POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Hardware-in-the-Loop Simulation of Nano-satellite Constellations

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Mattia Siriani**

Student ID: 977552
Advisor: Prof. Luca Mottola
Academic Year: 2022-23

Abstract

Nano-satellites are a milestone in space operation; they provide technologically advanced components, while simultaneously being a compact and cost-effective alternative to traditional satellites. To be more effective, a space mission usually deploys multiple nano-satellites, forming a *constellation*, which is generally used for different purposes, from Earth monitoring to providing worldwide internet access. Every space mission needs a proper testing phase, to increase its likelihood of success and decrease the costs related to planning and implementation: this is where *space simulators* come into play. However, mainstream space simulators cannot test the behavior of nano-satellites in a specific space mission because they do not use the energy consumption data of real nano-satellites, therefore, they cannot simulate all the complexities of a space mission, and the devices involved. Furthermore, the inability of mainstream space simulators to simulate the user-desired specific mission scenario limits the realism of the simulation, hindering a comprehensive understanding of how the nano-satellites would perform in real-world conditions.

Therefore, we implement an orbital simulation of a constellation of nano-satellites, which communicates in real-time with a physical nano-satellite. The literature refers to this type of simulation as *Hardware-In-the-Loop (HIL) simulation*. From the real device, the simulation retrieves the nano-satellite energy consumption, with which it simulates the behavior of the nano-satellites in the simulated constellation while executing the user-desired space mission. Since space radiation impacts the nano-satellites functioning, to enhance realism, our system simulates radiation-induced errors. Furthermore, the scalability we provide enables our system to interact with multiple physical nano-satellites simultaneously by exploiting parallelization in the simulation. The whole system is configurable by the user, to the extent of replacing the physical nano-satellite in a few steps. Our HIL simulation proved capable of managing and integrating real data, retrieved from a physical nano-satellite, in real-time. As for the scalability our system offers, we empirically verified that it leads to an 88% reduction in the total execution time of the simulation and the simultaneous management of ∼700 physical nano-satellites.

Keywords: Nano-satellites, Hardware-In-The-Loop, Simulation, Space radiation, Multi-device, User configurable

Abstract in lingua italiana

I nano-satelliti sono una pietra miliare nel settore spaziale; essi sono dispositivi tecnologicamente avanzati e rappresentano un'alternativa compatta ed economica ad i satelliti tradizionali. Per avere una maggiore resa, una missione spaziale dispiega più nano-satelliti, formando cosí una *costellazione*, la quale può essere utilizzata per diversi scopi, dal monitoraggio della Terra alla fornitura di accesso a Internet in tutto il mondo. Ogni missione spaziale ha bisogno di un'adeguata fase di test, per aumentare le probabilità di successo e diminuire i costi per la pianificazione e la realizzazione: è qui che entrano in gioco i *simulatori spaziali*. Tuttavia, i principali simulatori spaziali non possono testare il comportamento dei nano-satelliti in una specifica missione spaziale non utilizzando dati sul consumo energetico presi da nano-satelliti reali, quindi non possono simulare le complessità di una missione spaziale e dei dispositivi coinvolti. Inoltre, l'incapacità dei simulatori spaziali di simulare lo scenario desiderato dall'utente limita il realismo della simulazione, impedendo una comprensione completa di come i nanosatelliti si comporterebbero in condizioni reali. Perciò, abbiamo implementato una simulazione orbitale di una costellazione di nano-satelliti, che comunica in tempo reale con un nano-satellite fisico. La letteratura si riferisce a questo tipo di simulazione come *simulazione Hardware-In-the-Loop (HIL)*. Dal dispositivo reale, la simulazione recupera il consumo energetico del nano-satellite, simulando il suo comportamento nella costellazione simulata durante l'esecuzione della missione spaziale specificata. Poiché le radiazioni influenzano il funzionamento dei nano-satelliti, il nostro sistema simula gli errori indotti dalle radiazioni, per aumentare il realismo. Inoltre, la scalabilità offerta consente al nostro sistema di interagire con più nanosatelliti fisici simultaneamente, sfruttando la parallelizzazione nella simulazione. Il sistema è configurabile dall'utente, fino a poter sostituire in pochi passaggi il nano-satellite fisico.
Il nostro sistema si è dimostrato in grado di gestire ed integrare dati reali, recuperati da un vero nano-satellite, in real time. Per quanto riguarda la scalabilità che il nostro sistema offre, abbiamo verificato che essa riduce dell'88% tempo totale di esecuzione della simulazione e permette di gestire ∼700 nano-satelliti fisici in contemporanea.

Parole chiave: Nano-satelliti, Hardware-In-The-Loop, Simulazione, Radiazioni spaziali, Multi-dispositivo, Configurabile dall'utente

Acknowledgements

It was a challenging journey that made me understand many things about myself and helped me to develop the ability to face a problem from different perspectives.
I learned so much about the space environment, low-level software, hardware components, and the inner workings of simulators.
First, I want to thank my advisor, Prof. Luca Mottola, for all the support he gave me throughout this year while working on the thesis.
Then, I want to give a shoutout to my lab buddies, who helped me along my journey and also made my experience at NESLab beautiful. Furthermore, I want to thank all my friends, inside and outside the university, who made this journey even better.
Moreover, I thank my family for their encouragement and support throughout my studies, from elementary school up to this moment. Their faith in me has been a constant source of inspiration and motivation.
Last but not least, I want to thank my better half, Elisa, for always supporting me and continually giving me the strength to pursue my goals and never giving up, I am very grateful for her presence in my life.

Contents

List of Figures

List of Tables

1 | Introduction

Satellites are powerful instruments that have been used throughout the last decades for different purposes. Some of these are scientific discoveries, monitoring the Earth, and providing navigation systems, such as the Global Positioning System (GPS) [57] [25] [77]. However, the regular cost of a space mission involving satellites is tremendously high [73]. This reason led to the research and development of nano-satellites.

Nano-satellites are compact, but still powerful satellites, which are also significantly cost-effective compared to traditional satellites. Usually, a nano-satellite mission involves the use of multiple nano-satellites, to form a *constellation.* These constellations usually orbit close to Earth; however, they orbit above the atmosphere, therefore the nano-satellites are more prone to the impact of space radiation with their technological components. Radiation in space is dangerous for the technological components of the nano-satellites because the interaction of particles of the radiation with a component can severely damage it [17]. However, even if the mission of a constellation of nano-satellites is cost-effective compared to a satellite mission, launching a constellation of nano-satellites remains a significant cost. In addition, once the constellation of nano-satellites is deployed into space, we do not have full control over them. Therefore, every space mission needs to undergo a deep planning and testing phase to increase its likelihood of success. In the planning and testing phase, there is a study on the feasibility of the space mission by identifying and addressing potential challenges or issues before the actual deployment.

Space simulator plays a pivotal role in enhancing the chances of a successful space mission, by simulating the space mission in a controlled environment. The simulation of space missions also helps reduce the overall development cost by identifying and addressing potential issues in the virtual environment, mitigating the need for costly corrections and modifications in the actual implementation phase, which encompasses physical construction, assembly, and launch in the space. Usually, a space simulator of nano-satellites can simulate the communication between the nano-satellites and the ground stations on Earth or is more focused on simulating the orbit of the nano-satellites.

In the scope of this thesis, we focus on the second type of simulation, specifically the orbital simulation of a constellation of nano-satellites.

1.1. Problem and Contribution

Our main problem relies on the fact that a standard space simulator cannot test the behavior of the nano-satellites in a specific space mission scenario because it relies only on artificial data or generated data, which cannot simulate all the complexity of a space mission and the devices involved. Furthermore, the inability of standard space simulators to simulate the user-desired specific space mission scenario limits the realism of the simulation, hindering a comprehensive understanding of how the nano-satellites would perform in real-world conditions. Moreover, every space mission scenario provides a vastness of specific space missions, enhancing the limit of standard space simulators. For instance, in the context of an Earth monitoring scenario, some specific missions could be the monitoring of wildfires, the monitoring of ice melting, and the monitoring of the temperatures around the Earth.
Therefore, we focus on providing a simulation of a constellation of nano-satellites, which amplifies its realism by retrieving data directly from a physical nano-satellite. Our system must be able to simulate the user's desired mission scenario and offer flexible configuration options to the end user. Our system must be scalable and able to interact with multiple physical nano-satellites simultaneously to decrease the execution time of the simulation.

Starting from the *Computing On The Edge (cote)* simulator [41], we build a HIL simulation, which simulates a space mission scenario of a constellation of nano-satellites. The HIL feature enables the simulation to process real data retrieved from a real nano-satellite and not auto-generated data. To retrieve the real data we use on-site testing; specifically, the nano-satellite we use is the CubeSat Simulator (CubeSatSim), which is an open-source, cost-effective device that is able to emulate all the functionalities of a standard CubeSat.
The space mission scenario is simulated through the execution of programs, called tasks, on the real nano-satellite. A user is able to simulate different objectives of the mission scenario, called jobs, by coordinating the execution of multiple tasks. The simulation computes the behavior of the nano-satellites in the simulated constellation, by retrieving the data from a physical nano-satellite. This data pertains to the environment and the behavior of the physical nano-satellite; specifically, in the context of this thesis, we use a subset of the data retrieved from the physical nano-satellite, which is the voltage and current of the nano-satellite. By using the data of voltage and current we are able to compute the energy consumption of the nano-satellites in the simulated constellation.
We structure the nano-satellites in the simulated constellation, which we refer to as "simulated nano-satellites", as they are the simulated counterparts of the physical nano-satellite.

Therefore, each simulated nano-satellite must interact with a physical nano-satellite to be simulated.
We enable reliable and platform-independent communication between the simulation and the nano-satellite. The reliability of the communication channel is a key feature to enable the correct functioning of the system, while the platform-independent property provides more flexibility to the system.
The simulation of how the radiation impacts the nano-satellites is not trivial because involves modeling the intricate interaction between the radiation particles and the various components of the nano-satellite. However, the integration of a real nano-satellite in our system enables the simulation to simulate the radiation-induced errors in a more realistic way. These errors are simulated directly on the nano-satellite, aiming to reproduce better the effects caused by spatial radiation. The simulation simulates the errors induced by radiations, by corrupting the programs that simulate the space mission scenario.
We scale our system on both the nano-satellite side and the simulation side. On the nano-satellite side, we enable our system to manage the interaction with multiple physical nano-satellites simultaneously. On the simulation side, we exploit parallel execution to speed up the execution time of our simulation.
The whole system is highly configurable by the user. The user can configure simulation parameters, nano-satellite parameters, and the file structure of the system. Moreover, the user must provide the programs that simulate the space mission scenario and he also must control the assignment of the objectives of the mission scenario to each nano-satellite in the simulated constellation. The last thing that the user can configure is the physical nano-satellite employed in the HIL simulation, which can be substituted in a few steps without implementing again the whole system.
When the simulation ends, it generates as output different plots regarding its execution.
For our evaluation, we use the following setup. Our simulation runs on a machine with 48 CPUs, 256 GB of memory, and 1 TB of disk. For the physical nano-satellite involved in the HIL simulation, which is the CubeSatSim, we decided to mount the Raspberry PI 2 Zero W as the main board. We evaluate our system under three metrics: realism, fidelity, and scalability. In the realism metric, we set the system with 10 simulated nano-satellites, 10 jobs, and 1 physical nano-satellite. For this metric, we want to evaluate if our system can retrieve real data from the physical nano-satellite and is able to combine the real data retrieved with the data generated by the simulation in the corresponding nano-satellite of the simulated constellation, which requests the simulation of a space mission objective; providing a comprehensive HIL simulation. Specifically, we compare the energy consumption of the nano-satellites in the simulated constellation to the energy consumption of the components of the CubeSatSim, found in datasheets. From the results obtained, we prove

that our simulation is able to combine real data in real time.
In the fidelity metric, we focus on the physical nano-satellite to record the data regarding the energy consumption for 10 minutes while it is working. For this metric, we aim to record on the physical nano-satellite an increasing amount of data on the current and voltage of the nano-satellite to evaluate if this provides a more accurate result of the energy consumption. From the results obtained, we noted that the more data we record, the more accurate the results obtained because by collecting a higher amount of data there is a higher probability of recording current peaks, unseen by collecting less data.
In the scalability metric, we set the system with 10 simulated nano-satellites, 20 jobs, and 1 physical nano-satellite, to evaluate three different aspects. For this metric we want to evaluate: the behavior of the simulation on the increasing number of jobs, if the scalability in terms of the number of physical nano-satellites is able to decrease the execution time of the simulation, and the overall scalability of our system, to understand if it can manage the increased parallelization, maintaining the performance when the number of nano-satellite in the simulated constellation and physical nano-satellite increases.
From the results obtained, we can make the following observations regarding the scalability of our system. The increase in execution time is linear to the increase in the number of objectives of the space mission scenario simulated. The increase in the number of physical nano-satellites makes the execution times of the simulation faster up to a factor of 88% when using multiple physical nano-satellites, compared to using a single physical nano-satellite. Therefore, our approach to scalability proves capable of reducing the execution time of the simulation. Lastly, our system is able to handle the increased parallelization and maintain its performance under higher counts of nano-satellites until it saturates the processing cores on the machine that runs the simulation, also proving capable of handling up to ∼700 physical nano-satellites simultaneously. To make a comparison, our HIL simulation is able to manage nearly all the real nano-satellites of the OneWeb constellation, which plans to deploy 720 nano-satellites [74].

1.2. Thesis Structure

This thesis is structured in nine chapters. Initially, we give an overview of the space environment of the nano-satellites, and we outline the main simulation technologies regarding space. Then, we describe in detail our problem. Furthermore, we describe how we structured and implemented our system. Lastly, we report the experimental evaluation we conducted and the results we obtained.
In the following, we provide a summary of the chapters of our thesis:

- In **Chapter 2** we provide an overview of the space environment for nano-satellites. First, we analyze what are nano-satellites and how they work in space. Lastly, we describe popular space applications, focusing on specific application scenarios used in the context of constellations of nano-satellites.
- In **Chapter 3** we analyze the state of the art of the simulation systems and of the nano-satellite we used in our system. First, we analyze what we target in this thesis, to provide context on the decisions we made. Then, we analyze different types of simulators, ranging from general-purpose simulators to space communication simulators and orbital simulators. After this, we describe the general working of a HIL simulation, and finally, we analyze the nano-satellite we used in our system.
- In **Chapter 4** we formulate the problem of this thesis work. First, we analyze the main problem, which is the implementation of a HIL simulation that is able to simulate the specific mission scenario desired by the user, with all the subproblems that this implementation brings, such as interfacing the simulation and the physical nano-satellite, synchronizing the simulation and physical nano-satellite to work in conjunction, and retrieving real data from a physical nano-satellite. Furthermore, we analyze the problem of simulating errors caused by radiation on the nano-satellites. Then, we delve into the other two topics addressed by this thesis, which are the ability of the system to give high-configuration capabilities to the user and the focus on enhancing the scalability of the system.
- In **Chapter 5** we describe the design choices we made in the implementation of our system. First, we define the main structure of the components that simulate the specific mission scenario. Then, we analyze all the design choices we made for our system on the simulation side, on the nano-satellite side, and on the communication channel. Lastly, we define our design approach to scale our system.
- In **Chapter 6** we describe all the possible ways the user can configure the system. These configurations range from modifying simulation parameters to managing the simulated space mission scenario, and in conclusion, to the steps needed to modify the physical nano-satellite involved in the HIL simulation.
- In **Chapter 7** we describe the actual implementation on the simulation side and on the nano-satellite side. On the simulation, we describe how we synchronize the simulation with the physical nano-satellite and how the data retrieved from the real nano-satellite are requested and combined inside the simulation. On the physical nano-satellite, we describe the program that simulates the radiation-induced errors and the programs that record and send the data to the simulation. In conclusion,

we analyze the communication protocols we used to connect the two sides of our system.

- In **Chapter 8** we report on the experimental evaluation we conducted on our system. First, we define the inputs that we set to evaluate our system, and after this, we showcase the results obtained. Our evaluation proves that our system is able to integrate real data from a physical nano-satellite and that our scalable approach decreases the execution time of the simulation up to a factor of 88% when using multiple physical nano-satellites, compared to using a single physical nano-satellite, and that our system is able to support ∼700 physical nano-satellites working simultaneously.
- In **Chapter 9** we elaborate on conclusions we derived from our thesis and some possible future work.

2 | Background

In this chapter, we introduce in Section 2.1 the current space technologies in the field of satellites and nano-satellites. In addition, in Section 2.2 we describe some of the most interesting and utilized space applications, especially for nano-satellites. We aim to present a broader perspective on the context of space and satellite technology and applications, offering a more comprehensive understanding of this thesis work.

2.1. Space Technology

Throughout history, human curiosity has always been insatiable, and in this era defined by technological advancements, the space realm has never been more within reach. In trying to conquer space, humankind developed different technologies to retrieve data and operate in space, e.g., satellites. However, the soaring cost of space access slowed down these ambitions. As a reminder of this financial challenge, a typical weather satellite carries a staggering price tag of \$290 million, while the cost of a spy satellite might escalate by an additional \$100 million [73]. For this reason, efforts have been made to increasingly make this field more and more accessible, which led us to the domain of nano-satellites.

2.1.1. Nano-satellites

In the last few years, nano-satellites emerged in the realm of space as a new major technology. Nano-satellites are small-sized satellites, offering a powerful approach to satellite design, while simultaneously being a cost-effective alternative compared to a traditional satellite.

An interesting class of nano-satellites is the CubeSats. CubeSats are standardized nano-satellites that adhere to the CubeSat Design Specification (CSD) [56], which defines every characteristic of them. CubeSats come in various configurations, denoted by the number of "U" units they comprise. The dimension of a CubeSat can range between 0.25U to 27U and its weight can go from $\sim$0.2 kg up to $\sim$40 kg [61]; we can see the most used formats in Figure 2.1.

The standardization of CubeSat made space more accessible, since the simplification of the whole development cycle made it possible to lower the total cost of the mission, considering also the fact that these nano-satellites can be launched as secondary payloads on bigger missions. Furthermore, thanks to CubeSats' modular design, the process of assembly and integration of the components is easier and consequently, the modular design shortens also the development time, compared to traditional satellites.
Despite their compact size, imposed by the standardized designs, CubeSats can be equipped with advanced technologies. They can carry various devices, including high-definition cameras, communication systems, various types of sensors, and propulsion modules; depending on the specific mission design.
Most of the time CubeSats are used for scientific discoveries as we explain in Section 2.2.1, Earth observation as we analyze in Section 2.2.2, and interstellar communications as we describe in Section 2.2.3.

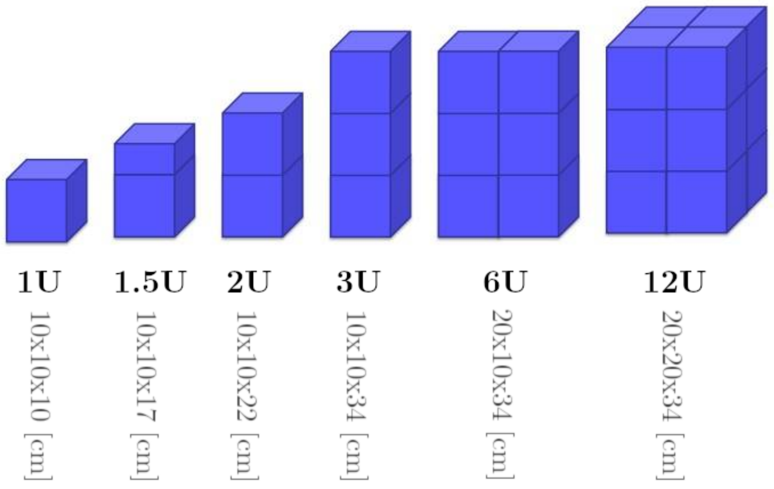


Figure 2.1: CubeSat satellite formats [64].

2.1.2. Nano-satellite Constellations

A nano-satellite constellation is a network of nano-satellites, which collaborate to achieve various objectives. Compared to traditional satellites, this is an innovative approach to space missions, which guarantees different benefits:

- **Global coverage and faster revisit time**: Constellations of nano-satellites offer a significant advantage in terms of global coverage, and thanks to this property they are able to revisit the same position in a faster time. These properties are crucial in the context of Earth observation because we can grant a higher data collection and a faster response when a dynamic event occurs.

- **Redundancy and reliability**: The presence of multiple nano-satellites in constellations guarantees redundancy. This property ensures that even if a nano-satellite fails to execute some assigned task or breaks down, we can still reach the goal of the mission by assigning the workload of the non-functioning nano-satellite to a functioning one. Therefore, the whole mission's reliability increases.
- **Data fusion and accuracy**: Within a nano-satellite constellation, a powerful approach for gathering information is data fusion. Instead of having a single source of information, a nano-satellite constellation retrieves data from multiple nano-satellites. By combining these data sources, we are able to correct errors and manage outliers, enhancing the overall accuracy.
- **Scalability and flexibility**: The nature of constellation allows for easy expansions or modifications of it. This implies that a nano-satellite can be excluded from the mission's objective of its constellation while physically remaining in the same orbit. This is particularly useful when a nano-satellite is no longer functional or when the objectives of the mission change.
- **Accessibility**: To improve the accessibility of the space environment and at the same time reduce its cost, nano-satellite constellations are often deployed as secondary payloads during larger space missions.

2.1.3. Spatial Orbits

The movement of any type of object in space follows the laws of motion of Isaac Newton and his law of universal gravitation. Therefore, when there is no significant external force, an object in space moves in a straight line without acceleration. However, when the presence of a massive body arises, for instance, a planet or a star, the object modifies its path and starts to follow a curved trajectory under the influence of the gravitational force of the massive body. The new curved path followed by the object is known as spatial orbit. Understanding these spatial orbits is critical to the success of any space mission dealing with the launch of a satellite or a nano-satellite constellation.
The process of reaching an orbit with a satellite is a really complicated maneuver. However, to comprehend the concept, we will simplify it in the following description.
As shown in Figure 2.2, first the satellite needs to reach the desired altitude, and in the end, it needs a "push" by the rocket's engine to enter an orbit; at this point, the satellite starts to follow its orbit under the influence of the gravitational force, nearly without any additional assistance [23].

An important consideration regarding orbits is the difference when they are visualized in 2D and in 3D. When a satellite is launched, it will follow a circular orbit around the Earth; but when this orbit is visualized on a 2D plane, a moving translated sine wave is shown. This happens because, as the satellite follows a circular path around the Earth, the latter rotates on its own [63]. A visual representation of this phenomenon is visible in Figure 2.3a and in Figure 2.3b.

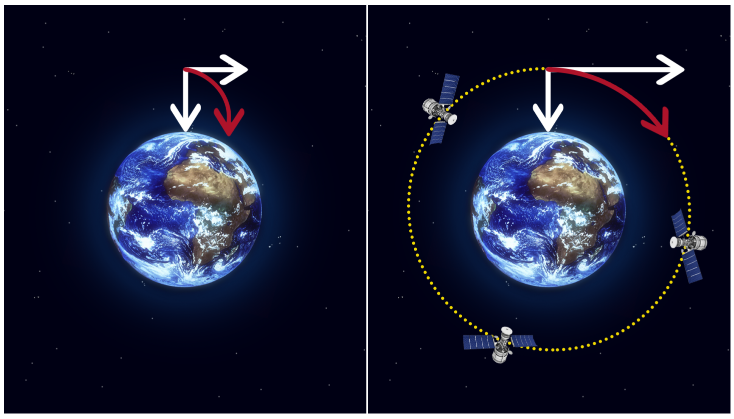
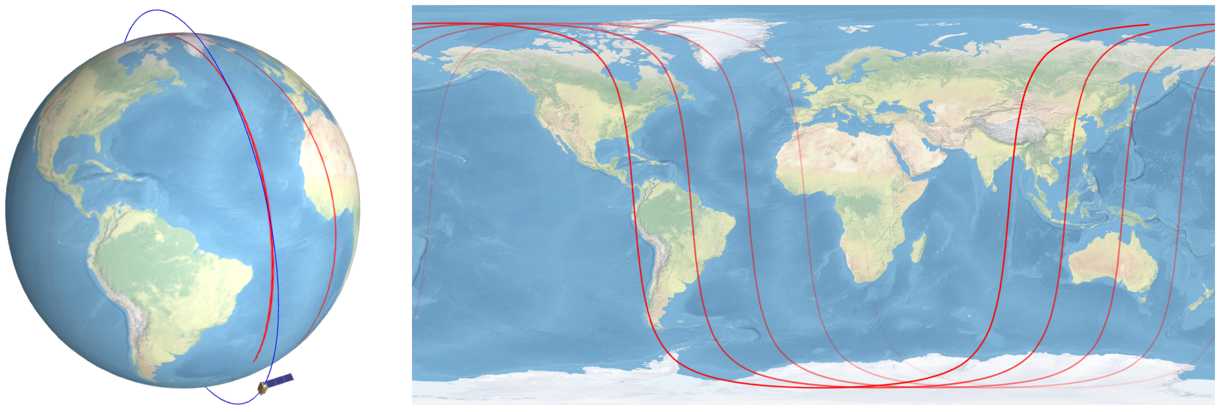


Figure 2.2: Reaching the orbit [19].


(a) 3D orbit.

(b) 2D orbit.

Figure 2.3: Orbit representations [63].

Around the Earth, the most widely used spatial orbits are:

- **LEO**: LEO is the closest orbit used by satellites. The spacecrafts in this orbit do not have an exact path to follow and they are usually at an altitude of 1000 kilometers, but they can range from 160 kilometers to 2000 kilometers. Thanks to its proximity to the Earth's surface, LEO is commonly used for Earth observation, further explained in Section 2.2.2. Another important service that satellites orbiting in LEO provide are telecommunications, which is usually accomplished by nano-satellite constellations to achieve a higher coverage [21]. We can find an in-depth description of interstellar communication in Section 2.2.3.

- **Geostationary Earth Orbit (GEO)**: GEO is an orbit that, unlike LEO, follows a path above the equator, and the satellites in this orbit travel at the exact same rate as the Earth. This characteristic makes the satellites that are using this orbit seem "stationary" relative to the Earth's surface. This "stationary" characteristic makes the satellites in this path suited for telecommunication with the ground because antennas from Earth can always point at a fixed point and always reach the satellite. Other satellites that are appropriate for the GEO orbit are the weather-monitoring ones. These satellites are particularly suited because, by exploiting the property of this orbit to be in the same fixed position relative to Earth, satellites can continuously observe the same location [20].

- **MEO**: MEO is an orbit located between LEO and GEO, indeed it ranges from 2000 to 35768 kilometers above the Earth's surface. The number of satellites used to achieve global coverage in the MEO orbit is less when it is compared to LEO because, as illustrated in Figure 2.4, thanks to its greater distance from Earth the coverage of a single satellite is wider.
  Furthermore, because of MEO's far distance from Earth, satellites orbiting in this orbit are more prone to radiation, and for this reason need to be shielded with radiation-resistant material, such as gold, aluminum, and kevlar [77]. MEO is commonly used by Global Navigation Satellite Systems (GNSS) such as the United States Global Positioning System (GPS) and the European navigation system, also called Galileo [22][77].

