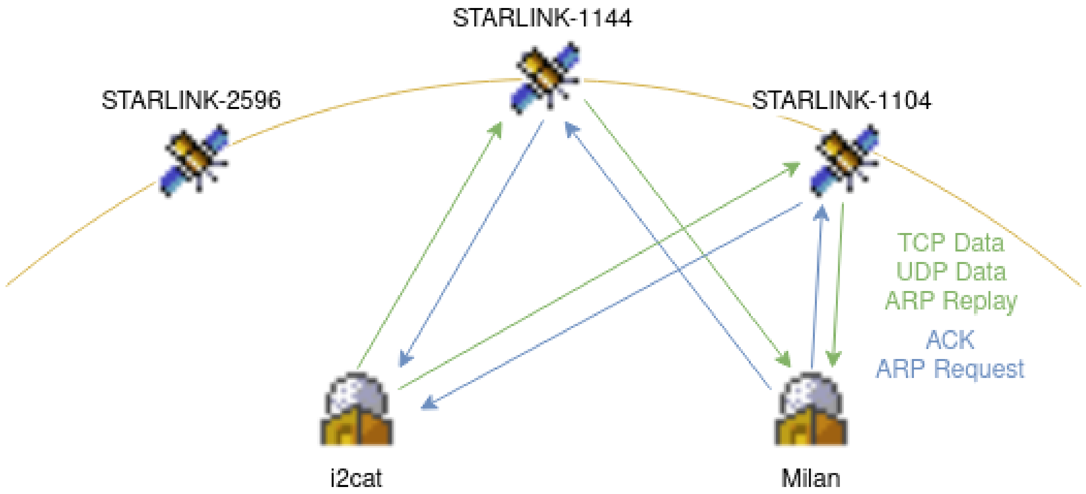


Figure 7.8: LEO Constellation Scenario: Scheme of the relay between i2Cat and Milan.

contact does not have instants with redundancy, so only individual contacts are established with the different satellites, for this reason, in Table 7.4 the information in the row of Total of the first contact is empty. This gives us a totally different case from the rest of the contacts, so the data obtained differ from the rest.

In order to further test how the emulator works, a new condition has been added, limiting the channel to a maximum throughput of 100.0 Mbytes/s. This allows to check if the channel limitations have a direct effect on the protocol, even if the limitation is not configured in the emitters.

If we analyze the scenario theoretically, we can determine the improvement of the channel when redundancy is added. By forming two paths, for an error to occur on the outgoing path, the packet will have to be lost on both paths. If we consider $P_L(n=2)$ as the losses from one GS to the other, then the losses become in $(P_L(n=2))^s$, where $s$ is the number of satellites forming the contact at that instant (0 %, 0,04 %, and 3.61 %). In the case of the probability of losses for a transmission with response, the probability of error becomes in $2*(P_L(n=2))^s$, because if an error occurs in one of the transmissions between GS will not be completed (0 %, 0,08 %, and 7.22 %). Figure 7.9 shows a schematic drawing of how the different probability calculations have been performed.

With the analysis of the characteristics obtained with TCP, as shown in Table 7.5, we observe two things. Firstly, the effect of the channel limitation, we observe that the throughput is at the limit with an average value of 93.89 Mbytes/s between contacts. Ideally this value should be 100 Mbytes/s due to the initial parameters imposed on the channel, but since the channel is shared with the flows of the other protocols being tested, it is not able to use the maximum capacity of the channel. Secondly, a clear improvement is observed in the cases where losses are added, because there are two independent paths creating the relay. This causes each packet to be duplicated and sent through each of the two paths, thus reducing the probability of error. The average throughput values obtained are 1.15 Mbytes/s and 86 Kbytes/s. If we analyze the first contact individually we can see that the results are in line with those obtained in the LEO scenario, because in this case there is no redundancy, instead they are individual contacts. It is important to note that the standard version of TCP has been used and not one specifically designed for multipath, MultiPath TCP (MPTCP) [70].

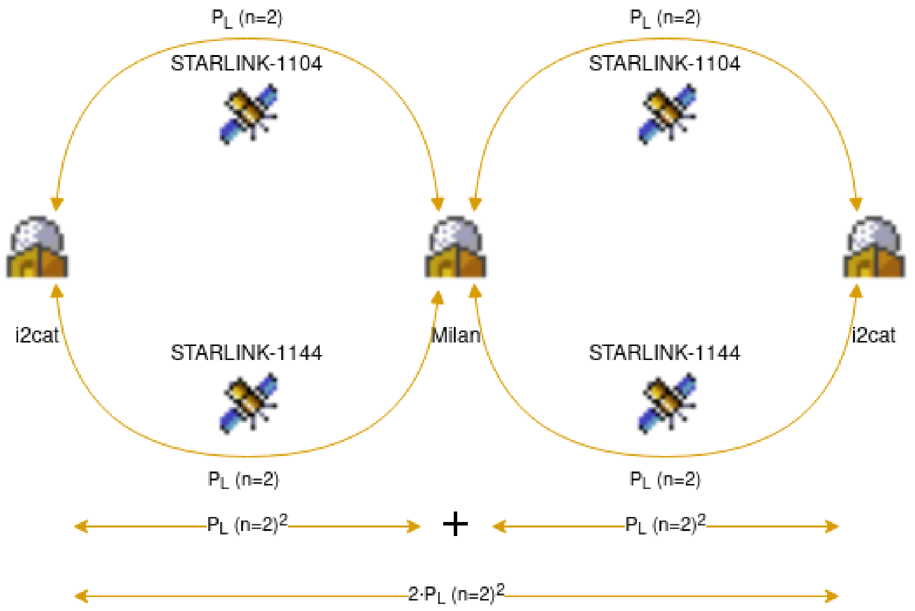


Figure 7.9: Graphical diagram of the losses produced when pinging from one GS to another with redundancy.

The data obtained with the UDP tests are of little interest, due to how duplicate data is analyzed, which causes an increase in the transmission rate that is not real. This is because duplicate data are not discarded so they are counted as additional data. In addition, in the analysis of lost datagrams, negative values appear when the packets arrive duplicated. This totally unsettles the mesurements and does not allow a meaningful analysis. In other words, the data collection is performed incorrectly, due to the way iperf3 processes the data. In the case of the first contact, where there are individual connections without packet duplication, different data appears than that presented in the previous scenario. This is caused by performing the data collection without stops, which implies that the losses that appear belong to the moments when the connection has been interrupted.

Finally, we find the analysis of the pings, with which to obtain the RTT and understand the effect that occurs when it is done simultenously with two satellites. Figure 7.10 shows the data collected throughout the simulation, in which you can see the different connections of the satellites and how packets are duplicated when there connection with two of them. In order to analyze in detail what happens, Figure 7.11 offers us a detailed view of only one of the contacts. As can be seen, the RTT values are similar to those obtained in the LEO scenario. This is within what was expected by using the same orbits and not modifying the threshold distance. In the instants where the two GSs have simultaneous vision of the same two satellites, each message that is sent by one of the GSs is duplicated. This causes the pings to arrive at three different times. These delays correspond to the fact that the request and the replay travel only through one of the two satellites, or that each one travels through a different satellite. These data can be very useful, allowing you to visualize which satellite offers the most optimal path. This could allow decisions to be made about a handover between one satellite and the other.

| **Constallation Scenario** | | | | |
|---|---|---|---|---|
| **First Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 101,435.392 | 123.881 | 09.70 | 90.30 |
| **1%** | 497.137 | 125.176 | 11.20 | 88.80 |
| **10%** | 6.547 | 120.622 | 26.70 | 73.30 |
| **Second Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 92,853.248 | 148.640 | 00.00 | 100.0 |
| **1%** | 1,515.520 | 145.754 | 00.60 | 99.40 |
| **10%** | 57.248 | 120.622 | 05.70 | 94.30 |
| **Third Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 98,355.200 | 142.600 | 00.00 | 100.0 |
| **1%** | 1,049.874 | 139.763 | 00.60 | 99.40 |
| **10%** | 73.368 | 115.464 | 09.70 | 90.30 |
| **Fourth Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 98,063.360 | 135.413 | 00.00 | 100.0 |
| **1%** | 790.840 | 132.823 | 00.60 | 99.40 |
| **10%** | 94.067 | 109.3128 | 14.50 | 85.50 |
| **Fifth Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 92,720.128 | 161.825 | 00.00 | 100.0 |
| **1%** | 1,642.496 | 158.434 | 00.60 | 99.40 |
| **10%** | 104.318 | 130.750 | 05.60 | 94.40 |
| **Sixth Contact** | | | | |
| **Test** | **TCP [Kbytes/s]** | **UDP [Kbytes/s]** | **PLR [%]** | **PDR [%]** |
| **0%** | 93,418.496 | 159.966 | 00.00 | 100.0 |
| **1%** | 1,540.096 | 156.772 | 00.60 | 99.40 |
| **10%** | 96.456 | 129.411 | 05.80 | 94.20 |

Table 7.5: Transmission parameters in Scenario 3.

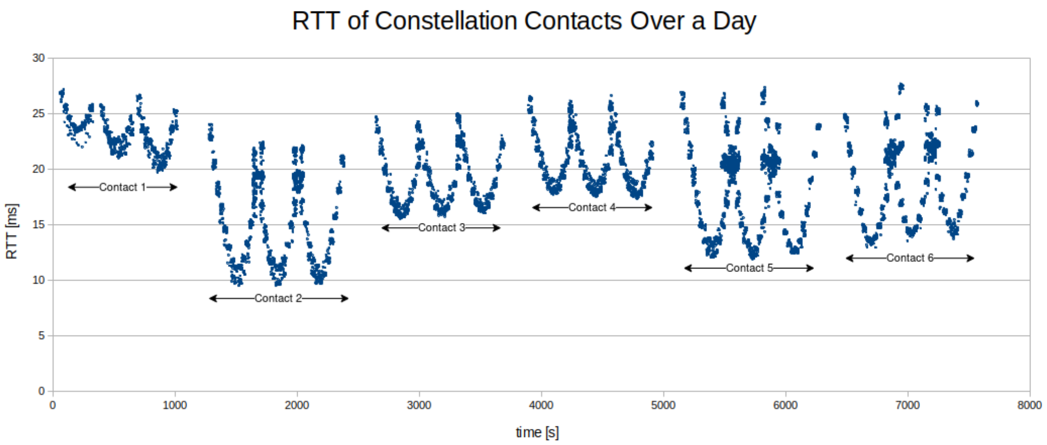


Figure 7.10: RTT of pings between i2Cat and Milan during one day of the Constellation Scenario.