- **Polar Orbit**: Satellites in polar orbit are located at an altitude between 200 and 1000 kilometers. In this orbit, the satellites follow a path perpendicular to the equator, passing approximately over the poles [24]. A particular type of polar orbit is the so-called Sun-Synchronous Orbit (SSO), which has the particularity of being in sync with the Sun. This feature enables the satellites orbiting in SSO to be in the same fixed position relative to Earth, at the same clock time. This feature of SSO is particularly useful for Earth monitoring tasks because, for instance, if we are monitoring a precise position on Earth at midnight, we are not interested in photos taken at different times [24].

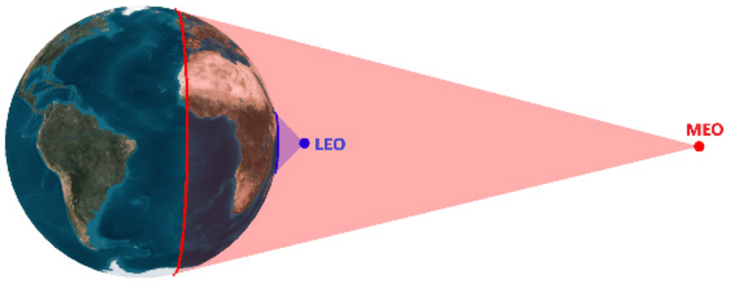


Figure 2.4: Coverage comparison between satellites in MEO and in LEO [77].

2.1.4. Radiation into Outer Space

Radiation in space is a concerning problem when we deal with missions of satellites or nano-satellites in space.
In the following we describe, what are these radiations, where they come from and why are they so dangerous for a satellite mission. Space radiation is not like Earth's radiation. Space radiation involves atoms that have lost their electrons while accelerating in space, nearly reaching the speed of light. Eventually, only the nucleus of the atom is left.
Radiations originate from various cosmic events, like particles trapped in the magnetic field of the Earth, particles released into space from the Sun, from which we receive the highest amount of radiation, and galactic cosmic rays from distant cosmic phenomena [72].
These radiations can have different unwanted effects on the satellite's components. The most often observed phenomena are the Single Event Effects (SEEs). These events occur when a particle collides with a component, generating a temporary electric charge, which leads to various possible consequences, such as memory bit-flipping, an error on the software, or in the most severe cases a short circuit, that results in a no more functioning satellite. For long-term missions, the Total Ionizing Dose (TID) becomes a problem for satellites. This is a continuous exposure to radiation, which brings a gradual functional deterioration [17][49]. Finally, the impact of the radiation can also affect the signals used for communication, whether transmitted from the satellite to Earth or vice-versa, potentially leading to errors or data corruption.
Satellites and nano-satellites exploit different techniques to reduce the errors caused by the radiation and increase the overall mission's reliability:

- **Shielding**: We can shield some components with materials that are able to absorb or redirect the radiation. Some examples of materials that shield from the radiations are gold, aluminum, and kevlar [77]. We use this prevention technique, above the satellite's components, to keep them safe from space radiation.

- **Error detection and correction codes**: We can exploit coding techniques to detect and correct errors caused by radiation. An interesting example is the Matrix Operation Microprocessor Architecture (MoMa), a low-power embedded architecture, capable of detecting and recovering errors with a success rate of 98,7% on a benchmark with 672,348,891 injected faults [43].

- **Radiation-hardened processor**: The use of radiation-hardened processors helps minimize the error caused. For instance, the Lunar Reconnaissance Orbiter, which is a satellite orbiting the Moon, uses the RAD750^{TM}, a radiation-hardened processor, to be more resistant to radiation[49]. Usually, the cost of a radiation-hardened processor is really high, for instance, the RAD750^{TM} has a price of ~200,000$ [68], which is definitely an outlier compared to the total cost of a nano-satellite. Therefore, missions involving nano-satellites usually use cheaper processors, such as the Raspberry PI's processor.

- **Redundancy**: By designing a satellite with redundant components we can guarantee the functionality of the system, even if a part is affected by radiation.
  When using redundancy an aspect to consider is the number of redundant components that we use to execute our operations. If we use all the components including the redundant ones, we need to be sure to synchronize the operation between all the components, and if a component stops working it must be excluded from the workload sharing. Instead, when we rely on a single component to execute our operations, it is essential to ensure that the redundant components remain updated. This approach is crucial when the primary component breaks. In this situation, the redundant component must resume operations from the same point where the previous one was interrupted. A method to achieve this is checkpointing.
  Checkpointing is a key property when we deal with redundant systems. This property can be achieved by periodically saving the state of the system on a stable storage. In this way, when a component fails, we can proceed with the redundant component from the last checkpoint saved.

2.2. Space Applications

In the following sections, we analyze some application domains that use both satellites and nano-satellites. In Section 2.2.1 we describe some scientific discoveries made possible by the utilization of satellites. Then, in Section 2.2.2 we describe why Earth observation is important and how we can achieve it, exploiting satellites and nano-satellites. In conclusion, in Section 2.2.3, we analyze how internet communication from space is enabled by nano-satellite constellations.

2.2.1. Scientific Discoveries

Satellites facilitate a wide range of progress in scientific discoveries, contributing to a better understanding of our universe.
Some interesting examples are:

- **Mapping the universe with the James Webb Space Telescope**: The James Webb Space Telescope (Webb) is one the most ambitious and intricate science programs ever undertaken. It was designed by the National Aeronautics and Space Administration (NASA), in collaboration with the European Space Agency (ESA) and the Canadian Space Agency (CSA). Its primary objective is to address crucial questions regarding the origins of galaxies, stars, and planets within the Universe. The instrumentation aboard Webb enables multiple types of scientific applications, targeting a wide range of astronomical objects [57]. From its launch, the Webb Telescope has discovered hidden details of galaxies, fresh insights regarding planets outside our solar system, and new images of the Unseen Universe with an unprecedented level of details, like the Carina Nebula and the Southern Ring Nebula, as we can see respectively in Figure 2.5a and in Figure 2.5b.

- **Cosmic Microwave Background (CMB) Radiation**: The Cosmic Background Explorer (COBE) satellite was developed to provide fundamental measurements of celestial radiation. A portion of this radiation is thought to have come from the events that took place at the beginning of the universe. Measuring this radiation is a milestone in understanding how the universe has originated and evolved [50]. Even with some uncertainty, the COBE satellite provided critical data to study the CMB radiation and the Cosmic Infrared Background (CIB) radiation [33][52]. Thanks to the widespread use of nano-satellites, also CubeSats have been used to detect and analyze these types of radiations [35][53].

(a) Carina Nebula [47].

(b) Southern Ring Nebula [16].

Figure 2.5: Images taken by the James Webb Space Telescope.

2.2.2. Earth Observation

Satellites help us monitor dynamic events happening on Earth while gathering a wide volume of data about Earth's surface. Image acquisition is an ongoing process that helps us to better understand the environment and respond to natural disasters.

The historical record of all the data retrieved from the satellites enables the discovery of patterns, revealing alarming trends. Some important examples discussed nowadays are the rise in sea level and the growing number of wildfires [25]. Earth observation has a huge impact on weather forecasting, helping to make climate prediction more realistic.
We can see how crucial Earth observation is for us, just by looking at the number of satellites deployed by ESA, which we can see in Figure 2.6.
In the last years, nano-satellites revolutionized Earth observation, thanks to their small size, advanced technological components, and low cost compared to satellites. The key advantage that nano-satellites bring is provided by their deployment in constellations. Exploiting the features of nano-satellite constellations already discussed in Section 2.1.2, nano-satellites can provide more data and better monitoring of dynamic events.

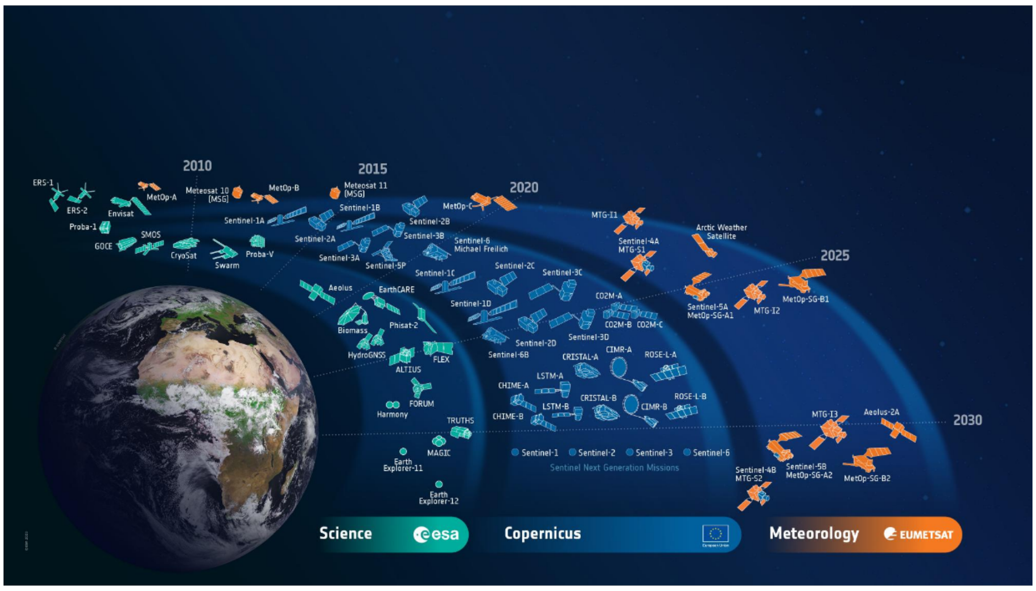


Figure 2.6: Earth observation missions [26].

2.2.3. Interstellar Communications

In the last few years, the increasing adoption of nano-satellite constellations made internet connectivity provided directly by nano-satellites a reality. Nano-satellites are extremely important in this field because even though their size is small, they carry advanced technological components, which are able to bring high-speed connectivity to Earth's surface. To bring the fastest possible internet connection, the nano-satellites orbit in the trajectory closest to the earth, which is LEO. This innovative technology is currently being pushed by major tech giants like SpaceX Starlink [11] and OneWeb [7]. The number of satellites that these two technologies plan to put in LEO to achieve global broadband is astounding: SpaceX aims to send into LEO nearly 12,000 nano-satellites, while OneWeb plans a similar global coverage using 720 nano-satellites [51] [67] [74]. In Figure 2.7 we can better visualize the already enormous amount of nano-satellites used by SpaceX Starlink, currently undergoing ∼5,000 nano-satellites.

Furthermore, another important aspect of this new interstellar communications is the ability to provide internet connectivity in regions where establishing traditional infrastructure is logically and economically challenging.

In conclusion, the applications we examined are deeply used in the context of nano-satellites. However, when developing such types of applications a rigorous testing campaign is crucial to ensure everything works properly before launching a nano-satellite into space, making simulation a valuable tool for guaranteeing their functionality.

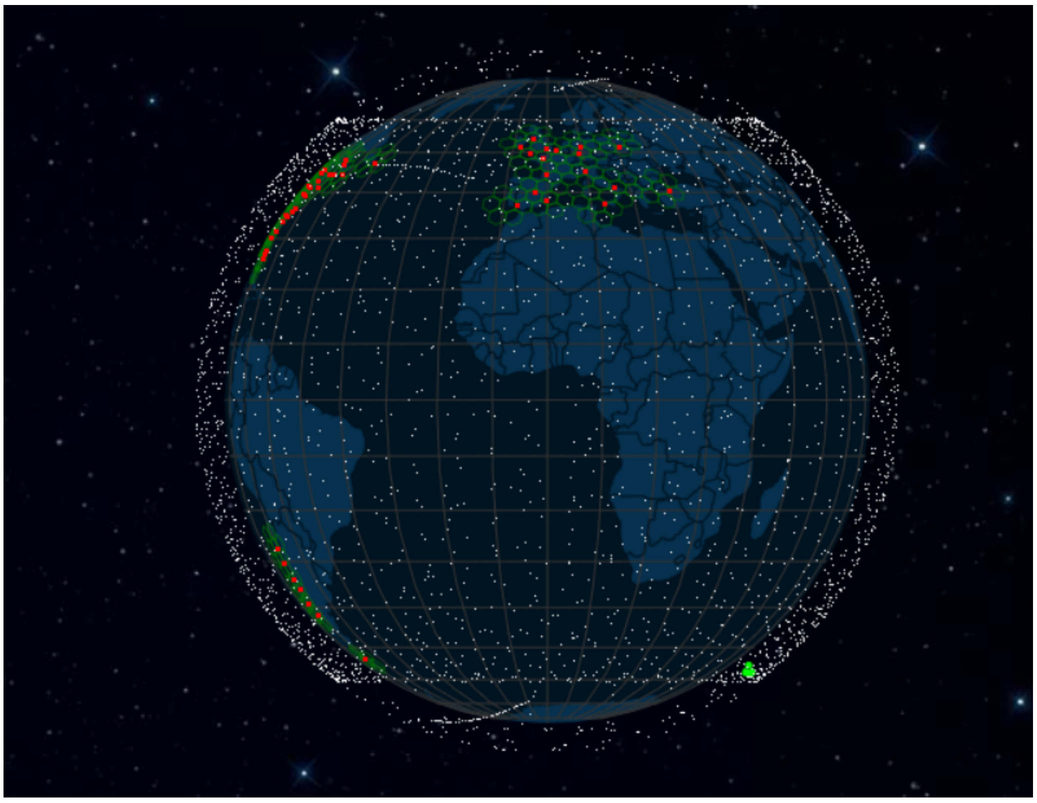


Figure 2.7: SpaceX Starlink constellation [12].

In conclusion, in this chapter, we analyze different general aspects of the technology in space, to introduce the space scenario of our interest. An in-depth examination of the technologies we have analyzed or used for our system is available in Chapter 3

3 | State of the Art

In this chapter, we introduce the state of the art of the main topics discussed in this thesis. First, in Section 3.1 we describe what are the targets of this thesis work. This section is pivotal for understanding the reasoning of the choices made in the other section of this chapter.
Additionally, in Section 3.2 we introduce the concept of simulation and discuss some general purposes we examined. Then, in Section 3.3 we discuss different simulators, which we take under consideration to use as a basis for the development of our simulator. Next, Section 3.4 describes how a HIL simulation generally works. Lastly, in Section 3.5 we discuss the hardware and software of the nano-satellite used for developing our simulator. The arguments discussed in this chapter lay the basis for comprehending the decisions we made and the underlying principles of the simulator we developed.

3.1. Target Applications

The objective of our project is to create an orbital simulator that is able to simulate the processing of a nano-satellite constellation, along with its recharging through time. Our simulator includes in its loop a real nano-satellite, enabling this simulation to be a Hardware-In-the-Loop simulation.
The programs that we provide to simulate the processing in the simulation, are programs regarding image processing and signal processing. These programs aim to simulate Earth observation and telecommunication scenarios, which are general scenarios that involve the use of nano-satellites.
Finally, the device we decide to use is the CubeSatSim, described in Section 3.5, which is an open-source, cost-effective project, that is able to emulate all the functionality of a CubeSat. Furthermore, the CubeSatSim is equipped with many different sensors that record data from the environment and from how the nano-satellite behaves. For instance, we can record various parameters such as the nano-satellite's speed, the external temperature, and the nano-satellite's power consumption, which the latter plays a fundamental role in implementing our HIL simulation. Moreover, on the CubeSatSim the maintenance

is more straightforward, and its modular design facilitates hardware modifications [54].

3.2. Simulation Technology

Simulations are a milestone in the discipline of computer science, serving as a fundamental tool that deeply influences research, experimentation, and problem solving. Simulations have been used for a very long time; for instance, the first computer simulation dates back to World War II, when the mathematicians Jon Von Neumann and Stanislaw Ulam were trying to solve a problem regarding neutron's properties [15].
In the last decades, the number of space missions involving satellites and nano-satellites has been increasing, and simulators play an important role in making them possible. These tools offer many benefits in the successful launch of satellites into space.
Some advantages are:

- **Testing**: The general cost of a space mission involving satellites is huge, for that reason during the development we want everything to work in the correct way. However, on Earth, we cannot properly test satellite missions as functioning in their target environment, i.e., space. This is where simulators can help.
  By simulating all the relevant aspects of the space mission, we can test whether it will be successful or not. This is crucial for fixing development or design errors before actually sending the satellite into space, where we cannot fix it. For instance, these aspects can regard communication protocols, propulsion systems, and power management. Another important part covered by these computational tools is the simulation of unexpected behaviors, such as anomalies caused by space radiation, which are difficult to physically test.

- **Lower development cost**: When dealing with satellites, everything is expensive, starting from the physical launch of the satellite to reach the orbit, till the physical construction of a satellite. Even when we talk about nano-satellites, where individual components have a relatively lower cost compared to satellites, the cumulative cost escalates when we deal with a constellation of nano-satellites. Thanks to simulators, we are able to lower the costs, by experimenting with mission parameters, subsystems, and software, without building physical prototypes, which would increase the total mission cost.

- **Performance evaluation**: We can use simulators to test the working mechanism of nano-satellites and their communication protocols before actually being deployed in a space mission. An important step in doing the performance evaluation is to assess the accuracy of the simulation compared to the real context we are simulating.

This is also called validation and verification, and in this step, we ensure that the real world's output aligns with the simulator's output.

Along with these benefits, the use of a simulation has also some drawbacks:

- **Limited real-world accuracy**: The simulation technologies aim at modeling the real world. Unluckily the real world is full of physical complexities that sometimes are challenging to model with just a mathematical equation. Most of the time, unpredictable events, such as a SEEs in the context of space radiation, cannot be entirely modeled; for that reason, simulators often rely on assumptions and approximations.
- **The butterfly effect**: The butterfly effect is an allegory used in the chaos theory, that suggests that a small perturbation in the initial conditions can influence a wider complex system [13]. In the simulation's context, even a small variation in the inputs given to the system can modify the simulation's result. For that reason, the process of tuning the inputs given to the simulation is a really complex task.
- **Resource Intensive**: Simulations are made of complex mathematical algorithms and involve the modeling of numerous interactions between different variables. For this reason, running a simulation can require a significant amount of computational resources and, therefore a significant cost.

In the following, we describe some generic simulators that we analyze in the initial stages of finding a simulator to use as a starting point, discussing also the reasons why we have not used them.

3.2.1. SimPy

SimPy, also called "Simulation in Python", is a versatile simulation framework built on Python. SimPy is developed to be a process-based Discrete-Event Simulation (DES) [9]. A DES is a type of simulation where we decompose the target of our simulation in different components, which progress without interaction between them. Instead, the process-based approach is a particular approach of a DES, and it involves defining individual entities, also called processes, and describing how they behave over time [80].
Thanks to SimPy's generality, we are able to create simulations that adapt to a wide spectrum of scenarios, going from the flow of traffic in a city to the simulation of the orbit of a constellation of nano-satellites. We think to use SimPy, with sgp4, which is a library that calculates the position and speed of a satellite that orbits around the Earth [10].
In conclusion, we are discarding this option because we have opted to use as a basis a

simulator that is already satellite-oriented.

3.2.2. Network Simulator 3 (NS-3)

NS-3 is an open-source DES for networks, designed to assist in the analysis of various networking protocols and scenarios; targeted primarily for research or educational use. This simulator is widely used in networking to simulate and analyze the behavior of different networking protocols or scenarios in a virtual environment.
The simulation models of NS-3 are designed to achieve a high realism that enables it to work as a real-time network simulator, which can be integrated in real-world scenarios [71]. Therefore, thanks to this real-time feature, NS-3 can be used as an HIL simulation, which is a fundamental property for the development of our simulator. However, since we are not interested in network simulators, we discard NS-3.

3.2.3. OMNeT++

OMNeT++ is a simulation library built on C++, designed to construct network simulators. Since this is a general framework, with the term "network" we do not refer to a satellite network, but everything associated with communication between two entities. Therefore, for instance, we can simulate wired and wireless communication, Internet protocols, but also communication between satellites.
The strength of OMNeT++ lies in the fact that it employs a component-based architecture, that promotes model reusability. These components are written in C++ and then assembled using a high-level language called NED [14].
OMNeT++ is a framework commonly used by scientific and industrial communities; indeed, on its official site, we can find a list of 100+ selected simulators using OMNeT++. Among these simulators, we can find OS^3, a simulator described in Section 3.2.4. Finally, we have not selected this framework as a basis because we are not interested in a network-oriented simulator.

3.2.4. OS^3

OS^3 is an open-source simulator that is oriented to simulate satellites. OS^3 is a framework built on the OMNeT++ environment, which we describe in Section 3.2.3, and we can use it to simulate different types of satellite communications.
An interesting feature of OS^3 is the ability to automatically incorporate the orbit of the satellite we are interested in, along with the weather conditions on this trajectory. In this way, the simulator is able to simulate the communication of a satellite in the past and in

the future [18].
In conclusion, even if OS^3 is a satellite-oriented simulator we discard this option because it is oriented to the communication of satellites.

3.2.5. Hypatia

Hypatia is a framework that simulates nano-satellite constellations in LEO. Hypatia pre-calculates the state of the network and simulates the transmission of a packet exploiting NS-3, which we describe in Section 3.2.2, also providing visual results to be further analyzed.
Hypatia also considers satellite trajectories and connectivity coverage constraints to make its network calculations. Additionally, we can also use Hypatia to evaluate satellite orbit design, inter-satellite topology, and routing along satellites.
In conclusion, since our simulator is not oriented to networking, we discard this option [58].

3.2.6. OpenSand

OpenSAND, also known as Platine, is a user-oriented powerful tool that emulates satellite communication systems, providing tools to perform analysis on the data retrieved during the simulation. An interesting feature provided by this simulator is the possibility of interconnecting real equipment with real applications, enabling the possibility of simulating with real components [8]. However, since we are not interested in network simulation we decided not to use OpenSAND as a basis for our simulator.

3.3. Nano-satellite Simulators

In the following, we describe some simulators that we take under consideration as a starting basis for the development of our simulator. These simulators are all oriented to simulate the orbit of the satellites, which is the property we are interested in, along with other properties described in Section 3.1.
In the following sections, we describe the functioning of some satellite-oriented simulators that we take under consideration, leaving the comparison between them and our final decision in Section 3.3.4.

3.3.1. NASA General Mission Analysis Tool (GMAT)

GMAT is an open-source simulator developed by NASA that is tailored to mission design, trajectory optimization, and navigation in space. It is used in private industry, but also

at the academic level. GMAT can simulate space missions, going from LEO to lunar exploration, interplanetary journeys, and even deep space missions.
We can interact with GMAT in two different ways: by using an ad-hoc Graphical User Interface (GUI) or by writing MATLAB scripts. The two interfaces have the same expressive power, which makes development on GMAT flexible and suitable for both experienced and inexperienced users. When we create a mission in GMAT we start by selecting the resources needed, in this way we can also see the cost of the mission and modify the resource to adjust the cost to the desired one. Once the mission starts, GMAT displays the trajectories of the spacecraft of our mission and collects data to be later analyzed. Exploiting the GUI we can also change the coordinate system or focus on specific objects [5].

3.3.2. STK

STK is a physics-based software developed by Analytical Graphics, Incorporated (AGI), which enables simulation and analysis in multiple domains, i.e., land, sea, air, and space. The space domain, which is what we are interested in, has different additional software available, which handles different aspects of the simulation:

- **Communication**: we can simulate communications between satellites and communication of the satellites with the ground stations [1].
- **Multidomain**: we can simulate missions that have multiple domains, for instance, interconnecting a satellite mission with a ground station to receive data and then use this data for a mission in the sea domain [2].

STK provides the user with real-time visualization of the whole mission, both in 2D and 3D, including the trajectories and the satellites.
Every design of a space mission in STK is called a scenario and only one scenario at a time can exist. In a scenario, we can define the number of satellites, the communication systems, the analysis period, and the units of measure used for the whole system.
Inside the scenario, we can insert the objects used by the simulation. The objects are the central components of the simulation, for instance, a satellite is considered as an object. Each object inherits the properties of the scenario. However, we can later modify these properties at our pleasure [79]. To guarantee more flexibility, STK provides the ability to control the simulation from an external application, through the use of a scripting interface or through a Component Object Model (COM). COM is a library that contains types, interfaces, and classes that represent different aspects of STK [79]. The use of a scripting interface enables STK to be controlled by an external script; the most supported

programming languages are MATLAB and Python.
A flow of a general simulation, also exploiting external tools, is shown in Figure 3.1.

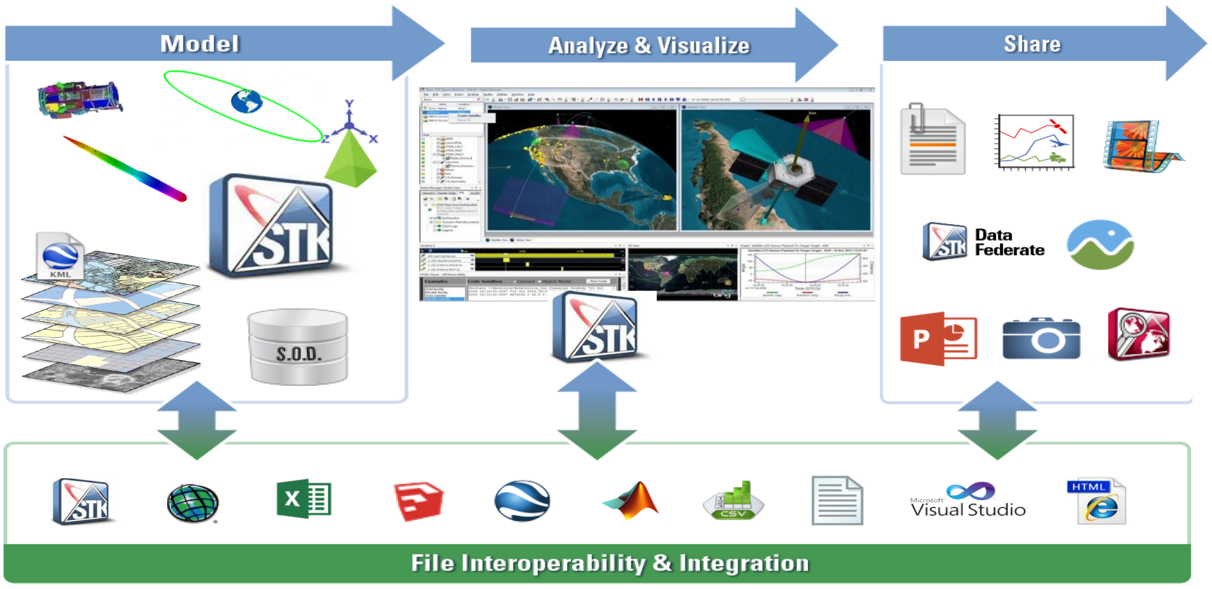


Figure 3.1: STK flow [75].