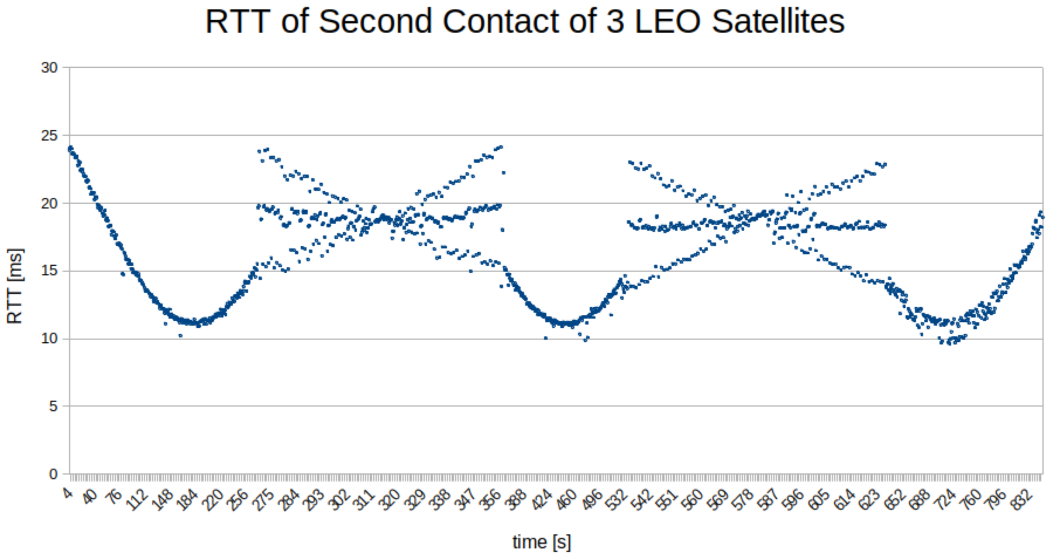


Figure 7.11: RTT of pings between i2Cat and Milan during of the second contact of the Constellation Scenario.

## 7.5. Discussion of the Results

Once all the collected data has been gathered, we compare and understand how each of the protocols works in the different scenarios. The first scenario, formed by a GEO satellite, offers a sustained coverage characterized by a very high latency. The second, based on a LEO satellite, offers discontinuous connections but with very low latency. Finally, the third scenario is composed by the constellation (three satellites located on the same orbit), these offer similar characteristics to those of Scenario 2 but increase the contact times and add redundancy in some moments.

The above-mentioned characteristics can be seen reflected in the data collected during the ICMP tests, where delays are observed practically constant with the GEO satellite but with values of around 513 ms. This is caused by the static position with respect to the Earth's surface that these satellites have and their altitude of 35 786 km, which implies a theoretical propagation time of at least 119 ms from a GS to the satellites. In Figure 5.8 on page 63 we can see a standard value of latency from the links of Scenario 1. This time must be multiplied by 4, which is the number of times the packets travel between the satellite and the GSs. On the other hand, the pings performed on the LEO satellite provide a discontinuous connection with variable delays over time, which also vary from one contact to another. This is caused by the characteristic ground track of these satellites, which changes in each period, as shown in Figure 7.12. The data from the last scenario reflect a clear increase in the number of pings received (hence in the connection time) and also allow visualizing how the packets are doubled at the time of simultaneous contacts.

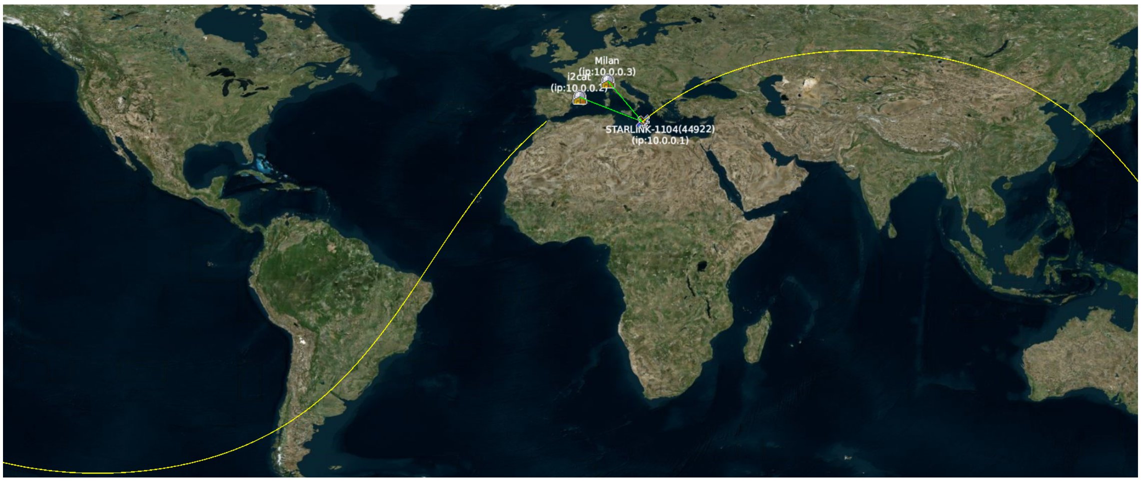


Figure 7.12: Ground track of a period of STARLINK-1104, in VSNeS GUI.

In the case of TCP, there are differences between the three scenarios. This is due to the operation of sliding window and ACKs for flow control [71]. This protocol sends a burst of packets and does not continue forwarding subsequent packets until it receives confirmation that the packet has arrived correctly and if this ACK is not received or is received too late (later than a certain threshold), it re-sends the information. Thus causes that in situations with high delays the data flow is not constant, as it goes a long time without sending new information while waiting for the confirmation of the previous packets, as shown in Scenario 1. In this case, a high level of losses also has a great effect, as there is a possibility of losing both the message containing the data and the confirmation message. This forces many of the messages to be re-transmitted, even if they have arrived correctly. For this reason, Scenario 3 offers better results, this is achieved by reducing the chances of packet loss in the moments with connection by 2 satellites. The poor metrics that TCP shows in the face of long delays is not a new or unknown concept, but has already been worked on in versions to solve this problem, in the article [72] are shown solutions such as modifying the sliding window, the Timestamp or applying Selective ACKnowledgement (SACK). We can also complement these concepts with TCP Extensions for Long-Delay Paths [73].

Finally, we find the UDP data. These are very similar to the first two scenarios. This is because the latency has no effect on the throughput, just the instant the datagrams are received. This is because it has no control method to resend data that does not arrive correctly, as it is used in applications where is not necessary receive all packets correctly, such as live-streaming. The data in the last scenario is altered by the instants when there is redundancy. Unlike TCP, UDP has no control over which packets arrive for later reordering. Therefore, all received packets are considered as relevant data, despite the fact that many of them are redundant. This ends in an erroneous data collection where a higher data reception is achieved than the ones sent.