3.3.3. *cote*

cote is a framework composed of two different parts, the former is the first Orbital Edge Computing (OEC) simulator called *cote-sim*, and the latter is a run-time service called *cote-lib*, which enables OEC on a real nano-satellite constellation. In our case, we are not interested in a real-time service for nano-satellites, so we are not considering *cote-lib*.
cote-sim provides a detailed, physical simulation exploiting the OEC design. The OEC architecture enables edge computing capabilities for each nano-satellite, allowing local processing when downlink to a ground station is not possible. In the OEC design, the nano-satellite constellations are arranged into pipelines, to tackle delay in processing. These pipelines parallelize the processing and collection of the data only based on their geographic location [41].
cote-sim is divided into two separate simulations, one more oriented to orbital position, exploiting the OEC architecture, and the other more prone to communication, using the classic bent-pipe design, where a satellite has to communicate with the ground station to receive instruction and provide corresponding responses [41].
In the following, we describe the two different implementations:

- **Orbital simulation**: In this simulation, we are going to simulate a nano-satellite constellation orbiting in a predefined trajectory. While orbiting, every nano-satellite

will execute different jobs in predefined positions along the orbit. In this way, *cote* simulates the processing happening on a nano-satellite. Specifically *cote-sim* simulates the acquisition and processing of Earth's images, dividing them along the orbit into a predefined number of Ground Track Frame, which we call jobs. The processing of a job will discharge the nano-satellite by a predefined amount, but continuing along its trajectory the nano-satellite recharges thanks to the energy harvesting system, simulated by solar cells.
This simulation can be executed in four different ways controlling two parameters, the difference between these parameters is also shown in Figure 3.2:

  - **Frame-spaced vs. Close-spaced**: This parameter determines the physical configuration of the satellites [41]. Using frame-spaced the nano-satellites start at a different time, instead when we use close-spaced the nano-satellites start at the same time. As we can see in Figure 3.2, with frame-spaced, the nano-satellites acquire from a different frame of the ground simultaneously, instead with closed-spaced, the nano-satellites acquire from the same frame of the ground.

  - **Tile-parallel vs. Frame-parallel**: This parameter determines the workload of every nano-satellite [41]. Using tile-parallel the nano-satellites divide the workload of processing a frame; instead, using frame-parallel the nano-satellites process a whole frame every time. As we can see in Figure 3.2, with tile-parallel the nano-satellites acquire from just a fraction of the frame of the ground, instead with frame-parallel, the nano-satellites acquire from the complete frame of the ground.

  We can see the general flow of the whole application in Figure 3.3. The flow works in the following way.
  First of all, the simulation script wants as input the depth of the pipeline, which represents the number of nano-satellites inside the simulated constellation. Then, the simulation sets up different parameters and models the nano-satellite constellation. At this point, we reach the real simulation core: if a satellite is in the same orbital position as a job, it will execute the corresponding job, otherwise, its simulation time is updated, along with its orbital position. When every nano-satellite completes all its jobs or reaches the end of the orbit, the simulation is concluded.
  At the end of the simulation, *cote* processes the retrieved logs to create different plots for the end user.

- **Communication simulation**: In this simulation, we are going to simulate the

orbit of a nano-satellite constellation, along with the communication between every nano-satellite in the constellation and some predefined ground stations. The flow of the whole application works in this way.
Before executing the actual simulation script, we need to set up the configuration files for the different ground stations, the orbit to use, the date-time to use, and the sensor specifications. At this point, we can start the simulation script. First, the simulation sets different parameters, retrieved from the configuration files, and models the ground stations and the nano-satellite. In this simulation, the processing and the energy harvesting of a nano-satellite are not considered. Therefore, the simulation models only the transfer channel and the receiver channel of the nano-satellites, along with a sensor to simulate the data transferred to the ground station. Then, the simulation's loop starts. In the loop, the ground station checks if there are any available satellites within its range to connect to. If there are no available nano-satellites, just the simulation time is updated, otherwise, if the ground station is free, and there is an available nano-satellite, the following steps occur. The downlink channel from the nano-satellite to the ground station is created, the ground station retrieves data from the nano-satellite, the uplink channel from the ground station to the nano-satellite is created, the ground station sends instructions to the nano-satellite, and in the end, the simulation time is updated. The simulation ends when the simulation time reaches the predefined amount of time to simulate.
We are not interested in a communication simulation for our system, for the reasons explained in Section 3.1.

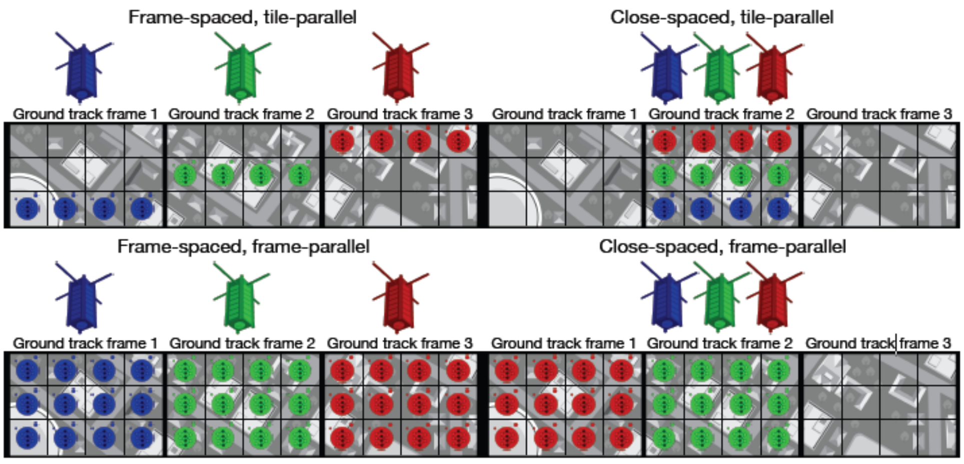

Frame-spaced, tile-parallel
Close-spaced, tile-parallel
Ground track frame 1
Ground track frame 2
Ground track frame 3
Frame-spaced, frame-parallel
Close-spaced, frame-parallel

Figure 3.2: Visual explanation of the parameters of *cote-sim* [41].

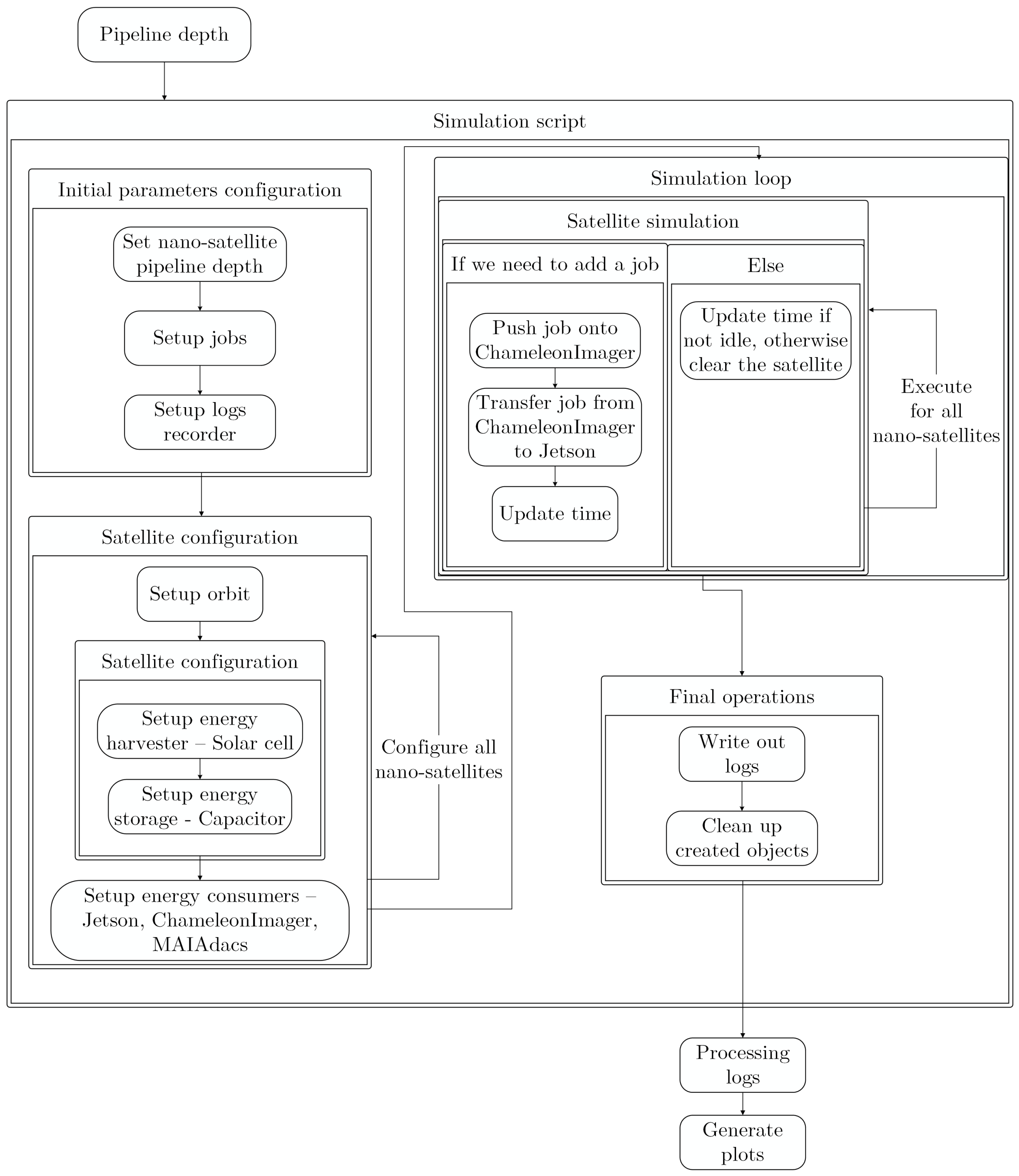

Pipeline depth
Simulation script
Initial parameters configuration
Set nano-satellite pipeline depth
Setup jobs
Setup logs recorder
Satellite configuration
Setup orbit
Satellite configuration
Setup energy harvester – Solar cell
Setup energy storage - Capacitor
Setup energy consumers – Jetson, ChameleonImager, MAIAdacs
Configure all nano-satellites
Simulation loop
Satellite simulation
If we need to add a job
Push job onto ChameleonImager
Transfer job from ChameleonImager to Jetson
Update time
Else
Update time if not idle, otherwise clear the satellite
Execute for all nano-satellites
Final operations
Write out logs
Clean up created objects
Processing logs
Generate plots

Figure 3.3: Flow of *cote*.

3.3.4. Comparison

	Property	**GMAT**	**STK**	***cote***
1	Energy harvesting support	✗	✓	✓
2	Energy consuming support	✗	✓	✓
3	Support processing of programs	✗	✗	✓
4	Support the development of a HIL simulation	✓	✓	✓
5	Real-time 3D and 2D visualization of satellite orbit	✓	✓	✗
6	Analysis of data retrieved	✓	✓	✓
7	CubeSats support	✓	✓	✓
8	Bent-pipe design	✓	✓	✗
9	Orbital Edge Computing (OEC) design	✗	✗	✓
10	Interaction between energy-constraint, intermittent computing, computational nanosatellite pipeline, data imagery collection and communication [41]	✗	✗	✓

Table 3.1: A comparison of the possible technologies to use as a basis for our simulator.

We deeply analyze the three softwares, and we list the appealing properties for our simulator that these softwares have in Table 3.1. As we can see in Table 3.1, both *cote* and STK take into consideration energy harvesting and energy consumption [41][59][48], but only *cote* supports the processing of programs based on orbit location; instead GMAT, does not support any of the three previous features. All simulators support the analysis of the data retrieved and support the modeling of CubeSats. All the simulation technologies can handle the development of a HIL simulation, because GMAT and STK have a programming interface to control the object of the simulation, instead *cote* does not have a GUI and it is open-source so we can directly modify the application code. Regarding the design, GMAT and STK have a bent-pipe design, which is the standard design for satellites; instead, *cote* provides a OEC design, which is more appropriate to simulate nano-satellite constellations.
In conclusion, the GUI that GMAT and STK provide is not a relevant property for our

simulator; instead the simulated processing and the OEC architecture, granted by *cote*, are appealing features for our simulator. For the previous reasons, we opt for *cote* to use as a basis for our simulator.

3.4. Hardware-In-The-Loop Simulation

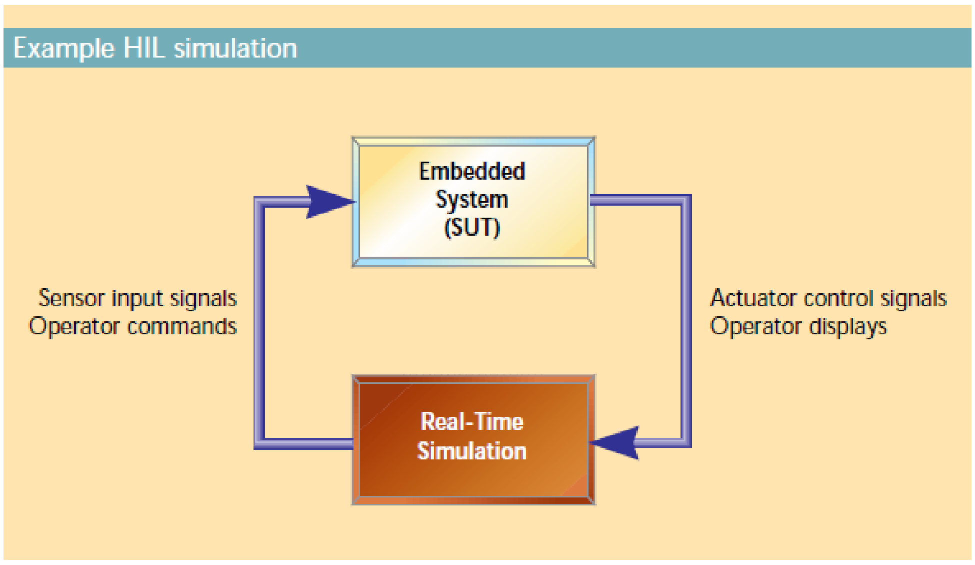


Figure 3.4: HIL simulation of an actuator [62].

The hardware and software complexity in embedded systems keeps increasing, leading to higher costs and effort in system-level design. Since many safety-critical systems rely on embedded devices, the need for deep testing becomes a crucial problem [62].
This is where HIL simulation comes in handy. HIL simulation is a cost-effective and repeatable technique generally used to test embedded systems at the system level. The idea behind this type of simulation is to include the physical device, which is also called System Under Test (SUT) in this context, inside the simulation, instead of emulating the device with a mathematical model [62][32].
However, some parts of the SUT, could be modeled in the simulation if the environment does not permit correct testing. An example of this is the testing of solar panels in a satellite context. In this specific case, physically reproducing the intricate energy harvesting and recharging properties of the solar panel in the space environment is a hard challenge. For this reason, a good option is to simulate that part with a mathematical model.
The simulation software in a HIL simulation must be real-time because we have a continuous communication and interchange of data between the SUT and the simulation [62]. We can see a simple scheme of this interchange of data in Figure 3.4. In this example,

the real-time simulation requests the SUT to perform some operations, and the latter executes the requested commands and replies with the data retrieved.
There are many benefits of using a HIL simulation:

- **Realistic testing**: The ability to test with real hardware helps reach more realistic testing compared to a mathematical model.
- **Flexibility**: HIL simulations are highly flexible and can be used in a wide range of scenarios. Some examples are aerospace and avionics, automotive, satellites, robotics, and medical devices.
- **Real-time interaction**: Thanks to its real-time feature, in a HIL simulation we can test what happens in real time to the SUT when it executes specific types of operation given by the simulation.
- **Debugging and optimization**: With a HIL simulation we can identify and debug hardware or software issues, which are not identifiable in a common simulation by using a mathematical model to emulate the whole SUT. In the following, we analyze how this feature is useful for hardware and software debugging and optimization:
  - **Hardware debugging**: In a HIL simulation, we simulate using the hardware components of the SUT. Thanks to this characteristic we can detect problems in the hardware, such as component failures, or electrical faults, that we cannot discover using a mathematical model.
  - **Software debugging**: In a HIL simulation, we simulate exploiting the software components of the SUT, including firmware and software interfaces. In this way, we can monitor how the software behaves when interfacing with the hardware components and with the instructions given by the simulation. Thanks to this characteristic we can discover bugs in the software, or unexpected behaviors, undetectable by using a mathematical model.
  - **Optimization**: In addition to debugging, we are able to optimize our SUT by exploiting HIL simulations. By experimenting with different hardware and software configurations, we can monitor how the SUT behaves and in doing so we can optimize it in a controlled environment.
- **Physical reproducibility**: By using just mathematical models in a simulation, we reproduce the same exact results. Instead, with a HIL simulation, we reproduce nearly the same results, taking into account small variations caused by performing operations with a real device and not with a mathematical model.

- **Environmental testing**: Thanks to the simulated environment we can test the real device in a wide range of environmental conditions, recreated by mathematical models. Furthermore, since the SUT is a physical device, if we can recreate the specific physical environment of our interest, we can simulate the SUT behavior within its physical environment without relying on mathematical equations. This last feature enables us to have a more realistic simulation approach.
- **Data Collection**: HIL simulations provide critical data for analysis, and to understand how a real device behaves in the simulation context. By exploiting these data, we can monitor how a real physical device behaves when integrated into a simulated environment.

3.5. CubeSatSim

The Radio Amateur Satellite (AMSAT) Corporation, which is an organization composed of Amateur Radio Operators spread across the world [60], observed that a lot of CubeSat are Dead On Arrival (DOA) when are deployed into an orbit. This is a pity, considering that the amount of resources and time used for a single mission can be very high, as we discuss in Section 2.1. For this reason, the AMSAT starts the development of the CubeSatSim, as a tool for educational purposes [54].
The CubeSatSim, shown in Figure 3.5, is a 1U CubeSat composed of a Raspberry PI, which we describe in Section 3.5.1, multiple energy harvesters, different sensor boards that retrieve real data from the environment and a transmission board along with the telemetry mode supported. All these components are described in Section 3.5.2.
The CubeSatSim can communicate with an ad-hoc Ground Station, which is developed by AMSAT. The requirement of the Ground Station is to receive the multiple types of telemetry that the CubeSatSim can send, decode them, and visualize the information retrieved [55]. Finally, the frame of the CubeSatSim is composed of 3D-printed 1U CubeSat parts.

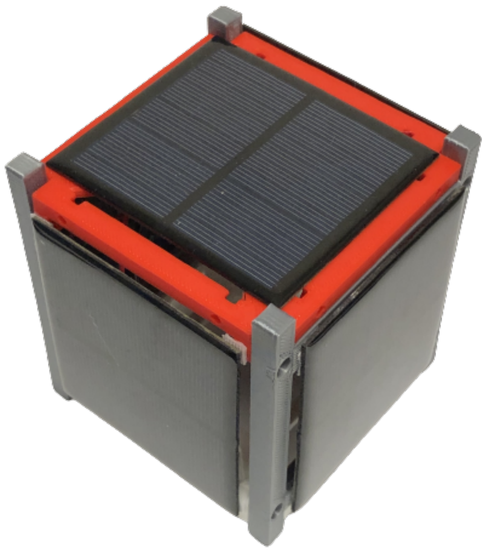


Figure 3.5: CubeSatSim structure [37].

3.5.1. Computing Unit

The CubeSatSim software is tested on the following Raspberry PIs: Pi Zero, Pi Zero W, Pi Zero 2 W, Pi 3B+, Pi 3B, and Pi 4B [36]. However, for the development and testing of our simulator, we only used the Raspberry PI Zero W and the Raspberry PI Zero 2 W. The use of a Raspberry PI reflects a Commercial Off-The-Shelf (COTS) approach. The design choice of using a Raspberry PI makes our nano-satellite a realistic representation of an actual system in space [78]. Both Raspberry PIs are a more compact, low-power version of the traditional Raspberry PI, enabling both of them to be used inside the CubeSatSim. To connect to the Raspberry PI without the need to plug a keyboard and a monitor directly into it, these versions of the Raspberry PI enable the connection to a PC exploiting the Ethernet over USB service, also known as RNDIS on Windows and USB Gadget on Linux [39]. In the process of developing our simulator, we use this functionality for managing purposes, such as managing the files on the satellite, and for enabling internet connectivity on the nano-satellite.
We can see the design of how the raspberry PI is interfaced with the other components through different busses in Figure 3.6. The CubeSatSim project, exploits the General-Purpose Input Output (GPIO) to connect the Raspberry PI with the other boards mounted. The GPIO contains two busses that are used to communicate with the different devices: the Universal Asynchronous Receiver/Transmitter (UART) bus and the I2C bus [54].

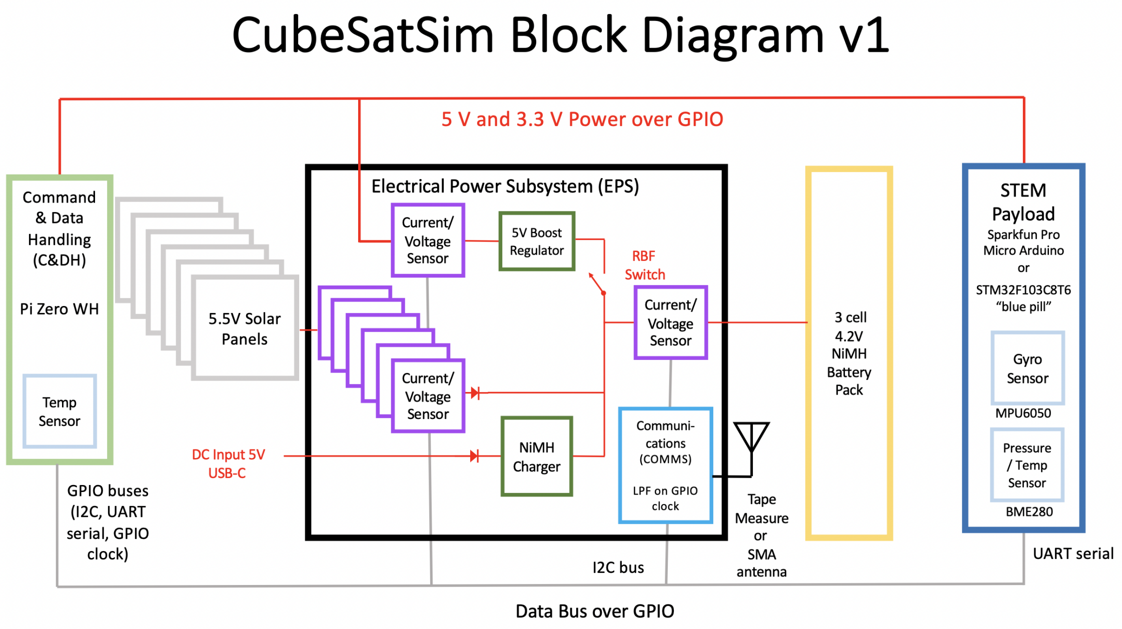


Figure 3.6: CubeSatSim block diagram [37].

3.5.2. CubeSatSim Components

Apart from Raspberry PIs, the CubeSatSim is composed of other many components that guarantee the ability to recharge itself with an energy harvesting system, the ability to retrieve data from many types of sensors, and the ability to transfer the telemetry with different modulation techniques.

The CubeSatSim features a battery board composed of either three AAA or three AA rechargeable batteries. These batteries can be charged with a standard micro-USB cable or through the six solar panels mounted on the CubeSatSim [29].

Furthermore, the board retrieving data from the environment is called the STEM Payload Board and it is an STM32F103C8T6 microcontroller equipped with a 32-bit ARM processor. We can see from Figure 3.6 that the Raspberry PI receives all the data retrieved from the STEM Payload Board exploiting the UART serial. The STEM Payload Board is equipped with a BME280 sensor for measuring temperature, pressure, altitude, and humidity. Additionally, this board has a GY-521 module, referred to as an Inertial Measurement Unit (IMU), which comprises a magnetometer, an accelerometer, and a gyroscope. This is not the only source of information of the CubeSatSim since it also receives data from the EPS, through the I2C bus, as we can see in Figure 3.6. From the EPS, the CubeSatSim can retrieve the current and the voltage of the following elements: the Power Supply Unit (PSU), the battery, and the solar panels. These data are retrieved by eight different INA219 current and voltage sense boards [29]. Once collected, all these

data are inserted into a payload, which the CubeSatSim transmits through telemetry.
The CubeSatSim is able to transmit telemetry through the use of a Transmitter/Filter Board (TFB). The TFB enables the CubeSatSim to transmit in different types of modulation schemes:

- **Automatic Packet Reporting System (APRS)/Audio frequency-shift keying (AFSK)**: AFSK is a modulation technique used on signal, which translates audio tones to digital bits. Lower-frequency tones translate into the bit '0', while high-frequency tones become the bit '1'. This modulation is frequently used within the Radio Amateur community and in communication applications like APRS [6].

- **FSK/DUV**: FSK is a modulation technique that translates a discrete analog signal into a digital one. Using the FSK modulation technique we translate 0 and 1 according to waves at specific frequencies. Those frequencies should not be too similar, otherwise, we encounter errors in understanding what becomes a zero and what becomes one, but we cannot have frequencies too far apart, because we create a lower throughput [46].
  The CubeSatSim uses FSK at 200 bps. In this case, the data is transmitted whenever the transmitter is active. The modulation technique of FSK at 200 bps is also referred to as DUV [31].
  We can see the description of the parameter of the encoded FSK message, shown in Appendix A, in Table A.1 and Table A.2.

- **Binary Phase Shift Keying (BPSK)**: In BPSK the modulation occurs when we alter the carrier signal, which is the underlying waveform that undergoes modulation. Specifically, we alter the carrier signal's phase by exactly 180° for each variation of a binary symbol. Therefore, when we want to transmit a 0, we shift the phase by 180°, otherwise, when we convey a 1, we do not alter the phase of the carrier signal [66].
  The payload of the BPSK messages is the same as the FSK/DUV ones, except for some control parameters at the end of the message. When the CubeSatSim sends a BPSK message, it sends the data multiple times. This procedure is done to enable error correction of the payload since sometimes the message could be corrupted during the transfer.
  Additionally, when CubeSatSim transfers through BPSK, it sends two additional messages containing the payloads regarding the minimum and maximum values obtained by every sensor deployed on the CubeSatSim since the last reset.

- **Slow-Scan Television (SSTV)**: SSTV employs frequency modulation to transmit images as a signal. To achieve this with a black-and-white image, we associate the different levels of brightness in the image with audio frequencies, so when we change the frequency of the signal, it signifies a brighter or a darker pixel. Instead, if we want to transmit a color image, we send the Red Green Blue (RGB) color components separately using the same technique for each color [27].

- **Continuous Wave (CW) with Morse Code**: CW is a modulation technique that transmits a precise signal, by altering the amplitude, frequency, or phase of a continuous wave. Usually, the Morse Code transmission is used with the CW modulation, because just by switching on and off the carrier wave, we can represent the symbols of the Morse Code [45].