# CONCLUSIONS AND FUTURE LINES OF WORK

## Conclusions

Space has been populated by a wide range of satellite systems from governmental and private space entities. Monolithic satellites have been ruling the space by providing a custom design that accomplishes a specific mission. However, novel user demands emerged requiring global coverage, low revisit time, and ubiquitous service. The possibility to integrate in-orbit infrastructure to support current mobile system is being discussed persistently in the last years. This motivated the research on the satellite emulation concept in order to optimize novel functionalities to in-orbit infrastructure and allocate properly resources for each service.

This work has presented VSNeS, a novel simulation engine capable to represent satellites and ground nodes in VMs and deploy a virtual network that represents the channel effects and dynamics. The concept, requirements and design of the software have been presented in details, remarking each component and their functionalities. Additionally, results of the performance evaluation have been discussed, demonstrating the viability of the system to deploy large constellations. Finally, a realistic scenario has been deployed with VSNeS.

In Chapter 3, Table 3.1 defines ten requirements that VSNeS must satisfy were established. The validation of these requirements is a first indication of whether the project has been carried out as expected. As we will now see, all the initial requirements have been met.

Firstly, VSNeS required to be able to have different types of nodes: ST, GW, and SAT. VSNeS represents and differentiates the nodes, with the small difference that finally only two types GS and SAT have been considered. This is because the difference between ST and GW is a configuration concept but for the system they work in the same way, being fixed nodes on the surface of the Earth.

The second requirement was the deployment of the nodes with VM, which has been more than fulfilled. This and the configuration of two networks to connect the VMs, has allowed the third requirement to be met.

R3 consisted in the possibility of being able to communicate through the emulator from an external machine. This has not been tested, but it would be totally possible with a simple configuration.

The fourth point was directly related to the channel characteristics. Initially it was considered only to implement delay between the machines. Finally, other channel characteristics such as transmission speed and packet loss have been added, although these are not dependent on the position of the nodes but are fixed.

R5 consisted of all concepts related to orbit simulation. For the emulator there was the possibility to replicate representative delays without generating real orbits or the representation and extraction of orbit data. The objective was to be able to represent real scenarios, which has been possible.

The sixth requirement was related to the protocols that can be implemented. In theory, the system accepts any protocol that can work on an Ethernet network. Only ICMP, TCP, and UDP have been validated because the implementation of a protocol was far from the objectives of this thesis.

The seventh requirement was the possibility of measurements at various points, being able to analyze the packets sent with those received between any point of the scenario. VSNeS does not offer a measurement tool, but it does allow measurement with other programs such as tcpdump or Wireshark.

Requirements eight and nine, configuration files and automatic deployment, have been successfully completed. The system startup is always produced by the information in a TOML file. With this information, the system is able to create VMs and configure them to work on the system.

Finally, the last requirement was the formation of a GUI to display the virtualized scenario, which has been achieved by using Cesium.

The validation of these requirements added to the realistic scenarios that have been worked with VSNeS and we can affirm that we have achieved the creation of the tool that was intended. This allows us to offer the world of satellite communications research a new way to develop and validate test protocols.

This engine is able to simulate environments defined up to 65 nodes defined by lightweight VMs on a local host, but these performance parameters could be improved depending on the computer used. Thus, the software allows the deployment of tests with large constellations. On the other hand, it has been demonstrated how VSNeS can work under different protocols, offering data that conforms to reality. The clearest case is the use of TCP over a GEO satellite, which implies a very low transmission rate due to latency.

The work described in this thesis has been presented in a paper submitted to Globecom 2022 workshop. In this paper a presentation of the engine is made together with its performance and a preview of the scenarios that have been carried out. In the end, this submission was not accepted. Among the most important points, a comparison between the performance of our engine and other existing engines was requested.

# Future lines of development

The development of this thesis opens the door to several paths. The most important of all is research in satellite communications. This tool has been developed for research purposes and is expected to be a useful tool for New Space projects. The first step in this direction will be done in the research on SDN over satellites by i2cat, where it is expected the possible implementation of the software in Jose Avila's PhD, which is a continuation of his master's thesis [42].

The other important path is the improvement of VSNeS. During the development of the tool, we have been observing concepts that could improve the program. These have not been implemented either because of their high complexity or because they are not directly related to the initial objectives.

A first improvement is the implementation of VMs on the cloud. As observed in the performance study, the RAM memory is a major limitation of the system, the deployment on the cloud would allow access to more resources, increasing the complexity of the scenario or the resources of the VMs.

The second improvement is focused on the configuration load time, which can be high. The improvement would be focused on the creation of a file containing enough information to start the program without performing calculations, avoiding delays in complex scenarios or with high resolution. In this way, we would only have a high loading time the first time the scenario is loaded.

The third important improvement would be to be able to manually control the emulator. Currently, the emulator runs linearly from start to finish and can only be stopped to be restarted. A clear improvement would be if the GUI could interact with the emulator and control it. This way the user could pause and replay moments of the simulation. It would also add value to the GUI, which is currently used in an informative way.

In addition to these ways of improvement, there are others that could also be of interest. These could be the incorporation of another orbit propagator or the possibility of deploying non-linux VMs, among others.

## Sustainability considerations

The development of new emulation methods with which to represent the world is a clear example of environmentally beneficial advances. The development of this type of technology makes it possible to avoid wasting resources on a design that is not expected to be functional. If we focus on VSNeS, it allows the study of the behavior of the network between satellites and how a protocol is affected in it. In this way we can validate new constellations before deployment and how they would work on their application, avoiding inefficient launches where the network does not fulfill the initial expectations. All these processes are performed through virtualization, which allows running several machines on a single host. This reduces the use of different physical machines, reducing costs and environmental resources.

Another benefit in terms of sustainability is the possibility of giving a second life to satellites that have been rendered useless for their initial function, but continue to operate telecommunication systems. VSNeS would allow a study of how this satellite fits in with another constellation or with specific GS.