In summary, this chapter has provided insights into our focal applications, described in Section 3.1. We explained the importance of simulation technologies, especially when they integrate a physical device into their loop. Our examination covered various simulators, going from general-purpose to satellite-oriented ones. In particular, none of the simulators we were interested in, among those analyzed, integrated a physical device into their loop. This led to our decision to develop one such simulator. Additionally, we identified the lack of simulation of radiation-induced errors, primarily due to the absence of a physical device in their loops, which is a relevant aspect to better simulate SEEs. Consequently, we decided to include this feature in the simulator we developed. Moreover, we think that could be useful for a nano-satellite constellation to be simulated by multiple nano-satellites, therefore, we decided to parallelize the execution of our simulation. In Chapter 4 we analyze the challenges of implementing these aspects in our simulator.

4 | Problem Statement

In this chapter, we introduce the problem we target in this thesis. First, in Section 4.1 we describe the problems derived from the decision to enhance the reality of our HIL simulation. Furthermore, in Section 4.2 we analyze the reasoning behind our choice of guaranteeing high-configuration capabilities to the user. Lastly, in Section 4.3 we describe the reasoning behind the scalability applied to our system.

Problem statement:

Our main challenge comes from the need to provide a simulation of a constellation of nano-satellites, which enhances its realism by retrieving data directly from a real nano-satellite, simulating the user's desired space mission scenario, and also providing high-configuration capabilities to the end user. Our system must be scalable and able to interact with multiple physical nano-satellites simultaneously to decrease the execution time of the simulation.

4.1. Realism

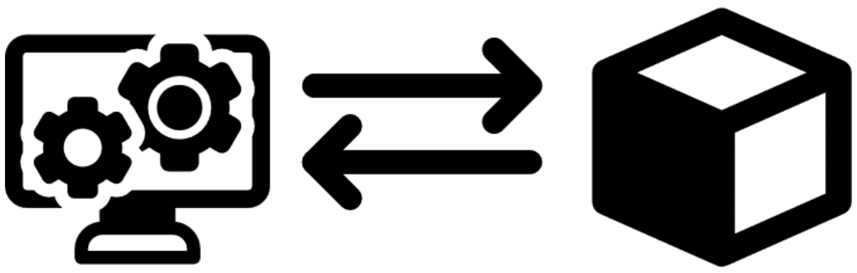


Figure 4.1: Flow of a HIL simulation.

Before deploying a space mission involving a constellation of nano-satellites, a deep planning and testing phase is conducted. In the planning phase, the objective of the space

mission is outlined. However, space simulators cannot test the behavior of the nano-satellites in the specific mission scenario. For instance, space simulators are unable to simulate the energy consumed by the nano-satellites performing Earth monitoring activities. This is due to a lack of real data retrieved from a real nano-satellite and the inability of the simulator to test the constellation of nano-satellites in the desired scenario.
The lack of real data leads to the development of artificial simulations, which cannot realistically predict the unpredictability of space and the devices involved in the simulated space mission. Therefore, we decided to retrieve the data used in the simulation from a real nano-satellite. This type of simulation, which is visually depicted in Figure 4.1, is referred to in the literature as Hardware-In-The-Loop (HIL) simulation.
Furthermore, every space mission has its own specific scenario. For instance, each Earth monitoring mission is unique, with its specific mission scenario, ranging from monitoring the wildfires [40] to observing plants and animals [65]. Simulating these scenarios is not trivial because it involves dealing with distinct characteristics, conditions, and variables associated with each space mission. By including the functionality of simulating the constellation of nano-satellites in the desired mission scenario of the user, we are able to provide further realism to our system.
The inclusion of a real nano-satellite in the simulation brings different challenges:

1. **Interface between the simulation and the physical nano-satellite**: Since we deal with two separate artefacts, we need a way to interconnect and enable communication between them, otherwise the simulation cannot function properly. For instance, as we can also see in Figure 4.1, the simulation must ask the nano-satellite for some information needed, and once the nano-satellite finishes processing the information requested, the nano-satellite needs a way to send back the results obtained. Furthermore, the communication must be reliable to guarantee that both artifacts receive what they need. Lastly, the communication must be platform-independent to provide the user with a higher level of customization. Guaranteeing all the previous properties underscores the difficulty in achieving an effective interface between the simulation and the physical nano-satellite.

2. **Coordination of the simulation with the nano-satellite**: This step is needed because when the simulation communicates with the nano-satellite, the latter must be aligned to provide the correct answer. Without ensuring correct coordination the simulations may lead to inaccurate conclusions about the behavior of the nano-satellites in the desired space mission scenario. For instance, if in our simulation we want to simulate an Earth monitoring scenario, but the physical nano-satellite is not coordinated, then, the nano-satellite might simulate a different scenario. This

is challenging because the mission scenario must be recreated for each execution of the simulation in a way that is consistent with both the simulation and the nano-satellite, ensuring also that the nano-satellite has received the mission scenario.

3. **Synchronization between the simulation and the nano-satellite**: These two artefacts operate at distinct temporal rates; the nano-satellite works in real time, whereas the simulation operates at a pace faster than real time. Therefore, we need a way for the simulation to work in conjunction with the nano-satellite, which is a challenging demand. If the simulation and the nano-satellite go out of sync two possible scenarios can happen: the simulation starts to ask for information from the physical nano-satellite while the latter is busy processing other information, or the nano-satellite sends back to the simulation results not expected.

4. **Retrieve real data from the nano-satellite**: Our system aims to simulate a space mission working on a desired mission scenario, for instance, Earth monitoring, or weather forecasting. By exploiting the physical nano-satellite integrated into the system, the simulation leverages real data to accurately replicate the behavior of the nano-satellite during the desired mission scenario. This step is required to enhance the realism of our system, avoiding relying on artificial data. However, the problem of using real data is not trivial, as it involves considerations related to the use of platform-dependent softwares, raising doubts about the fidelity of the data retrieved.

5. **Impact of space radiation**: Radiation in space is a serious problem for nano-satellites. When the particles of space radiation interact with an electronic component, this can lead to many different consequences, such as errors in the software, or even hardware malfunctioning. However, simulating these errors can be really challenging because of the dynamic and unpredictable nature of space radiation, but also for the variability in how radiation impacts the nano-satellite. Trying to further enhance the realism of our system, we focus on simulating the errors induced by radiation on the nano-satellite.

4.2. User Configuration

Space simulators must provide the end user with high-configuration capabilities. This necessity comes from the diversity that distinguishes space missions. Therefore, the user must have the flexibility to simulate his own space mission, retrieving results as accurately as possible. For instance, a user who wants to simulate a space mission with an Earth monitoring scenario has a completely different setup from the simulation of a space mission with an Internet-providing scenario.

The system must provide the user the ability to choose his desired mission scenario and to configure the system parameters according to his liking, ensuring the fact that the simulator can replicate the specifics of the user's space missions. However, this is a challenging problem because the system must be adaptable to all the modifications provided by the user while still guaranteeing steady functioning.

4.3. Scalability

Space missions involving a constellation of nano-satellites comprise the use of an ever-increasing number of nano-satellites. To give an example, the Starlink mission of SpaceX consists of a constellation of about ∼5,000 nano-satellites and aims in the near future to reach about 12,000 nano-satellites [67] [12]. Moreover, it is crucial to be able to realistically simulate an ever-increasing number of nano-satellites, but having a simulation that works with a single physical nano-satellite can be limiting in simulating a real-case scenario involving a constellation of nano-satellites. Furthermore, simulating constellations densely populated with nano-satellites involves a higher simulation time compared to simulating a constellation with fewer nano-satellites, because the higher number of nano-satellites requires more processing. Additionally, the simulation time is increased by the fact that having a single real nano-satellite can lead to a bottleneck in the system because the nano-satellite and the simulation work at different temporal times, as outlined in Section 4.1. To enhance the realism of the system and at the same time mitigate the increase in simulation time, caused by the growing number of nano-satellites to simulate and by the bottleneck of having only a physical nano-satellite, we focus on the aspect of providing more scalability to our system. However, this is not a trivial challenge because we have to optimize our HIL simulation, making it also more scalable while being careful at the same time not to decrease the overall realism provided by the simulation.

In summary, in this chapter, we described the reasoning behind our problem statement, without going into deep explanations about the definitive design and implementation. In the following chapters, we explore in more detail the design choices we made to address these challenges in Chapter 5, while in Chapter 6 and Chapter 7 we describe how our HIL simulation can be customized by the user and our final implementation.

5 | Design

In this chapter, we describe the design decisions we take. In Section 5.1 we define the components that simulate the behavior of our constellation of nano-satellites. Furthermore, in Section 5.2 we describe the coordination and synchronization phases needed in our system and the integration of the data received from the physical nano-satellite inside the simulation. In Section 5.3 we describe how we manage the errors induced by radiation. In Section 5.4 we describe the data retrieved from the physical nano-satellite, and how we design this data retrieval. In Section 5.5 we analyze how we decided to design the communication channels between the simulation and the physical nano-satellite. In Section 5.6 we describe our approach to provide more scalability to our system. These design choices are dictated by the fact that we wanted to generalize our system as much as possible in two ways:

- **Generality on the choice of the nano-satellite**: Every design choice we made during the integration of our simulation with the physical nano-satellite, which in our case is the CubeSatSim described in Section 3.5, was dictated by the fact that we want to make our system as generic as possible and not focused on developing a simulator centered on the CubeSatSim. Therefore, if a user wants to simulate a space mission with a specific nano-satellite, the user can decide to substitute the nano-satellite involved in our system, which is the CubeSatSim, in a few steps without reimplementing the whole system again. These steps are discussed in Section 6.3. The procedure of substituting the nano-satellite leaves multiple aspects of our system untouched. The aspects that remain untouched are pointed out in Section 6.3.

- **User configuration**: Since we want to let the user reconfigure our system in the way he wants, we decided to structure it to be configurable in many different ways. The user can decide to simulate a specific space mission he desires. For instance, he can decide to simulate a space mission regarding the Earth monitoring focus on the monitoring of wildfires. Furthermore, the user can configure parameters regarding the simulation, the physical nano-satellite, and the communication channel. We analyze these configurations in Chapter 6.

5.1. Jobs and Tasks

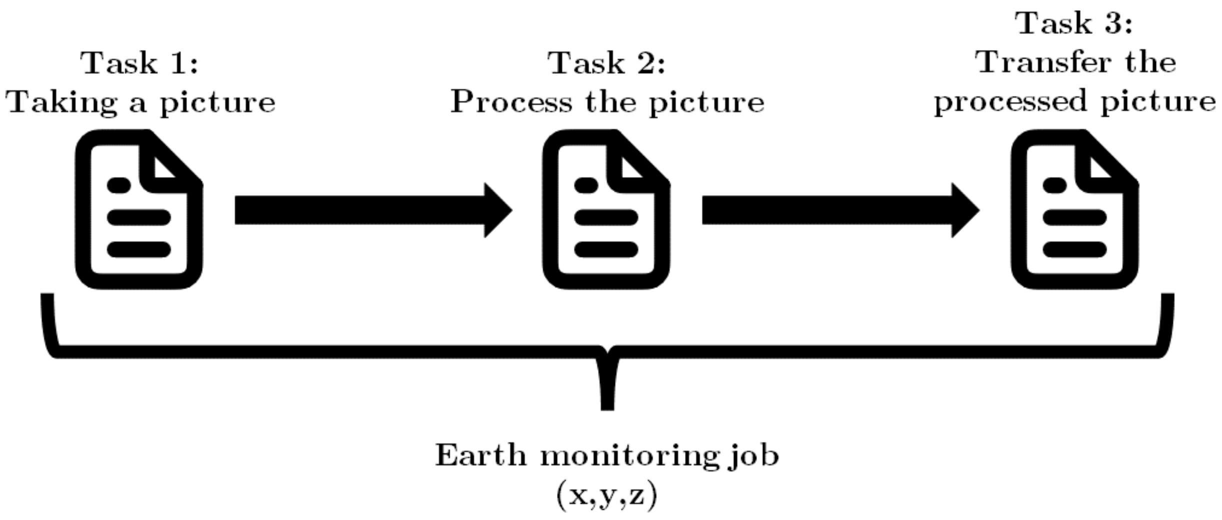


Figure 5.1: Tasks and jobs.

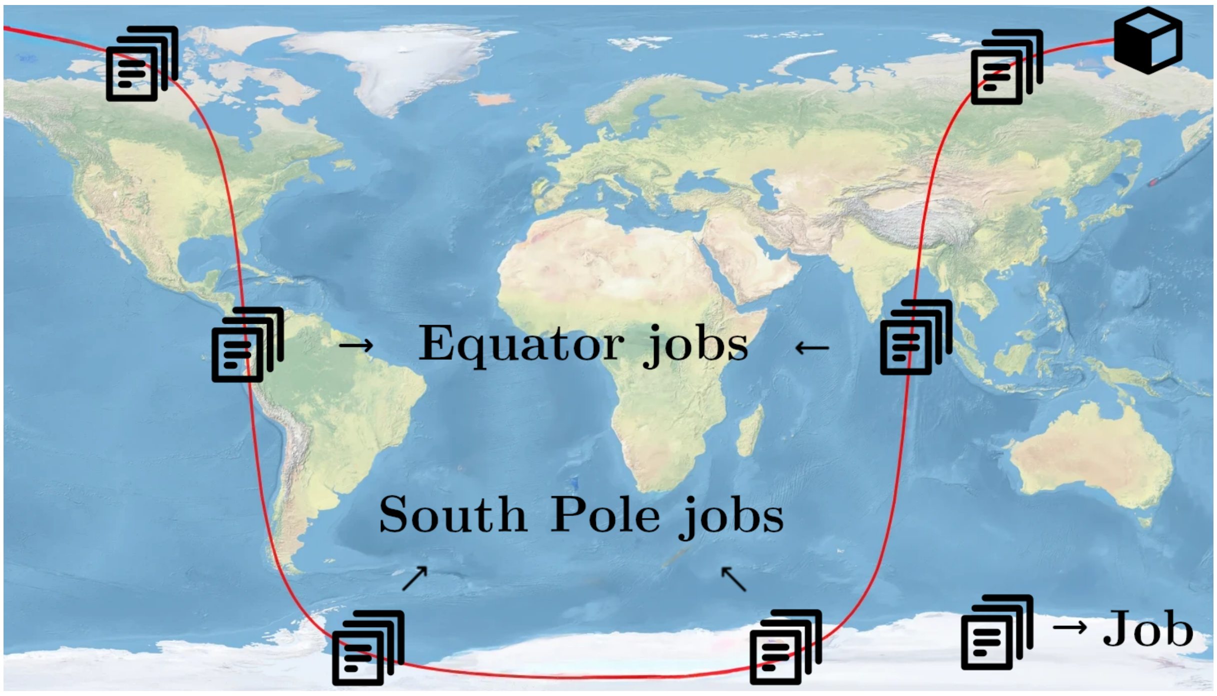


Figure 5.2: Tasks and jobs in orbit.

To enhance the realism provided in our HIL simulation, we let the user define his desired space mission scenario, computing how each nano-satellite in the constellation behaves while processing the objectives of the desired scenario. For instance, the user has the possibility to simulate a constellation of nano-satellites that has an Earth monitoring scenario as a space mission. The components that fulfill this assignment are the jobs and their sub-components, i.e. tasks, which are visually shown in Figure 5.1. A job is the objective that a nano-satellite in a space mission needs to fulfill in a position along the orbit. The subdivision of the jobs into different tasks is useful for structuring the different activities executed by a job. A task is a real program that is executed by the physical nano-satellite, and one task or the consecutive execution of multiple tasks forms a job. The ordering of how the tasks are executed is defined by the utilization of two mechanisms: the specification of dependencies between tasks and the application of a priority policy. The output of a task could be used as input for the following task of the job, forming a chain of tasks, as we can see in the example in Figure 5.1. For instance, in Figure 5.1 we can see a possible structure of the tasks of an Earth monitoring job. In this example, there are three tasks for the job that simulate the objective of an Earth monitoring scenario, which are: taking a picture, processing the picture in the desired way, and transferring the processed picture.

An important aspect of every task is that its simulation involves the retrieval of the data recorded on the physical nano-satellite during the execution time of the task. A subset of these data is composed of non-functional data, which are the data we use in the context of our thesis. These data represent the voltage and current retrieved from the PSU, the battery, and the solar panels of the physical nano-satellite. Specifically, we compute the energy consumption of the nano-satellite retrieving the data from the PSU. We do not use the battery data because in our configuration the nano-satellite is alimented by a mobile phone charger, therefore these data are not realistic. The explanation of why we do not use the solar panel data is available in Section 7.1.2.

As we can see in Figure 5.2, the simulation is structured to place the different jobs on different orbital positions. The structure of the placement of the various jobs along the orbit is already provided by *cote*. A possible example is shown in Figure 5.2, where the icon of the cube represents the nano-satellite, while the other icons represent the various jobs distributed along the orbit of the nano-satellite. This structure of simulating the different jobs along the orbit of a nano-satellite enables the simulation to simulate the same or multiple different objectives in the various orbital positions. For instance, a nano-satellite can simulate a job regarding the monitoring of the temperature when the orbital position is close to the Equator, such as the second job in Figure 5.2, and when the orbital position is close to the south pole, such as the third job in Figure 5.2, it can simulate a job regarding the observing of the ice melting.

5.2. Simulation

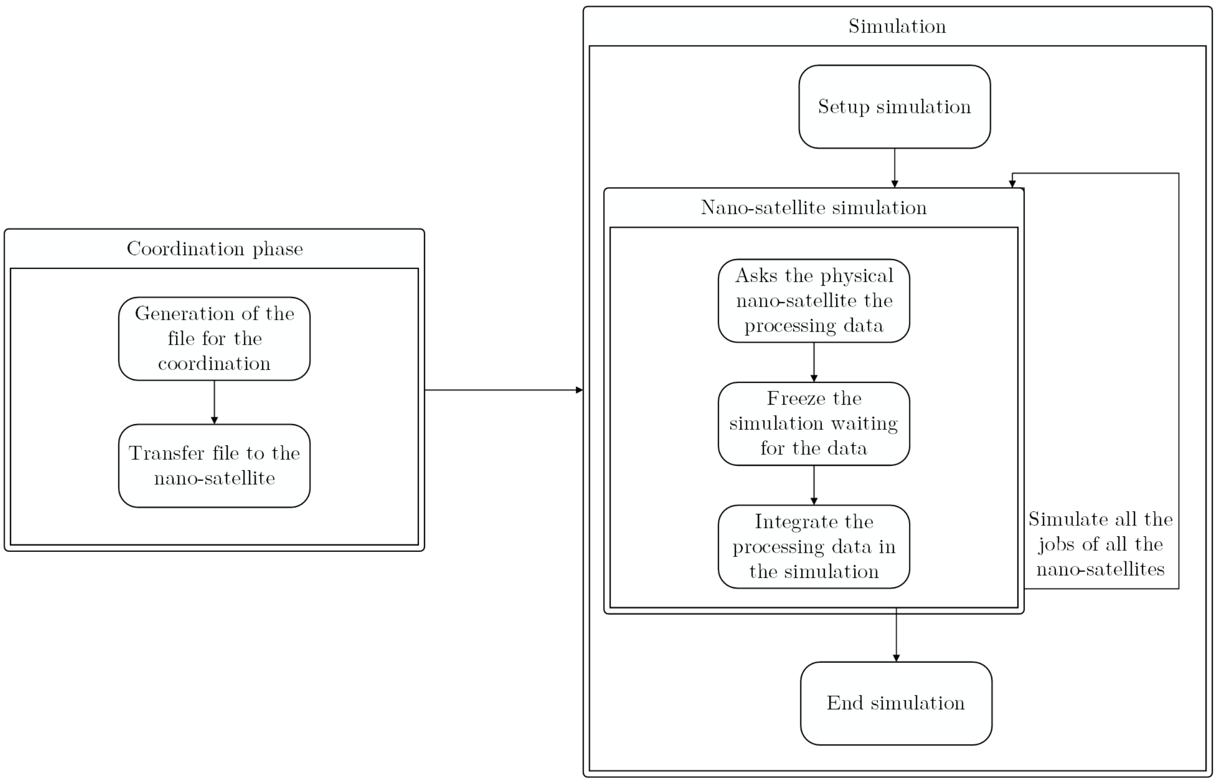

Coordination phase
Generation of the file for the coordination
Transfer file to the nano-satellite
Simulation
Setup simulation
Nano-satellite simulation
Asks the physical nano-satellite the processing data
Freeze the simulation waiting for the data
Integrate the processing data in the simulation
Simulate all the jobs of all the nano-satellites
End simulation

Figure 5.3: Design of HIL simulation.

We redesigned the *cote* simulator to convert it into a HIL simulation, but we decided to leave unaltered some characteristics of the design of the *cote* simulator, such as the orbit calculation, and the modeling of the nano-satellite components. Examples of these components include the solar panels and the energy consumer component, which is the collective representation of the overall energy consumption of the nano-satellite. The jobs and the tasks were components also of the *cote* simulator, but to align with our design we accordingly modify them, as described in Section 5.1.

In our simulation, we structure the nano-satellites in the constellation, which we refer to as "simulated nano-satellites", as they are the simulated counterparts of the physical nano-satellite. Therefore, each simulated nano-satellite must interact with a physical nano-satellite to be simulated. For instance, if we simulate a constellation of 10 simulated nano-satellites, every simulated nano-satellite must separately interact with a physical nano-satellite. In our new design, it is necessary to coordinate the simulation with the physical nano-satellite. This is crucial because every simulated nano-satellite has assigned one or multiple jobs. To achieve this, as we can see in the left part of Figure 5.3, a priori of the execution of the simulation, we generate and transfer to the nano-satellite a file that enables the real nano-satellite to coordinate with the simulation on the jobs to execute. Another important aspect to keep under consideration in our design is the synchronization between the simulation and the nano-satellite. These two components operate at distinct temporal rates; therefore, we need a way for the simulation to work in conjunction with the nano-satellite. Our approach, shown in the right part of Figure 5.3, is to temporarily freeze the simulation whenever a simulated nano-satellite requests the processing of a job, starting to wait for the real data from the physical nano-satellite. The data retrieved from the physical nano-satellite, are used by the simulation to correctly simulate the behavior of the simulated nano-satellite for the corresponding job. The pause on the simulation side, which we can see in the right part of Figure 5.3, enables the physical nano-satellite to execute the different tasks of the job requested and send to the simulation the real data retrieved. Once the simulation receives all the data for the simulated nano-satellite, the simulation is unfrozen and can proceed with its execution, as shown in the right part of Figure 5.3.

5.3. Radiation-induced Errors

Dealing with the simulation of radiation-induced error is a complex challenge because it involves accurately modeling the intricate interactions between radiation and the nano-satellite components. However, this obstacle is made manageable by exploiting the HIL simulation approach. In this context, we can take advantage of the physical nano-satellite integrated into the simulation to simulate the radiation-induced errors directly on the nano-satellite. We considered two possible design approaches for the simulation of these errors directly on the nano-satellite:

- **Inducing under-voltage situations**: This approach involves modifying the voltage levels to drop below the standard operating range. This approach would lead to a non-realistic simulation because, in a real scenario, the use of a voltage regulator can ensure a stable and consistent voltage supply.
- **Corrupting the tasks**: This approach involves the random corruption of the tasks. The corruption of a task is done before the physical nano-satellite executes the task. In this way, we simulate a radiation error occurring during the objective mission. For instance, in a job with an Earth monitoring scenario, if the task that is responsible for taking a picture of the Earth is corrupted, it can lead to an erroneous job, providing the simulation insights on how the physical nano-satellite involved in the system supports corruption errors.

Since the first approach is less realistic compared to the second one, and the second approach is more oriented to the focus of our system, we opted for the second approach.

5.4. Nano-satellite

The designs of how the real data of the physical nano-satellite is recorded, and how the physical nano-satellite manages the execution of the different tasks are crucial to enable the physical nano-satellite to be integrated in the HIL simulation.
The real data recorded on the nano-satellite are divided into two different subsets:

- **Non-functional data**: These data are the ones recorded from the EPS, shown in Figure 3.6. From the Electrical Power Subsystem (EPS), we can retrieve the current and the voltage of the following elements: the Power Supply Unit (PSU), the battery, and the solar panels. Additionally, we include in this set of data the temperature of the CPU to maintain the same structure of the data recorded by the CubeSatSim.
- **Application payload data**: These data are all other data that the CubeSatSim generates or calculates, mainly through the use of the STEM Payload, described in Section 3.5.2. Excluding the non-functional data described in the previous point, we can find the complete list of these data in the payload of a FSK message, defined in Section 3.5.2. The list of all these values is available in Table A.1 and Table A.2 of Appendix A.

The original design of the CubeSatSim does not include the storage of the real data on the physical nano-satellite. However, since we need the data retrieved from the physical nano-satellite for the whole duration of each task, we modified the design to support this. By recording these two subsets of data, the simulation is able to calculate the behavior of each simulated nano-satellite during the execution of the tasks. In the context of our HIL simulation, we use only the non-functional data to simulate the energy consumed by the simulated nano-satellites while they process the different jobs.
Lastly, we reason also about the fidelity of the data that we retrieve from the physical nano-satellite. Our approach is based on the reasoning that a greater amount of collected data contributes to a more faithful representation of the final result. This reasoning is based on the concept that by collecting more data in the same period of time, there is a higher probability of retrieving peaks unseen by collecting less data. Therefore, we focus on retrieving the highest possible number of data to obtain a more accurate final result.

5.5. Communication

In this section, we describe how we decided to design the communication between the simulation and the physical nano-satellite. In our system, we need two different communication channels at two different times. The first time we need a unidirectional channel before the beginning of the simulation; this channel is used to coordinate the simulation with the nano-satellite. The coordination phase is described in Section 5.2 and in Figure 5.3. The second time we need a bidirectional communication channel, which is used by the simulation to request the execution of the tasks of the different jobs and retrieve the real data from the physical nano-satellite in response.

Both communication channels need to be reliable. In the first case, without a reliable communication channel, we can encounter problems in the initial coordination between the simulation and the nano-satellite, which results in an erroneous simulation. This erroneous simulation is dictated by the fact that if the simulation and the physical nano-satellite are not coordinated on the jobs to execute, then, a simulated nano-satellite requests the execution of a job, and the physical nano-satellite executes a different job. In the latter case, without a reliable communication channel, the nano-satellite can lose the request to process a job from the simulation, or conversely, the simulation can lose the real data, sent by the physical nano-satellite, resulting from the execution of a task.

Lastly, in our approach, we establish communication channels that rely on platform-independent protocol. This choice was made focusing on the decision to provide high-configuration capabilities to our system.