# BIBLIOGRAPHY

[1] D. S. A. Golkar, O. Korobova, I. L. i Cruz, P. Collopy, and O. L. de Weck, “Distributed earth satellite systems: What is needed to move forward?”, *Journal of Aerospace Information Systems*, vol. 14(8), pp. 412–438, Aug. 2017. DOI: 10.2514/1.I010497.

[2] J. A. Ruiz de Azúa, A. Calveras, and A. Camps, “Internet of satellites (iosat): Analysis of network models and routing protocol requirements”, *IEEE Access*, vol. 6, pp. 20 390–20 411, 2018. DOI: 10.1109/ACCESS.2018.2823983.

[3] X. Lin, S. Rommer, S. Euler, E. A. Yavuz, and R. S. Karlsson, “5g from space: An overview of 3gpp non-terrestrial networks”, *IEEE Communications Standards Magazine*, vol. 5, no. 4, pp. 147–153, 2021. DOI: 10.1109/MCOMSTD.011.2100038.

[4] T. Chen, M. Matinmikko, X. Chen, X. Zhou, and P. Ahokangas, “Software defined mobile networks: Concept, survey, and research directions”, *IEEE Communications Magazine*, vol. 53, no. 11, pp. 126–133, 2015. DOI: 10.1109/MCOM.2015.7321981.

[5] F. Rinaldi, H.-L. Maattanen, J. Torsner, *et al.*, “Non-terrestrial networks in 5g & beyond: A survey”, *IEEE Access*, vol. 8, pp. 165 178–165 200, 2020. DOI: 10.1109/ACCESS.2020.3022981.

[6] M. M. Azari, S. Solanki, S. Chatzinotas, *et al.*, “Evolution of non-terrestrial networks from 5g to 6g: A survey”, *arXiv: Networking and Internet Architecture*, vol. abs/2107.06881, 2021. arXiv: 2107.06881. [Online]. Available: https://arxiv.org/abs/2107.06881.

[7] S. Xu, X.-W. Wang, and M. Huang, “Software-defined next-generation satellite networks: Architecture, challenges, and solutions”, *IEEE Access*, vol. 6, pp. 4027–4041, 2018. DOI: 10.1109/ACCESS.2018.2793237.

[8] Q. Chen, G. Giambene, L. Yang, C. Fan, and X. Chen, “Analysis of inter-satellite link paths for leo mega-constellation networks”, *IEEE Transactions on Vehicular Technology*, vol. 70, no. 3, pp. 2743–2755, 2021. DOI: 10.1109/TVT.2021.3058126.

[9] R. A. Ferrús Ferré, H. Koumaras, O. Sallent Roig, *et al.*, “Sdn/nfv-enabled satellite communications networks: Opportunities, scenarios and challenges”, pp. 95–112, 2016, ISSN: (1874-4907). DOI: https://doi.org/10.1016/j.phycom.2015.10.007.

[10] J. Guo, L. Yang, D. Rincón, S. Sallent, Q. Chen, and X. Liu, “Static placement and dynamic assignment of sdn controllers in leo satellite networks”, *IEEE Transactions on Network and Service Management*, pp. 1–1, 2022. DOI: 10.1109/TNSM.2022.3184989.

[11] L. Wood, “Internetworking and computing over satellite networks”, in Boston, MA, Apr. 2003, ch. Satellite Constellation Networks, pp. 13–34, ISBN: 978-1-4615-0431-3. DOI: 10.1007/978-1-4615-0431-3_2.

[12] “Types of orbits”. (), [Online]. Available: https://www.esa.int/Enabling_Support/Space_Transportation/Types_of_orbits#MEO. (accessed: Jul. 8, 2022).

[13] “Archivo:orbital elements.svg - wikipedia, la enciclopedia libre”. (), [Online]. Available: https://commons.wikimedia.org/wiki/File:Orbital_elements.svg. (accessed: Aug. 31, 2022).

[14] V. A. Chobotov, *Orbital mechanics*, eng, 3rd ed., ser. AIAA education series. Reston, Va: American Institute of Aeronautics and Astronautics, Inc., 2002, ISBN: 1-60086-097-4.

[15] D. J. S. Hernández, “Navegación aérea, cartografía y cosmografía”, in 2008, ch. 2.4. [Online]. Available: https://www.yumpu.com/es/document/read/13257663/navegacion-aerea-cartografa-y-cosmografa-gage-upc.

[16] “Celestrak: ”faqs: Two-line element set format””. (), [Online]. Available: https://celestrak.org/columns/v04n03/. (accessed: Jul. 1, 2022).

[17] “Satellites - orbits - propagators”. (), [Online]. Available: https://help.agi.com/stk/11.0.1/Content/stk/vehSat_orbitProp_choose.htm. (accessed: Jul. 11, 2022).

[18] “Sns3”. (), [Online]. Available: https://www.sns3.org/content/home.php. (accessed: Feb. 18, 2022).

[19] “Os$^3$ - the open source satellite simulator”. (), [Online]. Available: https://omnetpp.org/download-items/OS3.html. (accessed: Feb. 10, 2022).

[20] A. Valentine and G. Parisis, “Developing and experimenting with leo satellite constellations in omnet++”, 2021. [Online]. Available: https://summit.omnetpp.org/2021/assets/pdf/OMNeT_2021_paper_6.pdf.

[21] B. Niehoefer, S. Subik, and C. Wietfeld, “The cni open source satellite simulator based on omnet++”, Jul. 2013. DOI: 10.4108/icst.simutools.2013.251580.

[22] nsnam. “Ns-3”. (), [Online]. Available: https://www.nsnam.org/. (accessed: Feb. 21, 2022).

[23] “Omnet++ discrete event simulator”. (), [Online]. Available: https://omnetpp.org/. (accessed: Feb. 21, 2022).

[24] “Opensand”. (), [Online]. Available: https://opensand.org/content/home.php. (accessed: Fer. 17, 2022).

[25] “Real-time satellite network emulator”. (), [Online]. Available: https://artes.esa.int/projects/realtime-satellite-network-emulator. (accessed: Feb. 14, 2022).

[26] “Scne”. (), [Online]. Available: https://artes.esa.int/projects/scne. (accessed: Feb. 14, 2022).

[27] “Software for digital mission engineering”. (), [Online]. Available: https://www.agi.com/. (accessed: Feb. 21, 2022).

[28] T. Gayraud, P. Berthou, S. Josset, E. Fromentin, and O. Alphand, “Satip6 : Next generation satellite system demonstrator”, *International Federation for Information Processing Digital Library; Broadband Satellite Comunication Systems and the Challenges of Mobility;*, Jan. 2010. DOI: 10.1007/0-387-24043-8_6.

[29] P. Owezarski, P. Berthou, Y. Labit, and D. Gauchard, “Laasnetexp: A generic polymorphic platform for network emulationand experiments”, p. 24, Mar. 2008. DOI: 10.1145/1390576.1390605.

[30] R. Chertov, D. Havey, and K. Almeroth, “Mset: A mobility satellite emulation testbed”, in *Proceedings IEEE INFOCOM*, 2010, pp. 1–9. DOI: 10.1109/INFCOM.2010.5462053.

[31] K. Sobh, K. El-Ayat, F. Morcos, and A. El-Kadi, “Scalable cloud-based leo satellite constellation simulator”, *International Journal of Computer and Information Engineering*, vol. 9, no. 6, pp. 1460–1471, 2015, ISSN: eISSN: 1307-6892. [Online]. Available: https://publications.waset.org/vol/102.

[32] "Cesium: The platform for 3d geospatial". (), [Online]. Available: https://cesium.com/. (accessed: Feb. 28, 2022).

[33] "Core documentation". (), [Online]. Available: http://coreemu.github.io/core/. (accessed: Feb. 28, 2022).

[34] "Networking:netem [wiki]". (), [Online]. Available: https://wiki.linuxfoundation.org/networking/netem#non_fifo_queuing. (accessed: April. 17, 2022).

[35] "Worldwind java/nasa worldwind". (), [Online]. Available: https://worldwind.arc.nasa.gov/java/. (accessed: Mar. 03, 2022).

[36] E. Mahoney. "Disruption tolerant networking". (), [Online]. Available: http://www.nasa.gov/archive/content/dtn. (accessed: Mar. 03, 2022).

[37] I. Tzinis. "Interplanetary overlay network additional information". (), [Online]. Available: http://www.nasa.gov/directorates/heo/scan/engineering/technology/disruption_tolerant_networking_software_options_ion. (accessed: Mar. 03, 2022).

[38] "Satellite network emulation". (), [Online]. Available: https://itrinegy.com/solution-by-task/satellite-link-simulation/. (accessed: Fer. 12, 2022).

[39] "Oracle vm virtualbox". (), [Online]. Available: https://www.virtualbox.org/. (accessed: April. 25, 2022).

[40] "Kvm - debian wiki". (), [Online]. Available: https://wiki.debian.org/KVM/. (accessed: May. 6, 2022).

[41] "Kvm". (), [Online]. Available: https://www.linux-kvm.org/page/Main_Page. (accessed: May. 6, 2022).

[42] J. L. Avila Acosta, "Towards the deployment of software defined networks over satellites - an in-laboratory demonstration for geo satellite services", *Master Thesis, Universitat Politècnica de Catalunya (UPC)*, Jun. 2022. [Online]. Available: http://hdl.handle.net/2117/369611.

[43] "Starlink". (), [Online]. Available: https://www.starlink.com. (accessed: Aug. 18, 2022).

[44] "Ubuntu 20.04.4 lts (focal fossa)". (), [Online]. Available: https://releases.ubuntu.com/20.04/?_ga=2.132604235.414627611.1660654793-744850748.1660654793. (accessed: Jul. 20, 2022).

[45] "Downloads — alpine linux". (), [Online]. Available: https://alpinelinux.org/downloads/. (accessed: Jul. 20, 2022).

[46] "Scripted display tools (sdt)". (), [Online]. Available: https://www.nrl.navy.mil/Our-Work/Areas-of-Research/Information-Technology/NCS/SDT/. (accessed: Jun. 5, 2022).

[47] "Index - cesium documentation". (), [Online]. Available: https://cesium.com/learn/cesiumjs/ref-doc/. (accessed: Jun. 11, 2022).

[48] "Czml guide · analyticalgraphicsinc/czml-writer wiki · github". (), [Online]. Available: https://github.com/AnalyticalGraphicsInc/czml-writer/wiki/CZML-Guide. (accessed: Jun. 11, 2022).

[49] "Skysat - earth online". (), [Online]. Available: https://earth.esa.int/eogateway/missions/skysat. (accessed: Aug. 21, 2022).

[50] "Skyfield — documentation". (), [Online]. Available: https://rhodesmill.org/skyfield/. (accessed: May. 31, 2022).

[51] D. J. Stryjewski, “Coordinate transformations”, *Annalen der Physik*, p. 21, 2020. [Online]. Available: https://x-lumin.com/wp-content/uploads/2020/09/Coordinate_Transforms.pdf.

[52] J. Olmos, “Nacc - laboratoy work 2 - gps positioning and dilution of precision”, *Lecture notes of NACC course (EETAC)*, p. 13, 2020.

[53] “Toml: Tom’s obvious minimal language”. (), [Online]. Available: https://toml.io/en/. (accessed: May. 20, 2022).

[54] “Datetime — basic date and time types — python 3.10.4 documentation”. (), [Online]. Available: https://docs.python.org/3/library/datetime.html. (accessed: May. 10, 2022).

[55] “Math — mathematical functions — python 3.10.5 documentation”. (), [Online]. Available: https://docs.python.org/3/library/math.html. (accessed: May. 30, 2022).

[56] “Multiprocessing — process-based parallelism — python 3.10.5 documentation”. (), [Online]. Available: https://docs.python.org/3/library/multiprocessing.html. (accessed: May. 30, 2022).