5.6. Scalability

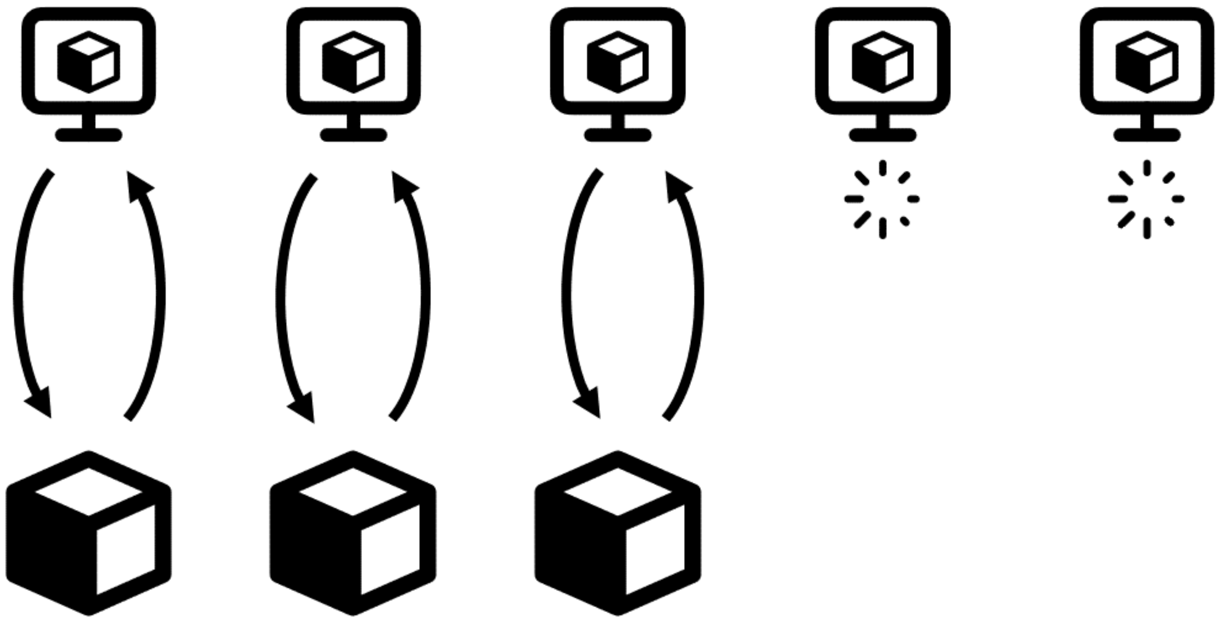


Figure 5.4: Scalability architecture.

Considering our focus on nano-satellite constellations, the use of a single physical nano-satellite can introduce bottlenecks in the system because when the simulation requests the execution of a job, the whole simulation needs to wait for the real data from the physical nano-satellite of all the tasks of the job to proceed. To enhance the scalability of our system, we can address both the nano-satellite side and the simulation side. In Figure 5.4, we can see the working mechanism of our scalability approach. In Figure 5.4, the elements at the top represent the simulated nano-satellites, while the elements at the bottom represent the physical nano-satellites. We assessed two possible design approaches to increase the scalability on the nano-satellite side of our system:

- **Emulating multiple devices**: This approach involves emulating multiple instances of a nano-satellite, which behaves the same way as the physical nano-satellite. This approach would lead to a decrease in the realism of the data used for our simulation because only a physical nano-satellite can provide real data to the simulation, while the emulating nano-satellites, even if they behave like the physical nano-satellite, still provide artificial data.
- **Using multiple physical devices**: This approach, visually shown at the bottom of Figure 5.4, involves the use of multiple physical nano-satellites, set in the same way as the already used physical nano-satellite. With this design choice, our system

would not lose realism because we use other physical nano-satellites, therefore we still provide real data to the simulation.

Since the first approach would decrease the realism of the data used by our simulation and the second approach maintain the same data realism, we opted for the second approach. In addition, our design choice focuses on addressing the problem of the bottlenecks in the system on the simulation side. Our approach, shown at the top of Figure 5.4, entails the use of parallel execution of the simulated nano-satellites. This approach combined with the use of multiple physical nano-satellites, enables the simulation to continue the computation of the simulated nano-satellites in case of a bottleneck of a simulated nano-satellite, as we can see in the center of Figure 5.4 where there are multiple connection from the various simulated nano-satellites to the various physical nano-satellites. For instance, if a simulated nano-satellite of the constellation has to process a long Earth-monitoring job, the other simulated nano-satellites are not forced to wait for the job execution to end, but they can continue with their own simulation. Our approach of parallel execution benefits also the problem of synchronization, discussed in Section 5.2, because even if a simulated nano-satellite is waiting for the real data from the physical nano-satellite, the rest of the simulation can proceed with its execution.

In conclusion, we modify the original design of the *cote* simulator to enable communication with a real device, integrating the data retrieved in the simulation. We also decided to design the different parts of our system to be as generic as possible and less focused on the nano-satellite we used. By doing so, we enable our system to adapt in case of modification of the physical nano-satellite involved. Furthermore, we describe the design choices done to increase the overall scalability of our system. In this chapter, we did not go in-depth on the discussion of the reconfiguration available to the user. We better analyze these configurations in Chapter 6.

6 | User Configuration

In this chapter, we analyze the types of configuration that the user can define to obtain the simulation he wants. In Section 6.1 we describe the parameters that the user can configure to control the simulation side, the nano-satellite side, and the communication channel. Furthermore, in Section 6.2 we define how the user can structure the tasks and the jobs to simulate the desired space mission. Lastly, in Section 6.3, we analyze the procedure that a user needs to apply if he wants to substitute the physical nano-satellite involved in the HIL simulation.

6.1. System Configuration

Our system is designed to give the user the ability to configure the system to simulate his desired space mission. The user can configure parameters regarding the simulation side, the nano-satellite side, and the communication channel.
On the simulation side, the user can configure parameters regarding the behavior of the simulation, such as the number of simulated nano-satellites, the number of jobs, or the duration of the simulation. On the nano-satellite side, the user can configure parameters regarding the functioning of the physical nano-satellite, such as the decision to simulate radiation-induced errors or not. On the communication channel, the user can configure the parameters regarding the multiple communication channels between the simulation and the physical nano-satellite. This procedure is useful if the user has to modify how is structured the communication between the two artefacts.

6.2. Configuring Tasks and Jobs

In our system, the user has the possibility to simulate his desired mission scenario for a constellation of nano-satellites. The components that enable the simulation of the user's desired scenario are the tasks and the jobs, which we describe in Section 5.1.
The jobs are the execution of one or multiple consecutive tasks, and in the scope of our

system, every job represents an objective that a space mission needs to fulfill. Instead, the tasks are the real programs that are executed on the physical nano-satellite. The user is responsible for providing the programs that are used as tasks, as it is the user who determines his desired space mission. For instance, to simulate an Earth monitoring scenario, a possible set of tasks is defined in Figure 5.1. The user must compose the jobs with the various tasks he provides, and then he must assign the jobs to the simulated nano-satellites. While forming the jobs, the user can control the ordering of how the tasks are executed through two mechanisms defined in Section 5.1. Lastly, as pointed out in Section 5.1, the user can control the number of jobs assigned to each simulated nano-satellite. If the user wants to simulate the same job along the orbits of the simulated nano-satellites, he has to assign only one job for each simulated nano-satellite. For instance, if the user wants to simulate the transmission of an Internet signal to Earth, the nano-satellites can perform the same job throughout the entire orbit, which is the transmission of a signal to the Earth. Otherwise, if the user wants to simulate different jobs along the orbits of the simulated nano-satellites, as described in the example at the end of Section 5.1, he has to assign the same number of jobs for every simulated nano-satellite.

6.3. Changing Nano-satellite

Since we design our system to be as generic as possible, a type of configuration available to the user is the substitution of the current physical nano-satellite involved in the HIL simulation, i.e. CubeSatSim, with a different nano-satellite. In this section, we analyze the main steps that the user must apply if he wants to use another nano-satellite as a physical device for the HIL simulation.

- **Modify platform-dependent programs**: The programs we used to record the data on our nano-satellite are based on the architecture of the CubeSatSim. Therefore, the user should record the data with programs based on the architecture of the nano-satellite he decides to use.
- **Modify the task component**: The task component, described in Section 5.1 is structured to receive the real data retrieved from the CubeSatSim, which we discuss in Section 5.4. If the user uses a different physical nano-satellite, he has to modify the task component according to the data retrieved from the platform-dependent programs used by the new physical nano-satellite.

Just by reconfiguring these two aspects, the user can integrate a new nano-satellite, without touching all the following aspects of the system: *i)* the interaction between the simulation and the nano-satellite, *ii)* the initial configuration of the simulation, *iii)* the functioning mechanism of the simulation, *iv)* the managing and configuration of the tasks and jobs, *v)* the simulation of radiation-induced errors, *vi)*the configuration capabilities provided to the user, *vii)* and the scalability provided to the system.

In conclusion, the user is able to configure parameters regarding the whole system. Furthermore, we analyze how the user is able to configure the different tasks and jobs simulated. Also, the user has the possibility to configure a different physical nano-satellite to collect the real data. However, till now we never described how our system is actually implemented. We describe our implementation in Chapter 7.

7 | Implementation

This chapter describes the implementation of our HIL simulation separating into two different sections the simulation side and the nano-satellite side. In Section 7.1, we describe the implementation of every single part on the simulation side. In Section 7.2, we analyze how we implemented the nano-satellite side.
Lastly, in Section 7.3 we analyze the protocols we used for the communication channels of our system.

7.1. Simulation

In this section, we analyze the implementation of the simulation side. In Section 7.1.1 we describe the file we used for the coordination between the physical nano-satellite and the simulation, which is the mapper file. Moreover, in Section 7.1.2 we analyze the structure of the simulation. Lastly, in Section 7.1.3 we describe the improvements in scalability provided to our system.
Since we use as a starting base the *cote* simulator, our simulation is written in C++ and can be started by a bash script. To ease the setup of our system we prepared different scripts. A script to set up MQTT for C++ on the machine that runs the simulation, following the instructions on the official wiki [42]. Two scripts to prepare the nano-satellite to be operative for our system. The order in which these scripts should be executed to set up both the simulation and the nano-satellite side is available in the main README of the project.

7.1.1. Mapper

The mapper is the file that is used to coordinate the simulation and the physical nano-satellite on the work to be done. This file maps every task to every job and every job to the corresponding instance of nano-satellite and it is generated by our system before the beginning of the simulation. We decided to use a file because this coordination phase requires a simple solution, which does not add additional computational overhead. Furthermore, a file can be easily inspected by the user before the beginning of the simulation to check if the jobs and tasks are assigned in the way he wants. The use of sockets would have been a reasonable choice too, but this choice could introduce a level of complexity that may be unnecessary for our coordination aspect. Furthermore, handling a socket connection can add overhead to our system. The mapper file is generated by the system following the ordering defined by the user for the tasks of the different jobs. To manage the scheduling of the different tasks a job has, we decided to treat the tasks within each job as if they are the nodes of a Direct Acyclic Graph (DAG), exploiting a topological sort algorithm, which follows the ordering mechanisms defined by the user, to order these nodes.

The user must also decide whether to simulate a single job for all the nano-satellites or multiple different jobs. This choice influences the structure of the mapper file. Simulating a single job, the generated mapper only has a single row for every simulated nano-satellite, which represents the job that the simulated nano-satellite executes. Simulating multiple jobs, the generated mapper has for each simulated nano-satellite as many rows as the jobs the user wants to simulate.

Every row is structured in the following way. All the elements are separated by a comma, the first element is the instance of the simulated nano-satellite and in the following, we can find all the tasks that the job of that simulated nano-satellite has.

Once created, the file must be transferred to the physical nano-satellite to enable coordination between the two artefacts of our system.

7.1.2. Simulation Structure

The structure of our simulation is shown in Figure 7.1. Before the beginning of the simulation, the system generates the mapper file, which we discuss in Section 7.1.1. This coordination phase is shown in the top right part of Figure 7.1. Then, the simulation can start. Firstly, the simulation sets up all the components it uses, shown in the left part of Figure 7.1. Then, as shown in the right part of Figure 7.1, whenever the simulation asks the physical nano-satellite to process a job, the simulation freezes, waiting for the results of the required job. The physical nano-satellite sends to the simulation the real data recorded for each task of the job, and the simulation integrates the real data of each task in the corresponding simulated nano-satellite which requests the simulation of the job. By using the real data received, the simulation is able to calculate the energy consumption of the constellation of simulated nano-satellites while simulating the user-desired mission scenario.

Whenever we integrate the real data from the physical nano-satellite into the simulation, for each simulated nano-satellite we simulate the effects that the energy consumption has on each simulated nano-satellite. For instance, when the jobs of the simulated nano-satellites consume a high amount of energy, the simulated nano-satellite should start sleeping, without processing further jobs, to recharge itself. The implementation of this part of the simulation was already provided by *cote*. We leave unaltered the simulation of these aspects of the simulated nano-satellites. We do not use the real data of the solar panels of the physical nano-satellite to simulate the recharging effects because the real data retrieved from the physical nano-satellite were too low and did not allow the simulation to function properly. Furthermore, adding the real data of the solar panels to the simulation of the solar panels provided by *cote* would have made the simulation even less realistic. Therefore, on this aspect of the simulation, we rely on the implementation provided by *cote*.

In conclusion, the simulation outputs the results of all the events occurring for the simulated nano-satellites. These outputs are used by the system to generate different plots, which are shown and described in Appendix B. We show two relevant examples of these plots in Figure 7.2.

In Figure 7.2a, on the x-axis we have the simulated nano-satellites, and on the y-axis, we have the energy consumption. In the plot in Figure 7.2a, we can see the total energy consumption for each simulated nano-satellite. This plot is useful to understand the various consumes of the various simulated nano-satellites. Our system also generates more in-depth plots regarding the energy consumption of each job executed by each simulated nano-satellite and the energy consumption of each task of each job. These plots are shown in Section B.1, specifically in Figure B.1b and in Figure B.1c.

In Figure 7.2b, on the x-axis we have the tasks used in the simulation, and on the y-axis, we have the energy consumption. In the plot in Figure 7.2b, we can see the average energy consumption for each task used in the current simulation. This plot is useful to check on average the consumes on the physical nano-satellite of the various tasks.

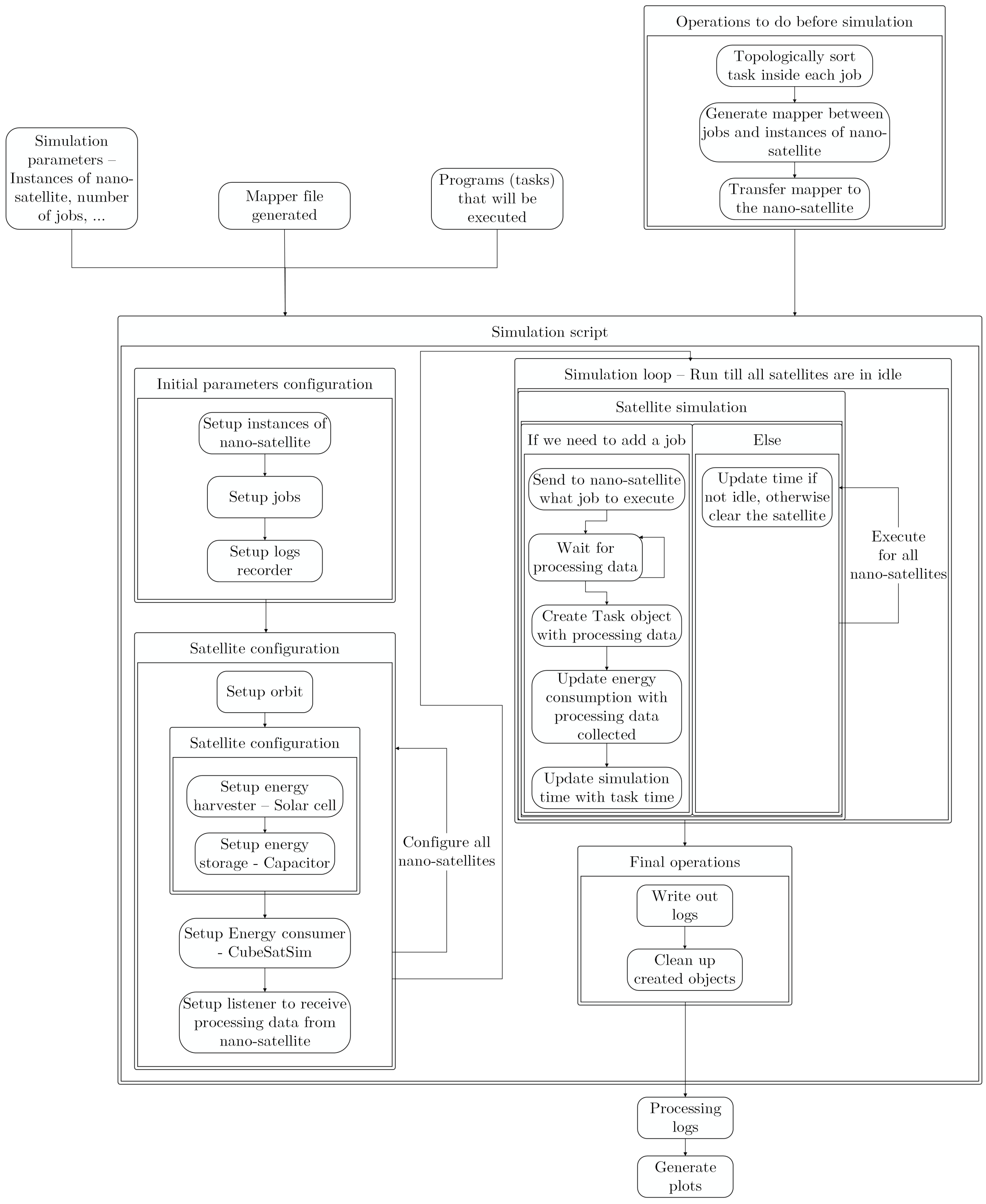
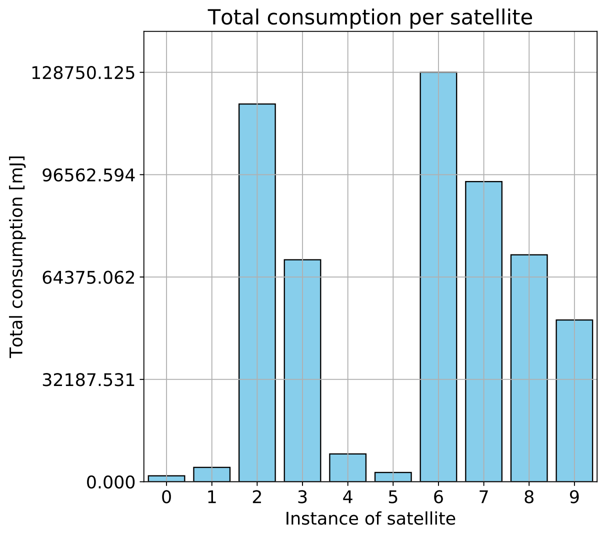
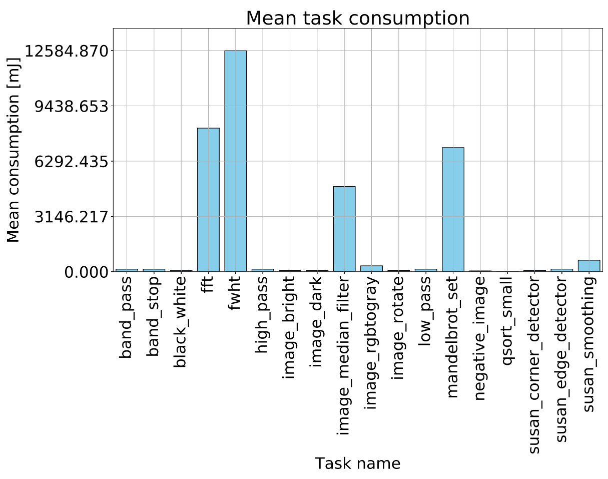

Operations to do before simulation
Topologically sort task inside each job
Generate mapper between jobs and instances of nano-satellite
Transfer mapper to the nano-satellite
Simulation parameters – Instances of nano-satellite, number of jobs, ...
Mapper file generated
Programs (tasks) that will be executed
Simulation script
Initial parameters configuration
Setup instances of nano-satellite
Setup jobs
Setup logs recorder
Satellite configuration
Setup orbit
Satellite configuration
Setup energy harvester – Solar cell
Setup energy storage - Capacitor
Setup Energy consumer - CubeSatSim
Setup listener to receive processing data from nano-satellite
Configure all nano-satellites
Simulation loop – Run till all satellites are in idle
Satellite simulation
If we need to add a job
Send to nano-satellite what job to execute
Wait for processing data
Create Task object with processing data
Update energy consumption with processing data collected
Update simulation time with task time
Else
Update time if not idle, otherwise clear the satellite
Execute for all nano-satellites
Final operations
Write out logs
Clean up created objects
Processing logs
Generate plots

Figure 7.1: Structure of simulation side.



(a) Total energy consumption.



(b) Mean energy consumption per task.

Figure 7.2: Plots generated by an execution of our HIL simulation.

7.1.3. Scalability

To enhance the scalability of our system we used a multithreading and multidevice approach, shown in Figure 5.4. In our approach, each simulated nano-satellite is under the control of a different thread, represented by the various elements in the top part of Figure 5.4, enabling the simulation to communicate with multiple physical devices simultaneously. This is advantageous because the simulated nano-satellite that requests the real data waits for the data from the physical nano-satellite on its own thread; but the other simulated nano-satellites, which are using their own thread, can proceed with their simulation. Thanks to the multidevice approach, shown in the bottom part of Figure 5.4, the other simulated nano-satellites can request to process their jobs to the other physical nano-satellites used in the system, until all the physical nano-satellites are working. At this point, the remaining simulated nano-satellites are required to pause their operations until at least one physical nano-satellite becomes available for use. This behavior results in enhanced parallelism, speeding up the total execution time of the HIL simulation, and also avoiding possible bottlenecks derived from the execution of a time-consuming job for specific a simulated nano-satellite.

7.2. Nano-satellite

In this section, we describe the implementation of the programs that operate on the nano-satellite. In Section 7.2.1 we describe the two programs that are used to record the non-functional and the payload data. Furthermore, in Section 7.2.2 we examine the implementation of the program that communicates with the simulation to receive and send the real data. Lastly, in Section 7.2.3 we examine the implementation of injecting faults to simulate errors caused by radiations.

7.2.1. Recordings Programs

Correspondences EPS

	INA219 Sensor	I2C Bus	I2C Address
1	+X I/V Sensor	1	0x40
2	+Y I/V Sensor	1	0x41
3	-Y I/V Sensor	3	0x44
4	-Z I/V Sensor	3	0x45
5	+Z I/V Sensor	3	0x40
6	-X I/V Sensor	3	0x41
7	BAT I/V Sensor	1	0x44
8	5V I/V Sensor	1	0x45

Table 7.1: Correspondence between the addresses, buses, and components in the EPS [38].

We use 2 different programs, which we show in the right part of Figure 7.3, to retrieve the non-functional and the application payload data from the physical nano-satellite. Both types of data are explained in Section 5.4. The two programs record data at predefined sampling periods and to increase the fidelity, we shorten this sampling period to retrieve a higher amount of data. The definition of the sampling periods is different for the two subsets of real data and is defined below where we explain the functioning of the two programs. In the following, we explain how these programs work.

- **Record non-functional data**: The data are recorded on the I2C buses 1 and 3 and on the addresses [0x40, 0x41, 0x44, 0x45], using eight different INA219 sensors. These sensors retrieve the voltage and current from eight different components of the CubeSatSim, which specifically are: the battery, the Power Supply Unit, and the six solar panels mounted. We can see, the correspondence between the addresses, buses, and the components recorded in the EPS in Table 7.1. After the nano-satellite records the data on all eight components, it retrieves the temperature of its CPU This program stores the non-functional data in a file, accessed by another program to send this data to the simulation. To retrieve the data from the eight components, the program has to request each component its voltage and current, and between the requests, the program must observe a brief pause to ensure that the data requested has arrived and to avoid overloading the I2C bus, on which the data is transferred, and the recording components. With this procedure, the amount of time to retrieve the data from all components composes the sampling period.

- **Retrieves application payload data**: This program opens a serial communication with the STEM payload board, defined in Section 3.5.2, mounted on the CubeSatSim, using the UART serial port of the Raspberry PI. On the STEM payload board are mounted an Arduino board and an STM32, which run scripts to retrieve all the application payload data. These scripts are already provided by the CubeSatSim. By using the communication channel, this program retrieves all the application payload data and stores these data in a file. This file is later accessed by another program to send the application payload data to the simulation. To retrieve the data from the device, the program needs to request the retrieval of the data on the communication channel opened and wait for the data on the channel. This amount of time composes the sampling period.
  These data are not actually used in the scope of this thesis, but we decided to integrate them into our system to provide a more complete set of the data provided by the CubeSatSim and facilitate the use of this data in a future extension of our system to enhance its realism.

All the programs in excess, which are executed on the physical nano-satellite, such as the two programs we just discussed and the program described in Section 7.2.2, generate overheads that should be not taken into consideration when calculating the energy consumption. These overheads should be subtracted from every value of energy consumption we calculate because these programs are used just in the scope of our system and are not used in the scope of a real space mission. Therefore, to enhance the realism of our system we decided to remove these overheads. To calculate this value we run for thirty minutes four different configurations all having a program in the background that records the non-functional data. The first configuration has no other program running and it is useful to be used as a basis to be subtracted from the other configuration. In the second configuration, we run the program to record the non-functional data. In the third configuration, we run the program to record the payload data. In the last configuration, we run the program described in Section 7.2.2.

To retrieve the overhead of each program we subtract the first configuration, which is the basis used in this overhead calculation, from the other three configurations, which represent the energy consumption for the three programs for which we want to calculate the overheads. In conclusion, by summing the three obtained values, we retrieve the total overhead of the three programs. This overhead can be subtracted in the simulation from the energy consumption of each task, to obtain a more realistic value of the energy consumption. When the simulation ends, it stops these two programs using the same communication channel used for the initial coordination of the system.

7.2.2. Command Receiver

The command receiver is a Python program used to communicate with the simulation. We can see the general structure of this program in the left part of Figure 7.3, while on the right part of Figure 7.3, we can see the interaction of the command receiver program with the files used to retrieve the real data. In the following, we describe in depth how the command receiver program works. When this program receives a command to execute a job, it reads the mapper file and prepares the sequential execution of all the tasks inside the requested job. Then, the command receiver program prepares the execution of the first task of the job. The program constructs a string with all the parameters configured by the user for that task. After the string is built, the program executes the task in a shell, spawned as a child of the command receiver process. In this way, the command receiver program has control over the shell. We need control over the spawned shell because the tasks corrupted to simulate radiation are unpredictable and can potentially stop the simulation for a very long time, or even make the nano-satellite crash, invalidating the execution of the simulation. Therefore, after a predefined amount of time, defined a priori by the user, the task is automatically killed.
When the task ends, the command receiver sends to the simulation the following data: output of the task, duration of the task, non-functional data recorded during the execution of the task, retrieved in the top right part of Figure 7.3, and payload data recorded during the execution of the task, retrieved in the bottom right part of Figure 7.3. After the program sends all the data of a task, it starts preparing the following task. This procedure is executed for all the tasks of the requested job. Once the command receiver program completes a job, it starts waiting for the request of execution of another job. When the simulation ends, it stops this program using the same communication channel used for the initial coordination of the system.