[57] “Threading — thread-based parallelism — python 3.10.5 documentation”. (), [Online]. Available: https://docs.python.org/3/library/threading.html. (accessed: May. 31, 2022).

[58] “Welcome to paramiko! — paramiko documentation”. (), [Online]. Available: https://www.paramiko.org/. (accessed: May. 28, 2022).

[59] “Subprocess — subprocess management — python 3.10.5 documentation”. (), [Online]. Available: https://docs.python.org/3/library/subprocess.html. (accessed: May. 31, 2022).

[60] “Sys — system-specific parameters and functions — python 3.10.5 documentation”. (), [Online]. Available: https://docs.python.org/3/library/sys.html. (accessed: May. 31, 2022).

[61] “Time — time access and conversions — python 3.10.5 documentation”. (), [Online]. Available: https://docs.python.org/3/library/time.html. (accessed: May. 20, 2022).

[62] “Webbrowser — controlador de navegador web conveniente — documentación de python - 3.10.5”. (), [Online]. Available: https://docs.python.org/es/3/library/webbrowser.html. (accessed: May. 25, 2022).

[63] W. Pearson. “Toml: Python library for tom’s obvious, minimal language”. (), [Online]. Available: https://github.com/uiri/toml. (accessed: May. 31, 2022).

[64] C. Ledermann. “Czml: Read and write czml in python”. (), [Online]. Available: https://github.com/cleder/czml. (accessed: May. 5, 2022).

[65] D. Zawada. “Julian: Simple library for converting between julian calendar dates and datetime objects”. (), [Online]. Available: https://github.com/dannyzed/julian. (accessed: April. 6, 2022).

[66] “Astropy”. (), [Online]. Available: https://www.astropy.org/. (accessed: April. 6, 2022).

[67] “Welcome to flask — flask documentation (2.2.x)”. (), [Online]. Available: https://flask.palletsprojects.com/en/2.2.x/. (accessed: May. 29, 2022).

[68] “Unittest — unit testing framework — python 3.10.6 documentation”. (), [Online]. Available: https://docs.python.org/3/library/unittest.html. (accessed: April. 5, 2022).

[69] B. Rhodes. “Ephem: Compute positions of the planets and stars”. (), [Online]. Available: http://rhodesmill.org/pyephem/. (accessed: May. 14, 2022).

[70] “Https://www.multipath-tcp.org/”. (), [Online]. Available: https://datatracker.ietf.org/doc/rfc1072. (accessed: Sep. 14, 2022).

[71] “Ibm documentation”. (), [Online]. Available: https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/spectrum-protect/8.1.8?topic=tuning-tcp-flow-control. (accessed: Sep. 05, 2022).

[72] S. Oueslati-Boulahia, A. Serhrouchni, S. Tohmé, S. Baier, and M. Berrada, “Tcp over satellite links: Problems and solutions”, *Telecommunication Systems*, vol. 13, pp. 199–212, Jul. 2000. DOI: 10.1023/A:1019192022690.

[73] “Tcp extensions for long-delay paths”. (), [Online]. Available: https://datatracker.ietf.org/doc/rfc1072. (accessed: Sep. 14, 2022).

[74] J. Olmos, “Navegació aèria, cartografia i cosmografia (nacc) - pràctica 1 - ground track of gps satellites”, *Lecture notes of NACC course (EETAC)*, p. 13, 2022.