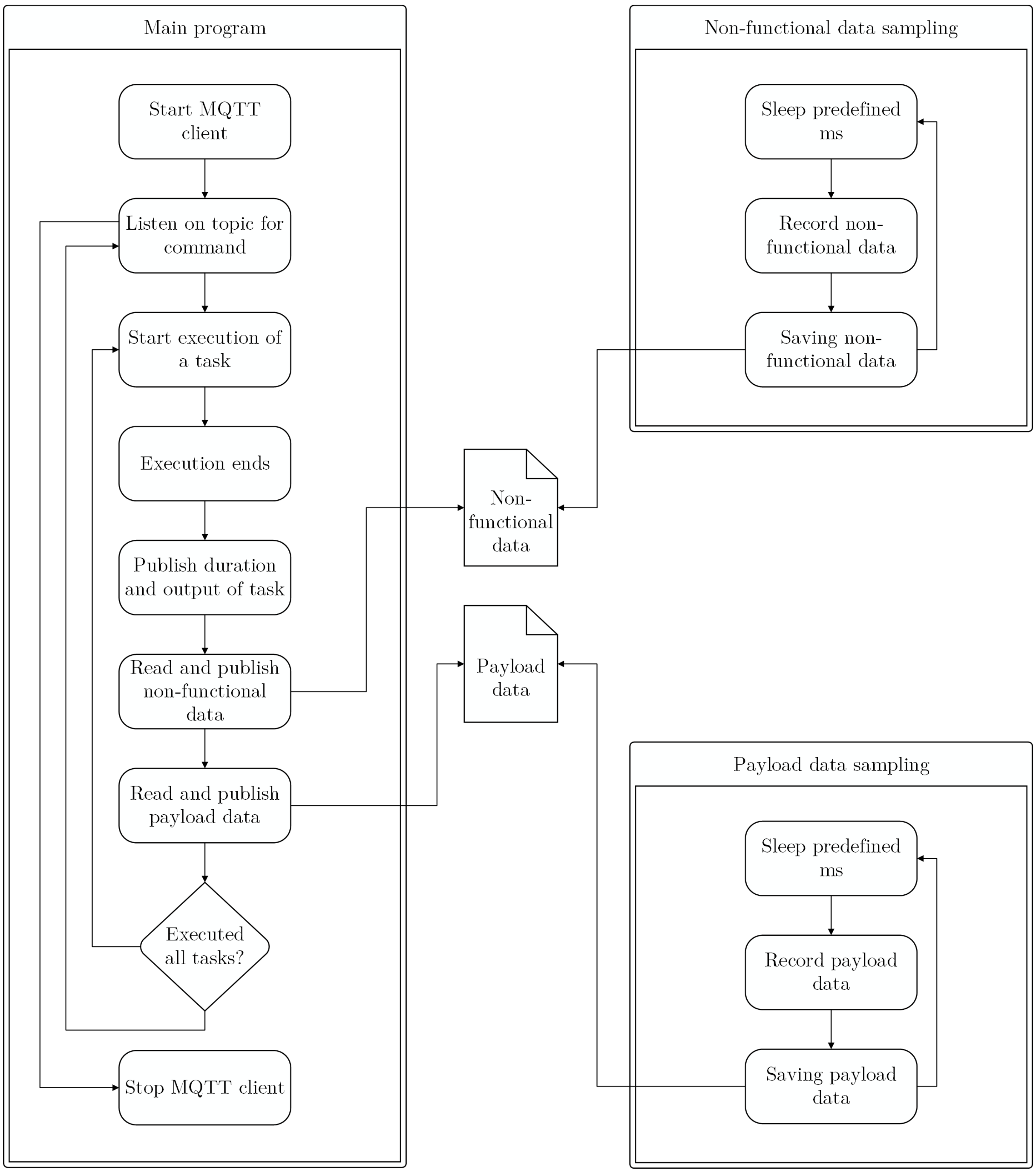

Main program
Start MQTT client
Listen on topic for command
Start execution of a task
Execution ends
Publish duration and output of task
Read and publish non-functional data
Read and publish payload data
Executed all tasks?
Stop MQTT client
Non-functional data
Payload data
Non-functional data sampling
Sleep predefined ms
Record non-functional data
Saving non-functional data
Payload data sampling
Sleep predefined ms
Record payload data
Saving payload data

Figure 7.3: Nano-satellite side.

7.2.3. Radiations

The simulation of radiation-induced errors is done on the nano-satellite side. We simulate these errors by randomly bit-flipping or corrupting the memory addresses of a task before its execution. To do so, we decided to use a fault-injection program called Chaos Duck [81]. We slightly modify the code to adapt it to our needs. This fault-injection program works in the following way. Chaos Duck takes two inputs: the task where to inject the faults and an int, which represents the type of fault injection the user wants; the choice in our case is between bit-flipping and modifying memory addresses. Then, Chaos Duck decompiles the task in Assembly. This decompile process is based on the processor on which the task was compiled. Since the nano-satellite we employed, which is the CubeSatSim, used the Raspberry PI Zero 2 W processor, which is an ARM processor, we define statically in the code to decompile in ARM instructions. Lastly, based on the user choice, Chaos Duck injects the corresponding faults. Both methods of randomly bit-flipping or modifying the memory addresses of a task, create a list of corrupted ARM instructions, and then the program chooses randomly only one instruction to corrupt. This is done to better simulate the unpredictability of the radiation-induced errors. Therefore, when we bit-flip, a random bit of an instruction is flipped, instead in the other case, we modify the address of a random jump instruction of the Assembly code. Lastly, since our system aims to provide random corruption every time, the nano-satellite dynamically corrupts the tasks at run-time.

7.3. Communication Channel

For the two communication channels used in our system, we use reliable and platform-independent protocols because we want to guarantee that both artefacts receive what they need and we want to provide higher flexibility to the system. The first communication channel, used for the initial coordination phase of our system, uses SSH. The second communication channel, used to actively enable communication between the two artefacts, uses MQTT as protocol. Among all the platform-independent and reliable communication protocols, we choose MQTT, because it is lightweight, scalable, and enables bidirectional communication [3]. The communication between the two artefacts is achieved by subscribing and publishing on the topics defined in Table 7.2.

MQTT topics

	Topic	Communicatio direction	Description
1	*cubesatsim/commands/'int: physical nano-satellite'/'int: simulated instance of nano-satellite'*	Simulation ↓ Nano-satellite	Used to send to the nano-satellite the task to execute.
2	*cubesatsim/#*	Simulation	The simulation subscribes and waits for the real data from the physical nano-satellite of the job it requested.
3	*cubesatsim/appdata/'int: physical nano-satellite'/'int: simulated instance of nano-satellite'/'int: task executed'*	Nano-satellite ↓ Simulation	Used to send to the simulation the final output of the corresponding task.
4	*cubesatsim/duration/'int: physical nano-satellite'/'int: simulated instance of nano-satellite'/'int: task executed'*	Nano-satellite ↓ Simulation	Used to send to the simulation the duration of execution of the corresponding task.
5	*cubesatsim/nonfuncdata/'int: physical nano-satellite'/'int: simulated instance of nano-satellite'/'int: task executed'*	Nano-satellite ↓ Simulation	Used to send to the simulation the non-functional data recorded during the time of execution of the corresponding task.
6	*cubesatsim/payload/'int: physical nano-satellite'/'int: simulated instance of nano-satellite'/'int: task executed'*	Nano-satellite ↓ Simulation	Used to send to the simulation the payload data recorded during the time of execution of the corresponding task.

Table 7.2: Description of the MQTT topics used by the simulation and by the nano-satellite.

In conclusion, the implementation of our system is available at the following repository: HIL Simulation.
In this chapter, we analyze in-depth the whole implementation of our system. We conducted an experimental evaluation to validate our system under several aspects. We examine this experimental evaluation in Chapter 8.

8 | Experimental Evaluation

This chapter describes the evaluation we conducted on our system. In Section 8.1 we explain the different inputs we use for our evaluation, while in Section 8.2 we discuss the results obtained.

We evaluate our system to discover if the HIL simulation we develop is actually able to retrieve and integrate real data from a physical nano-satellite. Furthermore, we want to discover if the decrease in the sampling period for the program that retrieves non-functional data enables the physical nano-satellite to gather more data, providing a better result of energy consumption, without compromising the truthfulness of the data. In conclusion, to test the scalability provided to our system, our evaluation aims to find the number of physical nano-satellites supported by the HIL simulation and the overall reduction in execution time of the simulation.

From our evaluation, we prove that our simulation is capable of integrating real data and the scalability provided to our system shortens the overall simulation time by a factor of 88% and enables the simulation to interact with ~700 physical nano-satellite simultaneously.

8.1. Inputs

In the following sections, we describe the inputs we used for our experimental evaluation. In Section 8.1.1 we define the metrics used. Moreover, in Section 8.1.2 we describe the baselines we compared our results with, and in conclusion, in Section 8.1.3 we illustrate the benchmarks we used in our evaluation.

We conduct the evaluation using the setup shown in Figure 8.1. Our simulation runs on a machine with the following specifications: Ubuntu 18.04 LTS, 48 CPUs, 256 GB of memory, and 1 TB of Disk. We work on this machine via SSH using the "Remote Development" feature provided by the CLion IDE, shown in the central monitor of Figure 8.1. On the left of Figure 8.1, we can see the computer that is connected directly to the real nano-satellite to provide an Ethernet connection to the nano-satellite, using the Ethernet over USB functionality. For the physical nano-satellite, shown in the right part of Figure 8.1, we decided to mount the Raspberry PI 2 Zero W on the CubeSatSim as the main board, choosing it from the two Raspberry PIs we tested, described in Section 3.5.1. Our choice is based on the fact that despite the higher energy consumption, the Raspberry PI Zero 2 W has greater computational capabilities compared to the Raspberry PI Zero W.

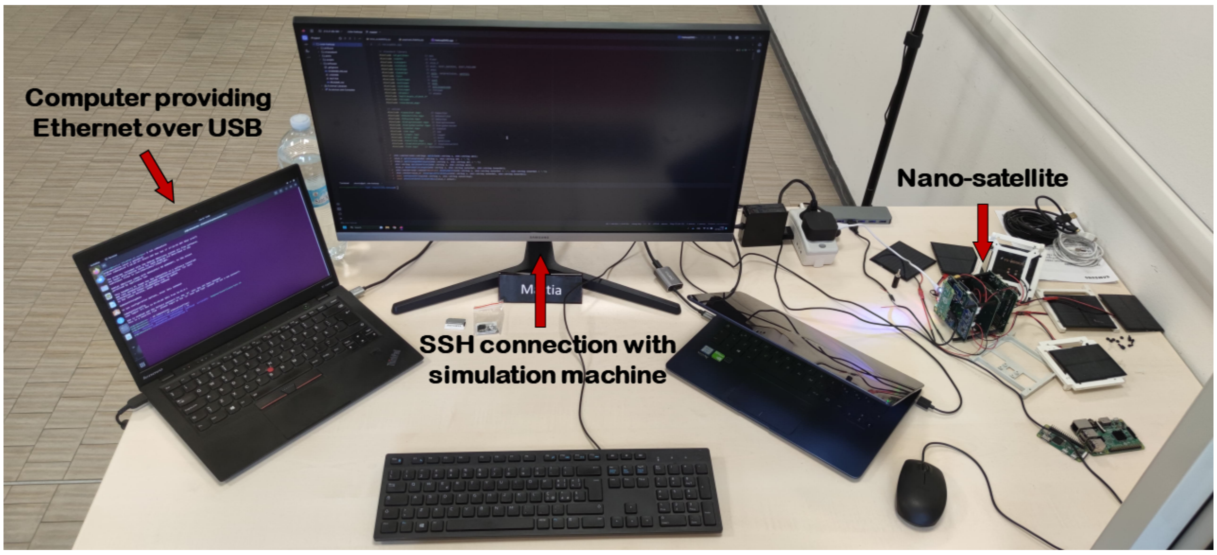

Computer providing Ethernet over USB
Nano-satellite
SSH connection with simulation machine

Figure 8.1: My setup.

8.1.1. Metrics

Our experimental evaluation is based on three different metrics: realism, fidelity, and scalability.

- **Realism**: For this metric, we want to evaluate if our system is able to retrieve real data from the physical nano-satellite and is able to integrate the real data retrieved in the simulation, providing a comprehensive HIL simulation. The data retrieved from the physical nano-satellite must be integrated into the corresponding simulated nano-satellite, which requests the processing of a job. This metric is crucial to understand if the system we provide is actually able to integrate real data in the simulation. Specifically, in this metric we opt to compare the energy consumption of the simulated nano-satellites to the energy consumption of the components of the CubeSatSim found in datasheets online.
- **Fidelity**: For this metric, we want to evaluate if the decrease in the sampling period of the program that retrieves the non-functional data, enables the physical nano-satellite to gather more data and provide a better result for the energy consumption, without compromising the truthfulness of the data. We focus on modifying the sampling period of the program that retrieves the non-functional data because in our

simulation we focus on computing the energy consumption of the simulated nano-satellites. We want to evaluate this metric, based on the reasoning that a higher amount of data provides a more faithful result, but we do not want to compromise the data retrieved, otherwise, we lose the realism of our system. Furthermore, as explained in Section 5.4, our reasoning is based on the fact that by collecting more data in the same period of time, there is a higher probability of retrieving peaks unseen by collecting fewer data.
To evaluate this metric we calculate the percentage variations in the number of samples and the variations in the mean current retrieved from the physical nano-satellite. By calculating the percentage variation in the number of samples, shown in Equation (8.1), we evaluate that the modification in the sampling period does not compromise the correct functioning of the program that retrieves the data. By calculating the percentage variation in the mean current, shown in Equation (8.2), we evaluate that the modification in the sampling period does not compromise the truthfulness of the data.

$$Fidelity_{Samples} = \frac{Samples - Samples_{Baseline}}{Samples_{Baseline}} * 100 \tag{8.1}$$

$$Fidelityy_{Current} = \frac{MeanCurrent - MeanCurrent_{Baseline}}{MeanCurrent_{Baseline}} * 100 \tag{8.2}$$

- **Scalability**: Firstly, for this metric, we want to evaluate how the number of jobs impacts the execution time of the simulation. This test is relevant to understanding the behavior of the simulation on the modification of the number of jobs that need to be simulated.
  Furthermore, we want to evaluate if the scalability we provided to our system is able to decrease the overall execution time of the simulation. We want to test this behavior because our enhanced scalability aims to speed up the overall execution time of the simulation, by enabling the system to parallelize the execution of the simulated nano-satellites and by enabling the simulation to interact with multiple physical nano-satellites simultaneously.
  Moreover, we want to evaluate how our system is able to scale. Therefore, we want to test if a simultaneous increase of simulated and physical nano-satellites provides a similar execution time of the simulation. We want to evaluate this behavior until we find a point of failure. This point of failure is important to understand the limit in our scalability. This test is useful to understand if our system is able to handle the increased parallelization and maintain its performance under higher counts of simulated and physical nano-satellites. To evaluate all the aspects of this metric, we

calculate the percentage variations of the seconds of execution time of the simulation by using Equation (8.3).

$$Scalability = \frac{Seconds - Seconds_{Baseline}}{Seconds_{Baseline}} * 100 \tag{8.3}$$

8.1.2. Baselines

Based on the specific metric, we use a different baseline to evaluate our system.

- **Realism**: The baseline we use for this metric corresponds to the energy consumption values found in datasheets online, which are defined in Section 8.1.3.
- **Fidelity**: The baseline we use for this metric corresponds to using a sampling period at 100 ms. We define this sampling period as a baseline because it is the original value used by the CubeSatSim. This value is used by the CubeSatSim for a specific reason: the CubeSatSim project does not need to obtain a high amount of data to calculate the energy consumption in an interval of time, as it is in the case of our simulation, which calculates the energy consumption during the duration of a task, but the CubeSatSim needs to retrieve just 1 sample of data, therefore it does not need a lower sampling period. Furthermore, using this sampling period, we do not overload the components that record the data.
- **Scalability**: For this metric, we provide a configuration using a single physical nano-satellite, to which we normalize the results we obtain.

8.1.3. Benchmarks

In our evaluation, we provide the following benchmarks:

- **Datasheets on the power consumption of the CubeSatSim components**: These datasheets regard the power consumption of the various components of the CubeSatSim. We use these values to evaluate the "realism" metric. In Table 8.1 we can see the value we use. We opt to use the power consumption of the datasheets found online because these data represent a real-world scenario and are a reflection of how the various components typically behave in the real world.
- **General processing**: These programs represent possible tasks that we use in the evaluation of our system. These programs are used to apply computational pressure to the nano-satellite. The complete list of programs, along with their sources and descriptions are listed in Table 8.2.

- **Image processing**: These programs represent possible tasks that we use in the evaluation of our system. These programs process an image given in input with different techniques. The scope of these programs is to simulate Earth monitoring scenarios, which is an activity usually accomplished by nano-satellites. The complete list of programs, along with their sources and descriptions are listed in Table 8.3.
- **Signal processing**: These programs represent possible tasks that we use in the evaluation of our system. These programs process an auto-generated signal in different ways. The scope of these programs is to simulate telecommunication scenarios and the processing of signals. The complete list of programs, along with their sources and descriptions are listed in Table 8.4.

Datasheets power consumption

	Component	Power consumption
1	Raspberry PI Zero 2 W	1.375 [W]
2	ACEIRMC BME280 Digital 5V	12.96 [μW]
3	HiLetgo GY-521 MPU-6050 MPU6050	12.456 [mW]
4	8 ACEIRMC INA219 I2C Bi-Directional	10,080 [mW]
5	Pro Micro (Dev-12587)	165 [mW]

Table 8.1: Power consumption derived from datasheets of the components of the CubeSatSim.

General processing tasks

	Program name	Description
1	mandelbrot_set [76]	Generates the Mandelbrot set in an image in the output.
2	qsort_small [34]	Sorts big arrays of data.
3	basicmath_large [34]	Executes different mathematical operations.

Table 8.2: Tasks for processing.

Image processing tasks

	Program name	Description
1	black_white [69]	Converts an image given in input to black and white.
2	image_bright [69]	Makes an image given in input brighter.
3	image_dark [69]	Darken an image given in input.
4	image_rgbtogray [69]	Converts an RGB image given in input to a grayscale one.
5	image_rotate [69]	Rotates an image given in input.
6	negative_image [69]	Makes a negative out of an image given in input.
7	image_median_filter [4]	Implements a median filter, which is normally used to reduce noise in an image [44].
8	susan_edge_detector [34]	Detects the edges inside an image given in input.
9	susan_corner_detector [34]	Detects the corners inside an image given in input.
10	susan_smoothing [34]	Reduces the noise inside an image given in input.

Table 8.3: Tasks simulating Earth monitoring activities.

Signal processing tasks

	Program name	Description
1	fft [34]	Implements the Fast Fourier Transform.
2	low_pass [28]	A low pass filter for an auto-generated signal.
3	high_pass [28]	A high pass filter for an auto-generated signal.
4	band_pass [28]	A band-pass filter for an auto-generated signal.
5	band_stop [28]	A band-stop filter for an auto-generated signal.
6	fwht [70]	Implements the Fast Walsh Hadamard Transform in C.

Table 8.4: Tasks simulating telecommunication activities.

8.2. Results

In the following, we exhibit the results obtained after conducting the experimental evaluation described in the previous sections. In Section 8.2.1 we describe the results obtained from the evaluation of the realism metric. In Section 8.2.2, we analyze the results obtained during the evaluation of the fidelity metric. In Section 8.2.3 we describe the results obtained while evaluating the scalability metric.
As we can see in Figure 8.2, the current of the data retrieved from the physical nanosatellite demonstrates statistical variability, highlighting a noticeable range in the values. Therefore we check that among all the data used for our evaluation the coefficients of variation, defined in Equation (8.4), is at most 5%.

$$\sigma^* = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{\sum_{i=1}^{N}(Element_i - \mu)^2}{N}}}{\frac{\sum_{i=1}^{N} Element_i}{N}} \tag{8.4}$$

In Equation (8.4), N represents the number of elements we have for the evaluation of every metric.
In addition, in the upcoming sections, each evaluation plot includes the standard devia-

tion, defined in Equation (8.5), for all the values.

$$Std.dv. = \sqrt{\frac{\sum_{i=1}^{N} (Element_i - \mu)^2}{N}} \tag{8.5}$$

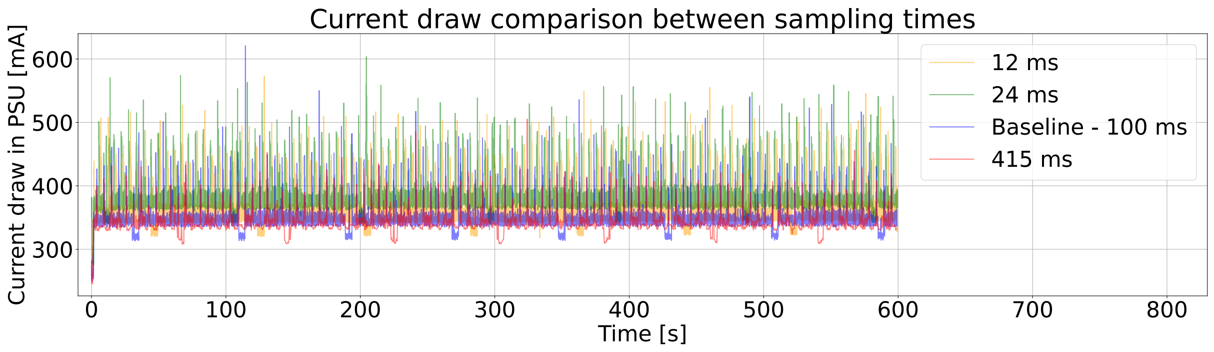


Figure 8.2: Visual representation of the data recorded using the different sampling periods.

8.2.1. Realism

Jobs configuration.

Instance of nano-satellite	Tasks
0	black_white + image_bright + image_dark
1	image_rgbtogray + image_rotate + negative_image
2	mandelbrot_set + image_median_filter
3	fft + qsort_small
4	susan_edge_detector + susan_corner_detector + susan_smoothing
5	low_pass + high_pass
6	band_pass + band_stop + fwht
7	fft + image_rotate
8	image_bright + mandelbrot_set
9	high_pass + susan_corner_detector + image_median_filter

Table 8.5: Configuration used in the evaluation for the jobs of the simulated nano-satellites.

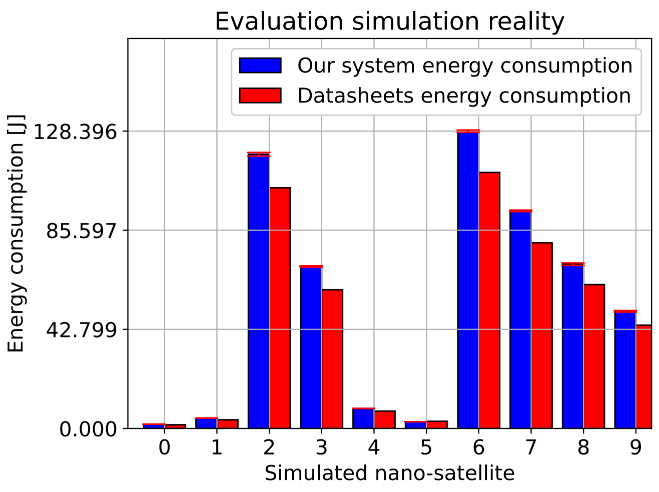


Figure 8.3: Results of the evaluation of the accuracy metric.

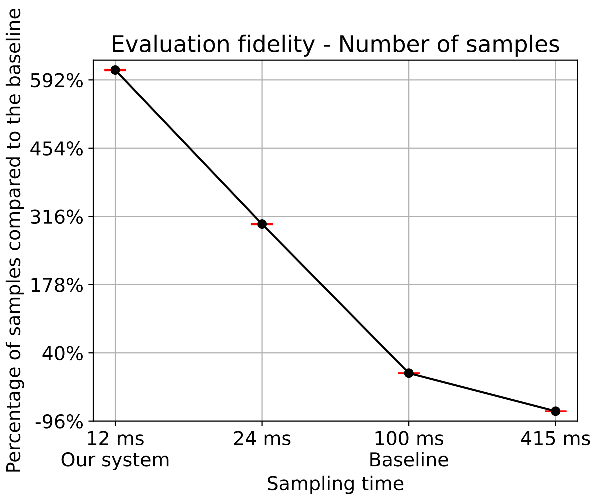
In the scope of the evaluation of this metric, we configure the system in the following way. We use 10 simulated nano-satellites, sampling period at 12 ms, 10 jobs, and 1 physical nano-satellite. Therefore, in this situation, the 10 simulated nano-satellites must interact with the single physical nano-satellite to retrieve the data. We define the configuration of the tasks and the jobs in Table 8.5. We use this configuration to enable each simulated nano-satellite to simulate different objectives of different mission scenarios. Therefore, the various objectives provide different values of energy consumption with which we can evaluate our system. For instance, the instance of nano-satellite 0 processes a job regarding an Earth monitoring scenario, while the instance of nano-satellite 6 processes a job regarding a communication scenario; these two scenario uses different tasks that provide two different values of energy consumption, as we can see from the energy consumption plots given in output by the simulation, shown in Figure B.1d. To evaluate if our system is able to manage real data, we compare the energy consumption retrieved from the physical nano-satellite and integrated into the various simulated nano-satellites, to the energy consumption retrieved from the datasheets of the components of the CubeSatSim. To convert the power consumption of the datasheets to energy consumption, which is the measure we use in Figure 8.3, we multiply for every simulated nano-satellite the power consumption of the datasheets by the interval in which the physical nano-satellite processes all the jobs of the simulated nano-satellite.

In Figure 8.3 we can see how we test our system to evaluate if it is able to retrieve real data from the physical nano-satellites and integrate these data in the simulated nano-satellites. On the x-axis, we can see the various simulated nano-satellites in which we integrate the real data. On the y-axis, we can see the energy consumption in Joule. In Figure 8.3, we can see two types of bars, the blue bar represents the energy consumption retrieved from the physical nano-satellite of the simulated nano-satellites, and the red bar represents the energy consumption retrieved from the datasheets. As we can see in Figure 8.3, for every simulated nano-satellite the energy consumption has nearly the same magnitude as the energy consumption retrieved from the datasheets. Nevertheless, there exists a subtle percentage variation between the datasheets and the various simulated nano-satellites, indicating that the energy consumption specified in the datasheets tends to be lower than the actual energy consumption observed in the simulations. This percentage variation can be attributed to different aspects:

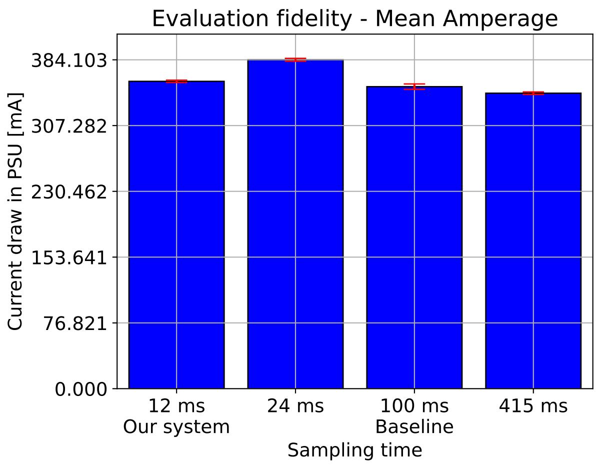
- **Environmental aspects**: There could be variations due to environmental factors, which in the case of our physical nano-satellite may be different from those considered in the datasheets.
- **CPUs usage**: The number of CPUs used may affect the power consumption retrieved because the use of more CPUs may require more power consumption since the device is working more. For instance, the power consumption of the Raspberry PI working with 4 CPUs is different from the power consumption when the Raspberry PI works with 1 CPU. Therefore, the power consumption from the datasheets may use a different amount of CPUs compared to our physical nano-satellite.
- **Program used to test**: The tasks that are executed on the physical nano-satellite, during the recording of the power consumption, can have a significant impact on the results because different programs may require different amounts of system resources, providing different power consumption. Therefore, the power consumption on the datasheets may be calculated using different programs to stress the various components of the nano-satellite.
- **Voltage variation**: In a real situation the voltage is not fixed on a single value. Therefore the standard voltage value of the components, which we use to calculate the power consumption of the datasheets, may differ from the real voltage value, which we use to calculate the power consumption of the simulated nano-satellites.

Finally, since the energy consumption we compared has nearly the same magnitude, we can conclude that our simulation is able to integrate the real data provided by the physical nano-satellite in the various simulated nano-satellites.

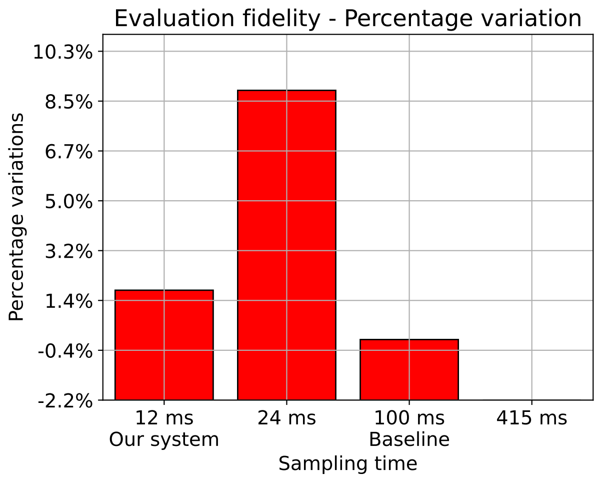
8.2.2. Fidelity



(a) Evaluation results on the number of samples.



(b) Evaluation results on the average current using the precedent sampling periods.



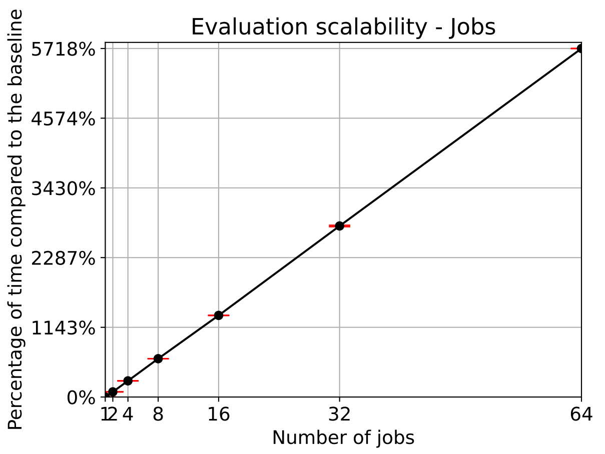
(c) Evaluation results of the percentage variations in the average current using the precedent sampling periods.

Figure 8.4: Results of the evaluation of the fidelity metric.

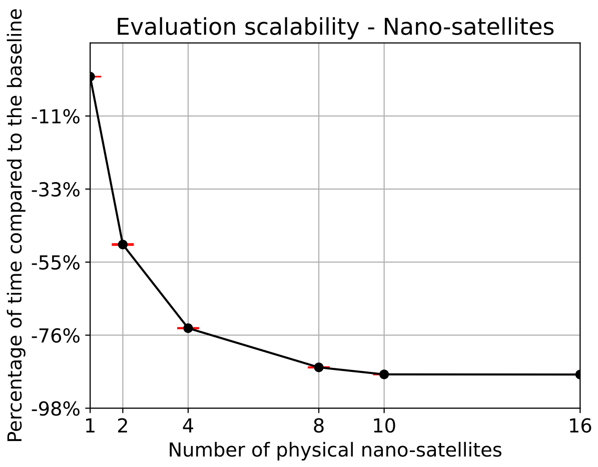
In the scope of the evaluation of this metric, we configure the system in the following way. In this metric, we focus on the nano-satellite side and we decided to record the data on the physical nano-satellite for 10 minutes while it is working. The specific program used for processing is not relevant, because we just need general processing, and for this reason, we decided to run *basicmath_large* in a loop for 10 minutes. In the process of modifying the sampling period, we *i)* modify the pauses used by the program that retrieves the data to not overload the recording components, in our evaluation these cases are represented

by the sampling period at 24 and 415 ms, or we *ii)* record the non-functional data only from the relevant components that we use in our simulation, which is the current drawn by the PSU, in our evaluation this case is represented by the sampling period at 12 ms.
In Figure 8.4a we evaluate the variations in the number of samples when we modify the sampling period. On the x-axis, we can see the sampling period we evaluated. On the y-axis, we can see the percentage variation in the number of samples compared to the baseline. As we can see in Figure 8.4a, the number of data recorded keeps increasing as long as we decrease the sampling period, therefore, the overloading on the components that record the data does not severely impact the number of samples recorded. Our system enabled us to record 671% more data compared to the baseline. A relevant case, not shown in Figure 8.4a, is the sampling period obtained when we completely remove the pauses used to not overload the recording components. In this case, the program that records the non-functional data stops functioning, and therefore, this sampling period cannot be used.
In Figure 8.4b and in Figure 8.4c we evaluate the difference of the data recorded among all the sampling periods. To evaluate this aspect, we consider the mean value of the current, measured in [mA], recorded on the 5V bus, which is the bus connected to the PSU. In Figure 8.4b on the x-axis, we can see the sampling period we evaluated and on the y-axis, we can see the current drawn in the PSU, recorded in the various sampling periods. In Figure 8.4c on the x-axis, we can see the sampling period we evaluated and on the y-axis, we can see the percentage variations of the current drawn in the PSU for the various sampling periods, compared to the baseline. As we can see in Figure 8.4b, by using a longer sampling period we find a lower average value, and by decreasing the sampling period we have a higher average value. This happens because by collecting more samples, there is a higher probability of finding peaks, compared to when we collect fewer samples. The bars in Figure 8.4b have similar values, therefore, in Figure 8.4c, we provide the percentage variations of the current drawn in the PSU to better visualize the data of Figure 8.4b. One noteworthy case is the sampling period at 24 ms. In this case, the average value is definitely higher compared to the other cases and this would appear to be in contradiction with our earlier statement. But, to obtain the sampling period at 24 ms, we are loading the device with more work than the other cases, overloading the recording components. Therefore, even if we obtain a higher current value, the sampling period at 24 ms is not directly comparable to the other sampling periods. To verify this statement, we checked with *htop* the CPU usage during the evaluation of all the sampling periods. We noted that in the case of the sampling period at 24 ms, the nano-satellite was using nearly 3 CPUs, instead in all the other sampling periods the nano-satellite was using nearly 2 CPUs.

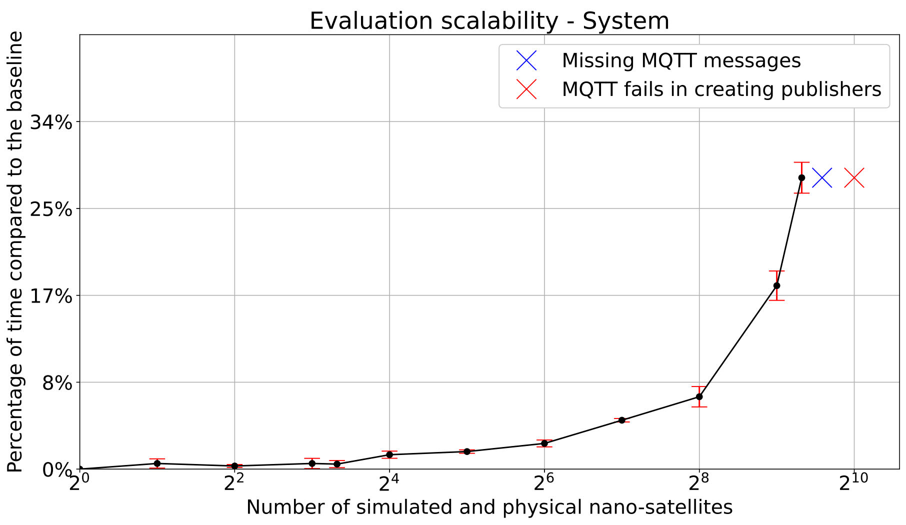
8.2.3. Scalability



(a) Evaluation results on the scalability of the jobs.



(b) Evaluation results on the decreasing of the overall execution time of the simulation.



(c) Evaluation results on the scalability of system.

Figure 8.5: Results of the evaluation of the scalability metric.

In the scope of the evaluation of this metric, we configure the system in the following way. We use 10 simulated nano-satellites, sampling period at 12 ms, 20 jobs, and 1 physical nano-satellite. Therefore, in this situation, the 10 simulated nano-satellites must interact with the single physical nano-satellite to retrieve the data. Since we do not have more than 1 physical nano-satellite, our approach involves utilizing multiple scripts to simulate the functioning mechanism of multiple nano-satellites, when we need more than 1 physical nano-satellite. The script we use is deterministic, the data that the script sends to the simulation is static, and the recorded data are 3 seconds long. Due to these considerations, we have configured the script to transmit the data to the simulation with a 3-second delay following the simulation's request. We define the configuration of the tasks and the jobs in Table 8.5. We use this configuration to enable each simulated nano-satellite to simulate different objectives of different mission scenarios. For instance, the instance of nano-satellite 0 processes a job regarding an Earth monitoring scenario, while the instance of nano-satellite 6 processes a job regarding a communication scenario.

In Figure 8.5a we want to evaluate the impacts that the modification of the number of jobs has on the simulation, to understand the behavior of the simulation when it needs to deal with more processing for each simulated nano-satellite; therefore, in this test we modify the number of jobs of each simulated nano-satellites. On the x-axis, we can see the number of jobs we evaluated. On the y-axis, we can see the percentage variation in the execution time of the simulation compared to the baseline, which is 1 job per simulated nano-satellite. As we can see, there is a linear increase in the time of execution of the simulation. This happens because we are just duplicating the jobs that the simulation has to process, hence we are duplicating the execution time of the simulation.

In Figure 8.5b we want to evaluate if the scalability in terms of the number of physical nano-satellites, decreases the execution time of the simulation; therefore, in this test, we modify the number of physical nano-satellites with which the simulation interacts. On the x-axis, we can see the number of physical nano-satellites we evaluated. On the y-axis, we can see the percentage variation in the execution time of the simulation compared to the baseline, which in this case is set to 1 physical nano-satellite. As we can see, by increasing the number of physical nano-satellites, the execution time of the simulation decreases. This behavior is attributed to the fact that since we are exploiting parallel execution to simulate each simulated nano-satellite separately and we are using multiple physical nano-satellites: if a physical nano-satellite is currently working for a simulated nano-satellite, the other simulated nano-satellites can request the data to the free physical nano-satellites. This behavior occurs until the number of physical nano-satellites reaches the number of simulated nano-satellites. At this point, we can see from Figure 8.5b that the curve flattens out because every simulated nano-satellite always has a physical nano-satellite to interact with, therefore, when we have more physical than simulated nano-satellites, the extra physical nano-satellites never work. For instance, in Figure 8.5b, when we have 16 physical nano-satellites and 10 simulated nano-satellites, 10 physical nano-satellites always retrieve the real data for the 10 different simulated nano-satellites, while the remaining 6 physical nano-satellites never work. In conclusion, from the results we obtained, we can say that the scalability provided to our system is able to decrease the overall execution time of the simulation by a factor of 88% when

using multiple nano-satellites, compared to using a single physical nano-satellite.
In Figure 8.5c we want to evaluate the scalability of our system, to understand if it can manage the increased parallelization, maintaining the performance when the number of simulated and physical nano-satellites increases. Therefore, in this test, we simultaneously modify the number of simulated and physical nano-satellites. When the number of simulated nano-satellites exceeds 10 simulated nano-satellites, we assign to the new simulated nano-satellites the same 10 jobs defined in Table 8.5. On the x-axis, we can see the number of simulated and physical nano-satellites we evaluated. On the y-axis, we can see the percentage variation in the execution time of the simulation compared to the baseline, which in this case is set to 1 physical nano-satellite and 1 simulated nano-satellite. As we can see, in the beginning, the curve is flat because there is always a physical nano-satellite ready to process the job requested by a simulated nano-satellite. At a certain point, we can see an exponential increase in Figure 8.5c. This situation is attributed to the saturation of the processing cores on the machine that runs the simulation. In this case, the machine is forced to limit its concurrent processes, reducing the parallelization. However, when we reach 768 instances of simulated and physical nano-satellites, pictured by the blue X-mark in Figure 8.5c, the situation changes. At this point, the MQTT communication channel of the simulation becomes unreliable, leading to a loss of some MQTT packets by some threads. This results in an erroneous simulation. From Figure 8.5c we can also see a red X-mark after the blue one. This red X-mark represents the limit of the simulation in creating MQTT publishers, resulting again in an erroneous simulation.

In conclusion, we evaluate that our system is able to manage real data retrieved from a physical nano-satellite. We also evaluate that the fidelity of the data retrieved from the real nano-satellites improves as the sampling period decreases. This is verified when we do not overload the recording components. Lastly, we evaluate that our simulation is able to manage ∼700 physical nano-satellites simultaneously and that the scalability provided to our simulation is able to decrease the overall execution time of the simulation up to 88%. In the following, we analyze our conclusions and some possible future work directions.

9 | Conclusion and Future Works

In this thesis, we addressed such problems, and our contribution consists of the implementation of a HIL simulation for a constellation of nano-satellites, which is able to simulate the user-desired mission scenario. Moreover, the system is able to simulate radiation-induced errors and provide the end user with high-configuration capabilities, to the extent of modifying the physical nano-satellite in use. Additionally, we contribute by enabling our simulation to interface with multiple physical nano-satellites simultaneously and to exploit parallel execution, aiming to speed up the overall execution time of the simulation.

We evaluate our system under three metrics: realism, fidelity, and scalability. Firstly, in the realism metric, we verify if our simulation is able to retrieve and integrate real data from a physical nano-satellite. Then, the fidelity metric evaluates how much the variations in the sampling period impact the record of the non-functional data. Moreover, the scalability metric evaluates the scalability we provided to our system based on the number of jobs, the physical nano-satellites used, and the simultaneous variation of simulated and physical nano-satellites.

From the results obtained, we can assert that our system is capable of managing and integrating real data inside the simulation. Furthermore, we can say that by collecting more data, such as by using a lower sampling period, the nano-satellite is able to provide a higher fidelity for the data retrieved, since with a lower sampling period the nano-satellite is able to record peaks unseen by using higher sampling periods. Lastly, the evaluation conducted proves that our simulation is able to handle up to $\sim$700 physical instances, also leading to an 88% reduction in the total execution time of the simulation when using multiple nano-satellites, compared to using a single physical nano-satellite.

Our system has impacted the field of space missions enhancing the realism of the simulation of nano-satellite constellations because our system does not rely on artificial data to calculate the energy consumption of the simulated nano-satellites, but on real data of the consumes of a physical nano-satellite. By providing a way to calculate how the nano-satellites behave in the user-desired mission scenario, our system can help better understand the energy consumption of the constellation of nano-satellites before the ac-

tual deployment. Furthermore, the scalability of our system enables the simulation of large constellations of nano-satellites. For instance, since in our simulation, we are able to manage up to ∼700 physical nano-satellite simultaneously, we are able to simulate a space mission using nearly all the physical nano-satellites of the OneWeb constellation, which plans to deploy 720 nano-satellites [74]. Moreover, the ability to simulate errors caused by radiations, directly in the space mission scenario, provides a useful tool to test the reliability of the physical nano-satellite in the planning and development phase.
Several potential extensions or refinements to our system may include:

- Improve the scalability of our system to enable the use of more than ∼700 instances of physical nano-satellites.
- Modify the communication protocol between the two entities to increase the overall reliability of the communication channel when multiple physical nano-satellites are employed in the HIL simulation.
- Provide a user interface to ease the configuration of the system by the end user.
- Extend the system to simulate other aspects of the space environment, not taken under consideration in the scope of this thesis, such as space debris interaction with the nano-satellites, simulation of the atmospheric entry of the constellation of nano-satellites, and simulation of the communication between nano-satellites.

In conclusion, simulators are fundamental components in space missions, which help to increase the likelihood of success of every space mission. However, most space simulators do not support the ability to simulate the specific user-desired mission scenario and do not rely on real data, but on artificial one. In our work, we provide a HIL simulation of a constellation of nano-satellites, which is able to manage real data, retrieved from a real nano-satellite. The simulation is able to simulate the specific mission scenario desired by the user, and it can also simulate radiation-induced errors, in real time, on the real physical nano-satellite. Lastly, we enhance the scalability of our system, enabling it to be interconnected with multiple physical nano-satellites simultaneously. Our results prove that our simulation is able to integrate real data in real time. Furthermore, we empirically verify that the scalability brought to our system shortens by a factor of 88% the overall execution time of the simulation when using multiple physical nano-satellites, compared to using a single physical nano-satellite, and it also enables the simulation to manage ∼700 physical nano-satellites simultaneously.

Bibliography

[1] Communications in agi, . URL https://www.agi.com/missions/communications.

[2] Multi-domain in agi, . URL https://www.agi.com/missions/multi-domain.

[3] Mqtt. URL https://mqtt.org/.

[4] Median filter written in c. URL https://rosettacode.org/wiki/Median_filter#C.

[5] Gmat projects. URL https://opensource.gsfc.nasa.gov/projects/GMAT/index.php.

[6] Not black magic: Afsk. URL https://notblackmagic.com/bitsnpieces/afsk/#:~:text=AFSK%20stands%20for%20Audio%20Frequency,Packet%20Reporting%20System%20(APRS).

[7] Oneweb main page. URL https://oneweb.net/.

[8] Opensand main page. URL https://www.opensand.org/overview.html.

[9] Simpy project description, . URL https://pypi.org/project/simpy/.

[10] sgp4 project description, . URL https://pypi.org/project/sgp4/.

[11] Starlink main page, . URL https://www.starlink.com/.

[12] Starlink satellite constellation tracker, . URL https://satellitemap.space/?constellation=starlink.

[13] The butterfly effect. URL https://thedecisionlab.com/reference-guide/economics/the-butterfly-effect.

[14] What is omnet++? URL https://omnetpp.org/intro/.

[15] Introduction to simulation and modeling: Historical perspective. URL https://uh.edu/~lcr3600/simulation/historical.html#:~:text=The%20history%20of%20computer%20simulation,was%20too%20complicated%20for%20analysis.

[16] Southern ring nebula. URL https://webbtelescope.org/contents/media/images/2022/033/01G70BGTSYBHS69T7K3N3ASSEB.

[17] Radiation satellites unseen enemy - esa, Jul 2011. URL https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Radiation_satellites_unseen_enemy.

[18] Os3 - the open source satellite simulator, 2013. URL https://omnetpp.org/download-items/OS3.html.

[19] Image of reaching orbit - esa, 2020. URL https://www.esa.int/ESA_Multimedia/Images/2020/03/Reaching_orbit.

[20] Geostationary orbit - esa, Mar 2020. URL https://www.esa.int/ESA_Multimedia/Images/2020/03/Geostationary_orbit.

[21] Low earth orbit - esa, Mar 2020. URL https://www.esa.int/ESA_Multimedia/Images/2020/03/Low_Earth_orbit.

[22] Types of orbits - esa, Mar 2020. URL https://www.esa.int/Enabling_Support/Space_Transportation/Types_of_orbits.

[23] Reaching orbit - esa, Mar 2020. URL https://www.esa.int/ESA_Multimedia/Images/2020/03/Reaching_orbit.

[24] Polar and sun-synchronous orbit - esa, Mar 2020. URL https://www.esa.int/ESA_Multimedia/Images/2020/03/Polar_and_Sun-synchronous_orbit.

[25] Monitoring earth's surface - esa, Dec 2022. URL https://www.esa.int/Enabling_Support/Preparing_for_the_Future/Discovery_and_Preparation/Monitoring_Earth_s_surface.

[26] Esa developed earth observation missions - esa, Jun 2023. URL https://www.esa.int/ESA_Multimedia/Images/2019/05/ESA-developed_Earth_observation_missions.

[27] J. Aarem. Guide - sstv. URL https://norac.bc.ca/index.php/instruction-guides/730-guide-sstv.

[28] adis300. Adis300/filter-c: Elegant butterworth and chebyshev filter implemented in c, with float/double precision support. works well on many platforms. you can also use this package in c++ and bridge to many other languages for good performance., Oct 2019. URL https://github.com/adis300/filter-c.

[29] D. W. Alan Johnston, Jim McLaughlin and P. Kilroy. A new design for the amsat cubesat simulator. *The AMSAT journal*, 43(2):10–15, March/April 2020.

[30] J. M. Alan Johnston, Pat Kilroy and D. White. A guide to the amsat cubesatsim. *The AMSAT journal*, pages 7–31, September/October 2020.

[31] AMSAT. Amsat telemetry decoder manual, Oct 2022.

[32] M. Bacic. On hardware-in-the-loop simulation. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 3194–3198. IEEE, 2005.

[33] C. L. Bennett, M. Bay, M. Halpern, G. Hinshaw, C. Jackson, N. Jarosik, A. Kogut, M. Limon, S. Meyer, L. Page, et al. The microwave anisotropy probe* mission. *The Astrophysical Journal*, 583(1):1, 2003.

[34] J. Bennett. Embecosm/mibench: The mibench testsuite, extended for use in general embedded environments, Nov 2012. URL `https://github.com/embecosm/mibench`.

[35] W. Berg, S. T. Brown, B. H. Lim, S. C. Reising, Y. Goncharenko, C. D. Kummerow, T. C. Gaier, and S. Padmanabhan. Calibration and validation of the tempest-d cubesat radiometer. *IEEE Transactions on Geoscience and Remote Sensing*, 59(6): 4904–4914, 2020.

[36] A. Bjohnston. Release v1.0 · alanbjohnston/cubesatsim, Jun 2021. URL `https://github.com/alanbjohnston/CubeSatSim/releases/tag/v1.0`.

[37] A. Bjohnston. Home, Jun 2022. URL `https://github.com/alanbjohnston/CubeSatSim/wiki`.

[38] A. Bjohnston. Correspondences eps, Mar 2022. URL `https://github.com/alanbjohnston/CubeSatSim/wiki/7.-Main-Board-3`.

[39] A. Bjohnston. 2. software install, Mar 2022. URL `https://github.com/alanbjohnston/CubeSatSim/wiki/2.-Software-Install`.

[40] V. G. Bondur, K. A. Gordo, O. S. Voronova, and A. L. Zima. Satellite monitoring of anomalous wildfires in australia. *Frontiers in Earth Science*, 8:617252, 2021.

[41] B. Denby and B. Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–954, 2020.

[42] Eclipse. Paho mqtt, Aug 2022. URL `https://github.com/eclipse/paho.mqtt.cpp`.

[43] R. R. Ferreira, J. Da Rolt, G. L. Nazar, A. F. Moreira, and L. Carro. Adaptive low-power architecture for high-performance and reliable embedded computing. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 538–549. IEEE, 2014.

[44] R. Fisher, S. Perkins, A. Walker, and E. Wolfart. Median filter, 2003. URL `https://homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm`.

[45] M. Frąckiewicz. Continuous wave (cw) modulation, May 2023. URL `https://ts2.space/en/continuous-wave-cw-modulation/`.

[46] A. Froehlich. What is frequency-shift keying (fsk)?, Oct 2021. URL `https://www.techtarget.com/searchnetworking/definition/frequency-shift-keying`.

[47] R. Garner. Nasa's webb reveals cosmic cliffs, glittering landscape of star birth, Jul 2022. URL `https://www.nasa.gov/image-feature/goddard/2022/nasa-s-webb-reveals-cosmic-cliffs-glittering-landscape-of-star-birth`.

[48] M. González, H. Gilardi-Velázquez, S. Gutiérrez, and O. Ruíz-Martínez. Time management of modes of operation for survival of a satellite mission: Power simulation in matlab and stk. *IFAC-PapersOnLine*, 54(12):74–79, 2021.

[49] E. W. Gretok, E. T. Kain, and A. D. George. Comparative benchmarking analysis of next-generation space processors. In *2019 IEEE Aerospace Conference*, pages 1–16. IEEE, 2019.

[50] S. Gulkis, P. M. Lubin, S. S. Meyer, and R. F. Silverberg. The cosmic background explorer. *Scientific American*, 262(1):132–139, 1990.

[51] M. Harris. Tech giants race to build orbital internet [news]. *IEEE Spectrum*, 55(6): 10–11, 2018.

[52] M. G. Hauser, R. G. Arendt, T. Kelsall, E. Dwek, N. Odegard, J. L. Weiland, H. T. Freudenreich, W. T. Reach, R. F. Silverberg, S. H. Moseley, et al. The cobe diffuse infrared background experiment search for the cosmic infrared background. i. limits and detections. *The Astrophysical Journal*, 508(1):25, 1998.

[53] B. R. Johnson, C. J. Vourch, T. D. Drysdale, A. Kalman, S. Fujikawa, B. Keating, and J. Kaufman. A cubesat for calibrating ground-based and sub-orbital millimeter-wave polarimeters (calsat). *Journal of Astronomical Instrumentation*, 4(03n04):1550007, 2015.

[54] A. Johnston and P. Kilroy. The amsat cubesat simulator: A new tool for education and outreach, November/December 2018.

[55] A. Johnston and P. Kilroy. The new amsat cubesat simulator: Part 4, the ground station. *AMSAT J*, 42(4):21–27, 2019.

[56] A. Johnstone. Cubesat design specification rev. 14.1 the cubesat program. *Cal Poly SLO*, pages 1–34, 2022.

[57] J. Kalirai. Scientific discovery with the james webb space telescope. *Contemporary Physics*, 59(3):251–290, 2018.

[58] S. Kassing, D. Bhattacherjee, A. B. Águas, J. E. Saethre, and A. Singla. Exploring the "Internet from space" with Hypatia. In *ACM IMC*, 2020.

[59] E.-J. Kim, E.-S. Sim, and H.-D. Kim. Development of the power simulation tool for energy balance analysis of nanosatellites. *Journal of Astronomy and Space Sciences*, 34(3):225–235, 2017.

[60] KU2Y. The radio amateur satellite corporation, Jul 2021. URL `https://www.amsat.org/amsat-cubesatsim-first-official-release-v1-0/`.

[61] E. Kulu. What is a cubesat? URL `https://www.nanosats.eu/cubesat`.