# ACRONYMS

**3GPP** 3rd Generation Partnership Project

**ACK** ACKnowledgment

**AGI** Ansys Government Initiatives

**API** Application Programming Interface

**ARP** Address Resolution Protocol

**ARQ** Automatic RepeatreQuest

**CLI** Command-Line Interface

**CNES** Centre National d’Études Spatiales

**CNRS** French National Centre for Scientific Research

**CORE** Common Open Research Emulator

**CPU** Central Processing Unit

**CZML** Cesium Markup Language

**DAMA** Demand Assigned Multiple Access

**DSS** Distributed Satellite Systems

**DTN** Delay-Tolerant Networking

**DVB-RCS** Digital Video Broadcasting - Return Channel via Satellite

**DVB-S2** Digital Video Broadcasting – Second Generation

**ECEF** Earth-Centered, Earth-Fixed

**ECI** Earth-Centered Inertial

**ESA** European Space Agency

**GCS** Geographic Coordinate System

**GEO** Geostationary Equatorial Orbit

**GNSS** Global Navigation Satellite System

**GS** Ground Station

**GUI** Graphical User Interface

**GW** GateWay

**HAPS** High Altitude Platform Stations

**ICMP** Internet Control Message Protocol

**ION** Interplanetary Overly Network

**IP** Internet Protocol

**ISL** Inter-Satellite Links

**JSON** JavaScript Object Notation

**KVM** Kernel-based Virtual Machine

**LAAS** Laboratory for Analysis and Architecture of Systems

**LAN** Local Area Network

**LEO** Low Earth Orbit

**LLH** Latitude, Longitude and Height

**LoS** Line of Sight

**LTP** Local Tangent Plane

**MAC** Media Access Control

**MEO** Medium Earth Orbit

**MPTCP** MultiPath TCP

**MSET** Mobility Satellite Emulation Testbed

**NASA** National Aeronautics and Space Administration

**NCC** Network Control Center

**NED** North East Down

**NFV** Network Functions Virtualization

**NMC** Network Management Center

**NORAD** North American Aerospace Defense Command

**NRL** Naval Research Laboratory

**ns-3** Network Simulator 3

**NT** Terrestrial Network

**NTN** Non-Terrestrial Network

**OMNeT++** Objective Modular Network Testbed in C++

**OS** Operating System

**OS3** Open Source Satellite Simulator

**PDR** Packet Delivery Ratio

**PLR** Packet Loss Ratio

**QEMU** Quick EMUlator

**QoS** Quality of Service

**RAM** Random Access Memory

**RTT** Round Trip Time

**SAT** Satellite

**SATCAT** Satellite Catalog

**SCNE** Satellite Constellation Network Emulator

**SDN** Software-Defined Network

**SDS** Software Defined Satellite

**SDT** Scripted Display Tools

**SGP4** Standard General Perturbations Satellite Orbit Model 4

**SGW** Satellite GateWay

**SNS3** Satellite Network Simulator 3

**SPDA** Static Placement with Dynamic Assignment

**SSH** Secure Shell

**ST** Satellite Terminal

**STK** Systems Tool Kit

**TCP** Transmission Control Protocol

**TLE** Two Line Elements

**TOML** Tom’s Obvious Minimal Language

**UAS** Unmanned Aircraft Systems

**UDP** User Datagram Protocol

**UT** User Terminal

**VLAN** Virtual Local Area Network

**VM** Virtual Machine

**VNF** Virtual Network Function

**VSNeS** Virtual Satellite Network Simulator

**WGS-84** World Geodetic System 84

**WLAN** Wireless Local Area Network

# APPENDICES

# APPENDIX A. ECEF COORDINATES OF A SATELLITE FROM ITS KEPLERIAN ELEMENTS

This section details the algorithm used to propagate an orbit from a TLE. This algorithm has been taken from the practice 1: Ground track of GPS satellites of the NACC course taught by Professor Joan Olmos [74].

The orbits are characterized by the six Keplerian parameters. Using these Keplerian parameters the algorithm compute the ECEF coordinates of any satellite at any time. This algorithm starts with the follow inputs:

- $a$ Orbit major semi axis [m]
- $i_0$ Orbit inclination [rad]
- $e$ Orbit eccentricity
- $\Omega_0$ Longitude of the ascending node (AN) at the ToA [rad]
- $d\Omega_0$ Rate of change of the perigee ar ToA [rad]
- $\omega$ Argument of the perigee at ToA [rad]
- $M_0$ Mean anomaly at ToA [rad]
- $n$ Satellite mean motion [rad/s]
- $dt$ Elapsed time from the ToA [s] (negatiave if the ToA is in the future)

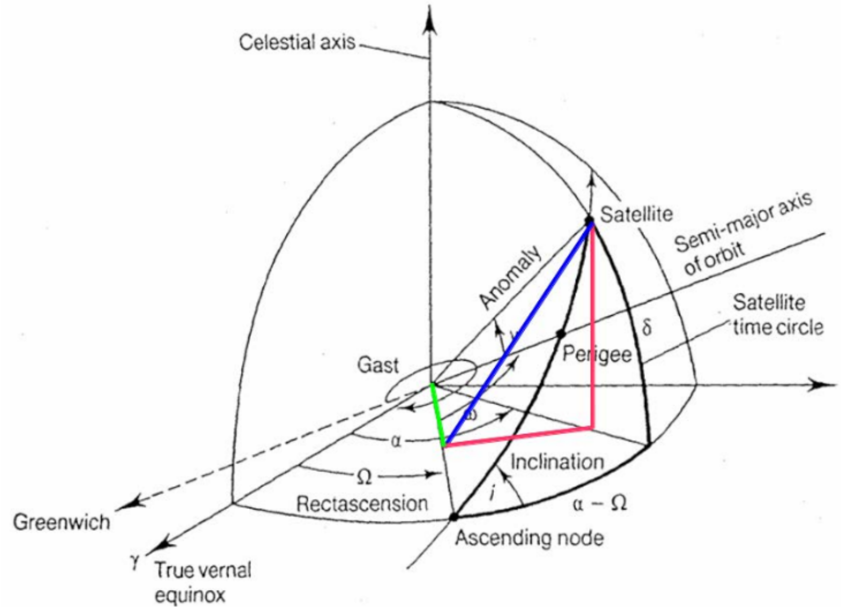


Figure A.1: From Keplerian elements to ECEF coordinates, from [74].

The algorithm apply the following steps to get the satellite ECEF coordinates at specific elapsed time from the ToA (see Figure A.1):

1. Compute the current mean anomaly:

$$M_k = M_0 + n * dt \tag{A.1}$$

2. Find the current eccentric anomaly (solve iteratively for $E_k$):

$$E_k(n) = M_k + e * sin(E_k(n-1)) \tag{A.2}$$

   Take $E_k(0) = M_k$ and compute repeatedly $E_k(n)$ until $|E_k(n) - E_k(n-1)| < 10^{-8}$

3. Find the true anomaly (from the sine and cosine expressions the true anomaly can be extracted without any ambiguity):

$$sin(\upsilon_k) = \frac{\sqrt{1-e^2} * sin(E_k))}{1 - e * cos(E_k)} \tag{A.3}$$

$$cos(\upsilon_k) = \frac{cos(E_k) - e}{1 - e * cos(E_k)} \tag{A.4}$$

4. Find the argument of latitude:

$$u_k = \upsilon_k + \omega \tag{A.5}$$

5. Find the orbit radius (current distance to Earth center):

$$r_k = a * (1 - e * cos(E_k)) \tag{A.6}$$

6. Compute the current longitude of the ascending node using:

$$\Omega_k = \Omega_0 + d\Omega_0 * dt - d\Omega_e * dt \tag{A.7}$$

7. Get x coordinate within the orbital plane:

$$x_p = r_k * cos(u_k) \tag{A.8}$$

8. Get y coordinate within the orbital plane:

$$y_p = r_k * sin(u_k) \tag{A.9}$$

9. ECEF x-coordinate:

$$x = x_p * cos(\Omega_k) - y_p * cos(i_0) * sin(\Omega_k) \tag{A.10}$$

10. ECEF y-coordinate:

$$y = x_p * sin(\Omega_k) + y_p * cos(i_0) * cos(\Omega_k) \tag{A.11}$$

11. ECEF z-coordinate:

$$y = y_p * sin(i_0) \tag{A.12}$$