[62] J. Ledin. Hardware in the loop simulation, embedded syst, 1999.

[63] J. Low. Satellite ground track visualizer, Apr 2019. URL `https://observablehq.com/@jake-low/satellite-ground-track-visualizer`.

[64] E. Mabrouk. What are smallsats and cubesats?, Mar 2015. URL `https://www.nasa.gov/content/what-are-smallsats-and-cubesats`.

[65] P. Mairota, B. Cafarelli, R. Labadessa, F. Lovergine, C. Tarantino, R. M. Lucas, H. Nagendra, and R. K. Didham. Very high resolution earth observation features for monitoring plant and animal community structure across multiple spatial scales in protected areas. *International Journal of Applied Earth Observation and Geoinformation*, 37:100–105, 2015.

[66] Mathuranathan. Bpsk - binary phase shift keying, Apr 2010. URL `https://www.gaussianwaves.com/2010/04/bpsk-modulation-and-demodulation-2/`.

[67] J. C. McDowell. The low earth orbit satellite population and impacts of the spacex starlink constellation. *The Astrophysical Journal Letters*, 892(2):L36, 2020.

[68] J. Mehta. Space grade electronics: How nasa's juno survives near jupiter, Apr 2018. URL https://www.planetary.org/articles/0417-space-grade-electronics.

[69] A. Nathwani. Abhijitnathwani/image-processing: Image processing codes using c, without the use of any external libraries. the codes in this repository apply traditional image processing algorithms with use of plain c language, which is almost run everywhere., Apr 2021. URL https://github.com/abhijitnathwani/image-processing/tree/master.

[70] S. Nilsen. Bvssvni/fwht: Fast walsh hadamard transform in c, Sep 2012. URL https://github.com/bvssvni/fwht.

[71] Nsnam. What is ns-3. URL https://www.nsnam.org/about/what-is-ns-3/.

[72] J. Perez. Why space radiation matters, Apr 2017. URL https://www.nasa.gov/analogs/nsrl/why-space-radiation-matters.

[73] G. S. Phones. The cost of building and launching a satellite, 2015.

[74] J. Radtke, C. Kebschull, and E. Stoll. Interactions of the space debris environment with mega constellations—using the example of the oneweb constellation. *Acta Astronautica*, 131:55–68, 2017.

[75] A. Ridgeway. Stk for cubesats.

[76] C. Rocchini. Mandelbrot set written in c. URL https://rosettacode.org/wiki/Mandelbrot_set#C.

[77] E. T. SatNow. Home, Aug 2022. URL https://www.satnow.com/community/what-do-you-mean-by-medium-earth-orbit-meo.

[78] P. Shah and A. Lai. Cots in space: From novelty to necessity. In *35th Annual Small Satellite Conference*, 2021.

[79] B. R. Smalarz. Cubesat constellation analysis for data relaying. 2011.

[80] D. Weitz. Introduction to simulation with simpy, Feb 2022. URL https://towardsdatascience.com/introduction-to-simulation-with-simpy-b04c2ddf1900.

[81] Zavalyshyn. Zavalyshyn/chaosduck: An automatic fault-injection tool used to evaluate software security and fault-tolerance, Dec 2020. URL https://github.com/zavalyshyn/chaosduck.

A | Appendix A

In this appendix, we analyze the encoding of a FSK/DUV message of the CubeSatSim nano-satellite. As encoded message, we used the following message:
FSK/DUV message encoded

```
20230404144911,7,67,4092,1,0,0,442,2052,2050,2141,1823,279,356,338,
408,202,287,150,2048,2048,2048,2048,2048,2048,505,2048,1001,992,0,
2048,424,2048,2048,2048,174,2268,2294,2048,0,0,0,0,1,0,0,1,0,0,1,0
```

This message is decoded in Table A.1 and in Table A.2.

FSK/DUV message decoded

	Value	Real value	Description
1	*20230404144911*	*Same value*	Timestamp of the packet
2	*7*	*Same value*	Fox-ID (7: FSK 200bps, 99: BPSK 1200bps)
3	*67*	*Same value*	Amount of time the device has been reset
4	*4092*	*Same value*	Time since the device is ON
5	*1*	*Same value*	Telemetry type (1: FSK, 2: BPSK)
6	*0*	$Value/100$	Voltage battery A, always 0 (V)
7	*0*	$Value/100$	Voltage battery B, always 0 (V)
8	*442*	$Value/100$	Total Voltage Battery A+B+C cells (V)
9	*2052*	$(Value - 2048)/100$	Acceleration of CubeSatSim on X axis (g)
10	*2050*	$(Value - 2048)/100$	Acceleration of CubeSatSim on Y axis (g)
11	*2141*	$(Value - 2048)/100$	Acceleration of CubeSatSim on Z axis (g)
12	*1823*	$Value - 2048$	Current of the battery (mA)
13	*279*	$Value/10$	BME280 temperature sensor (°C)
14	*356*	$Value/100$	Voltage on solar panel +X (V)
15	*338*	$Value/100$	Voltage on solar panel -X (V)
16	*408*	$Value/100$	Voltage on solar panel +Y (V)
17	*202*	$Value/100$	Voltage on solar panel -Y (V)
18	*287*	$Value/100$	Voltage on solar panel +Z (V)
19	*150*	$Value/100$	Voltage on solar panel -Z (V)
20	*2048*	$Value - 2048$	Current received from solar panel +X (mA)
21	*2048*	$Value - 2048$	Current received from solar panel -X (mA)
22	*2048*	$Value - 2048$	Current received from solar panel +Y (mA)
23	*2048*	$Value - 2048$	Current received from solar panel -Y (mA)
24	*2048*	$Value - 2048$	Current received from solar panel +Z (mA)
25	*2048*	$Value - 2048$	Current received from solar panel -Z (mA)
26	*505*	$Value/100$	Voltage of the PSU (V)
27	*2048*	$(Value - 2048)/100$	Calculated spin rate using solar cells

Table A.1: First table containing the description of the parameters of a FSK/DUV message of the CubeSatSim.

	Value	Real value	Description
28	*1001*	*Same value*	BME280 pressure sensor (hPa)
29	*992*	$Value/10$	BME280 altitude sensor (m)
30	*0*	*Same value*	Resets
31	*2048*	$Value - 2048$	Received Signal Strength Indicator (RSSI)
32	*424*	$Value/10$	Internal temperature of PI (°C)
33	*2048*	$Value - 2048$	Angular velocity of CubeSatSim X axis (dps)
34	*2048*	$Value - 2048$	Angular velocity of CubeSatSim Y axis (dps)
35	*2048*	$Value - 2048$	Angular velocity of CubeSatSim Z axis (dps)
36	*174*	$Value/10$	BME280 humidity sensor (hPa)
37	*2268*	$Value - 2048$	Current of the PSU (mA)
38	*2294*	$(Value - 2048)/10$	Diode temperature sensor (°C)
39	*2048*	$Value - 2048$	Sensor2
40	*0*	*Same value*	Status of the STEM Payload board (0: OK, 1: FAIL)
41	*0*	*Same value*	CubeSatSim in safe mode. Enabled when the voltage of the battery is low (0: OFF, 1: ON)
42	*0*	*Same value*	CubeSatSim simulate the telemetry (0: OFF, 1: ON)
43	*0*	*Same value*	Status of the payload (0: OK, 1: FAIL)
44	*1*	*Same value*	Status of i2C-0 bus (0: OK, 1: FAIL). No longer used in this design [29]
45	*0*	*Same value*	Status of i2C-1 bus (0: OK, 1: FAIL)
46	*0*	*Same value*	Status of i2C-3 bus (0: OK, 1: FAIL)
47	*1*	*Same value*	Status of connection to the camera (0: OK, 1: FAIL)
48	*0*	*Same value*	Amount of command received from the ground station. Receiver not implemented [30]
49	*0*	*Same value*	Status of the receiver (0: Stowed, 1: Deployed). Receiver not implemented [30]
50	*1*	*Same value*	Status of transfer antenna (0: Stowed, 1: Deployed)
51	*0*	*Same value*	Pad

Table A.2: Second table containing the description of the parameters of a FSK/DUV message of the CubeSatSim.

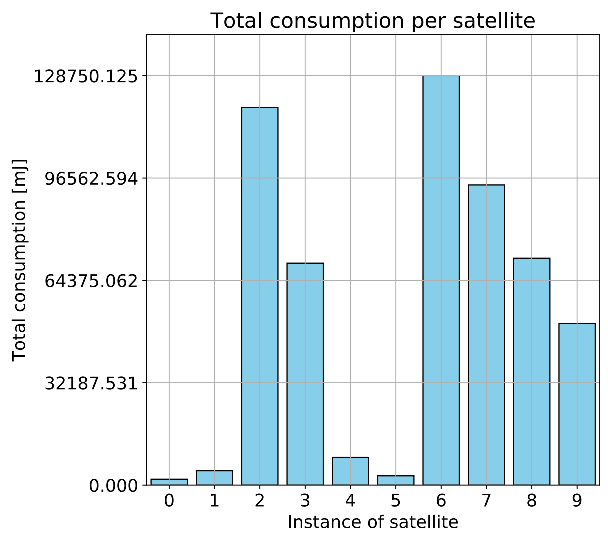
B | Appendix B

In this appendix, we analyze all the plots that our HIL simulation generates. The plots generated cover different aspects of the simulation, and we analyze these plots in separate sections.

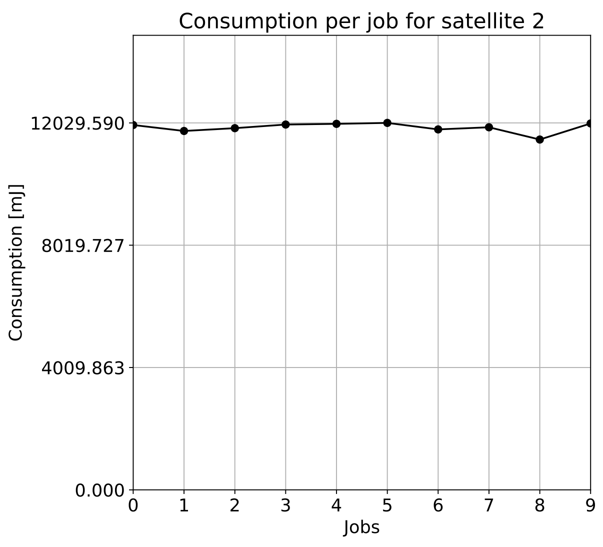
In Section B.1 we analyze the plots concerning the energy consumption of the nano-satellite while performing the various tasks. Furthermore, in Section B.2 we describe the plots concerning the time taken by the different tasks to execute. Moreover, in Section B.3 we describe the plots regarding the simulation of the errors induced by radiation. Additionally, in Section B.4 we analyze the plots that are more focused on the general working of the jobs. Lastly, in Section B.5 we describe the plots concerning the operational states covered by the simulated nano-satellites.

The following plots are generated by running two standard HIL simulations. The manner is configured with the following parameters: 10 jobs, 9 instances of nano-satellites, and radiation-induced errors with bit-flipping. The latter is configured with the following parameters: 10 jobs, and 9 instances of nano-satellites. We decided to split the plots because we prefer to show the plots that are not inherent to errors, in a correct simulation without error-induced errors. Most of the following plots are generated by the simulation as bar plots. The remaining are continuous plots, except for two plots shown in Figure B.4c and in Figure B.5a, which use mixed types of plots and are analyzed in their corresponding sections. We use bar plots, when there is no continuity in the scenario represented by the plot. For instance, when we represent the plot of the total energy consumption, shown in Figure B.1a. We use continuous plots when there is continuity in the scenario represented by the plot. For instance, when we represent the plot of the energy consumption per job, shown in Figure B.1b.

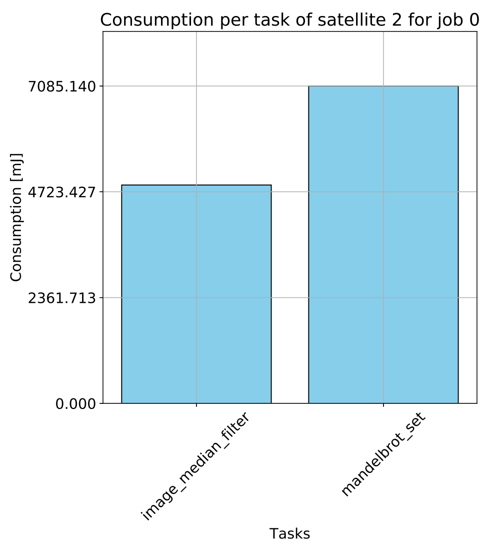
B.1. Energy Consumption



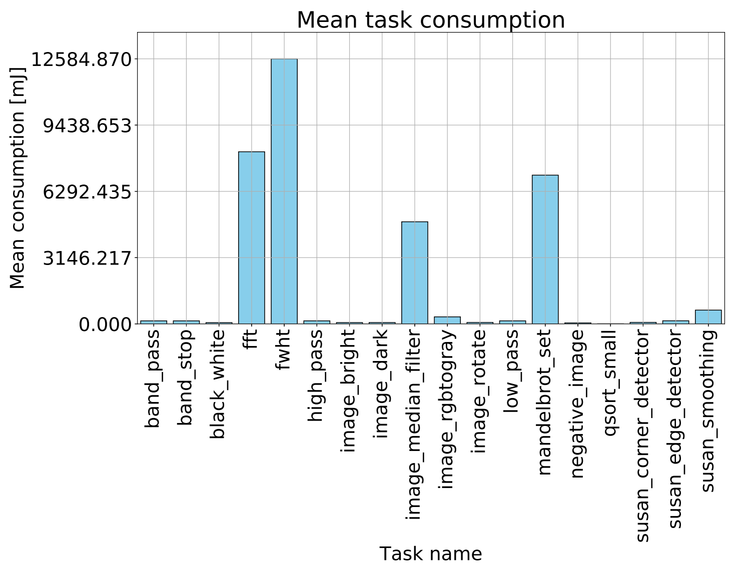
(a) Total consumption.



(b) Consumption per job.



(c) Consumption per task.



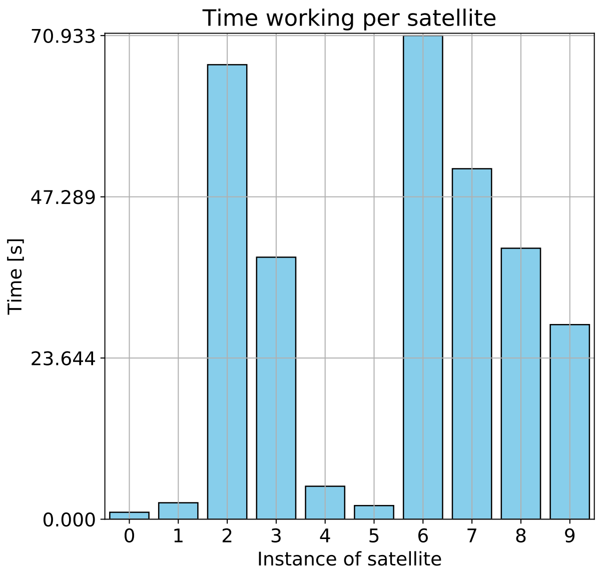
(d) Mean consumption per task.

Figure B.1: Consumption plots generated by our HIL simulation.

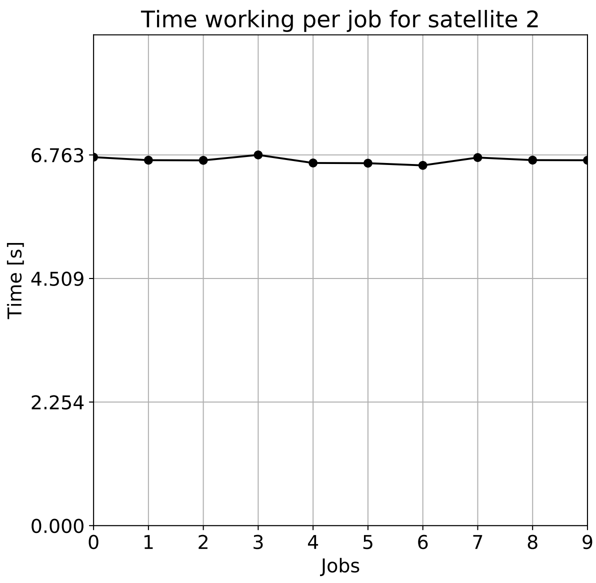
These plots show the amount of energy consumption in the execution of the tasks on the nano-satellite. The plots are generated at different levels of granularity.
First, the *total consumption* plot, shown in Figure B.1a, analyzes the energy consumption of all the simulated nano-satellites along the whole simulation. Furthermore, the *consumption per job* plot, shown in Figure B.1b, examines the energy consumption of

all the jobs in a nano-satellite. This plot is generated for every simulated nano-satellite. In addition, we can see the *consumption per task* plot in Figure B.1c, which examines the energy consumption of all tasks in a job. This plot is generated for every job of every simulated nano-satellite. Lastly, the *mean consumption per task* plot, illustrated in Figure B.1d, analyzes the mean energy consumption used by every task involved in the current execution of the simulation.

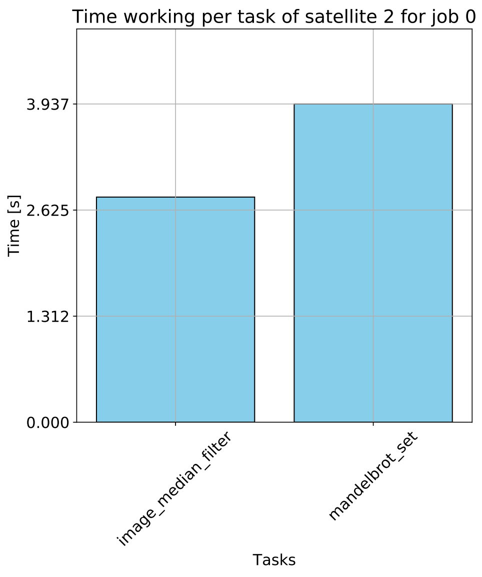
B.2. Time



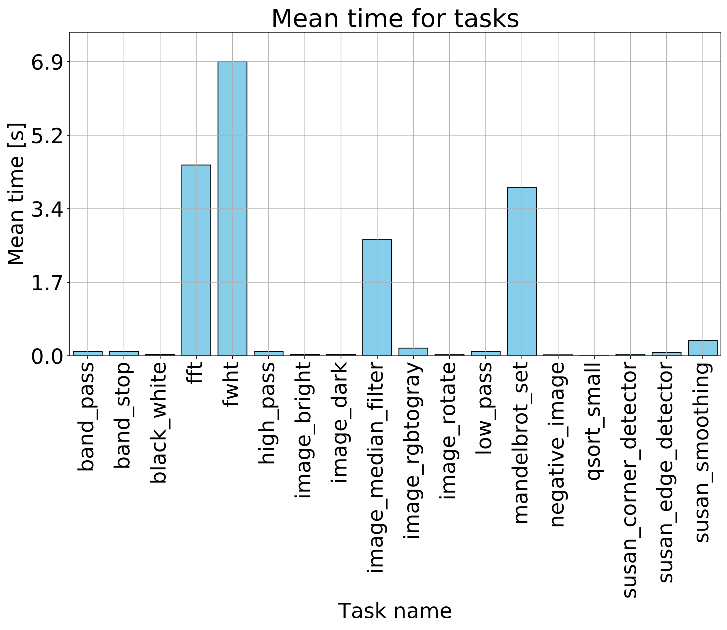
(a) Total time.



(b) Time per job.



(c) Time per task.



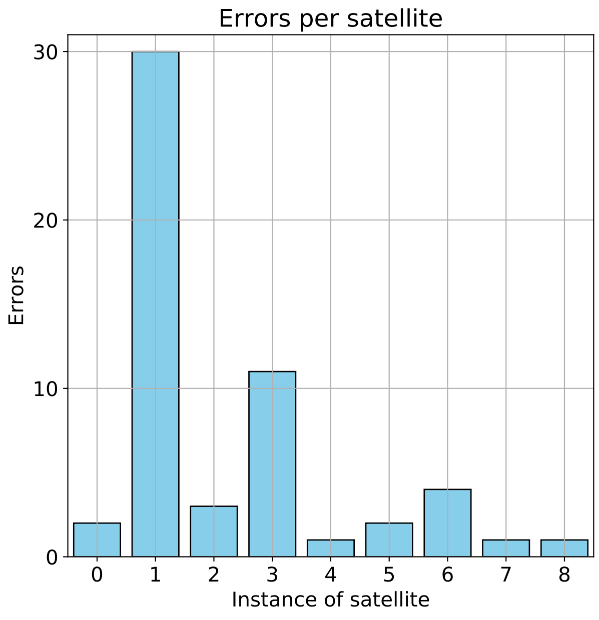
(d) Mean time per task.

Figure B.2: Time plots generated by our HIL simulation.

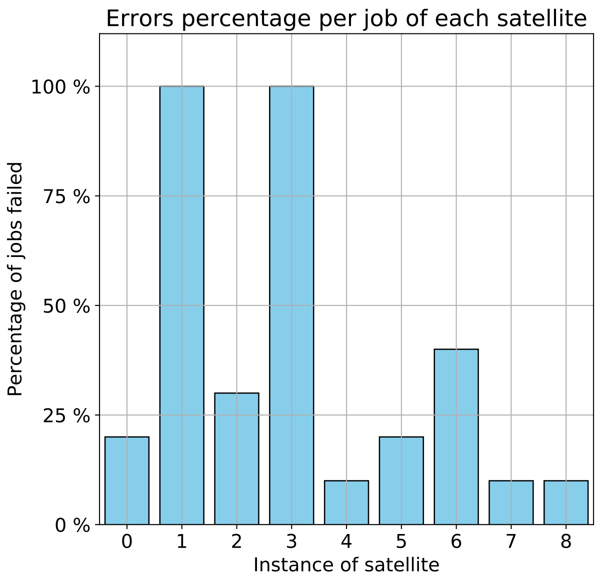
These plots show how long it takes for the tasks that simulate the processing to execute on the nano-satellite. The plots are generated at different levels of granularity.
First, the *total time* plot, shown in Figure B.1a, analyzes the total time used to process all

the jobs for all the simulated nano-satellites. Furthermore, the *time per job* plot, shown in Figure B.1b, examines the differences in execution time of all the jobs in a nano-satellite. This plot is generated for every simulated nano-satellite. In addition, we can see the *time per task* plot in Figure B.1c, which examines for each job of each nano-satellite the time used by every task to end its execution. This plot is generated for every job of every simulated nano-satellite. Lastly, the *mean time per task* plot, illustrated in Figure B.1d, analyzes on average the execution times of every task involved in the current execution of the simulation.

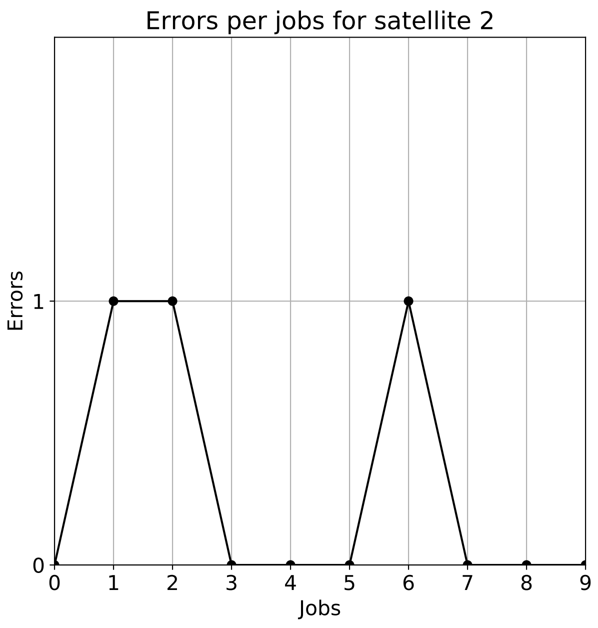
B.3. Radiation-Induced Errors



(a) Errors per satellite.



(b) Errors percentage per satellite.



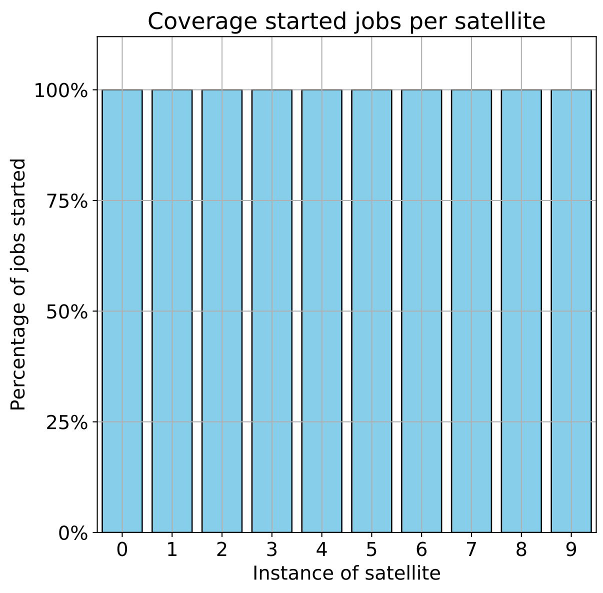
(c) Errors per job.

Figure B.3: Radiation-induced error plots generated by our HIL simulation.

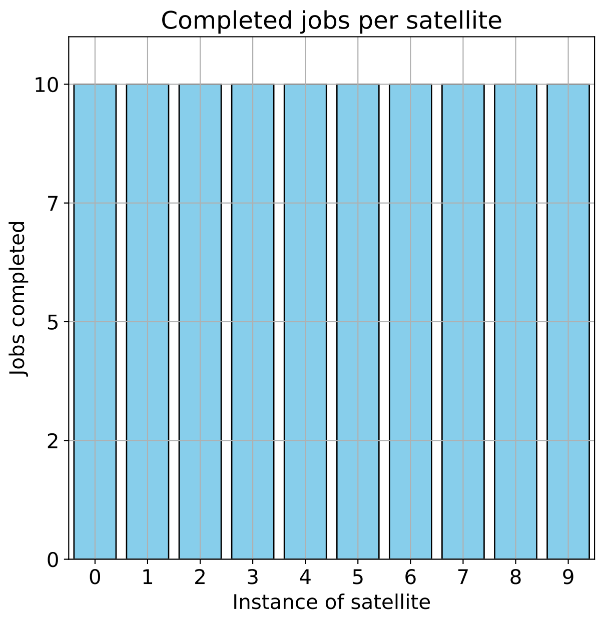
These plots analyze the behavior of the radiation-induced errors in our HIL simulation. The plots are generated at different levels of granularity.

First, the *errors per satellite* plot, shown in Figure B.3a, analyzes the total errors of each simulated nano-satellite. Furthermore, the *errors percentage per satellite* plot, shown in Figure B.3b, examines for each simulated nano-satellite the percentage of jobs that failed at least one task. Lastly, the *errors per job* plot, illustrated in Figure B.3c, examines the radiation-induced errors of every job of every simulated nano-satellite. This plot is generated for every simulated nano-satellite.

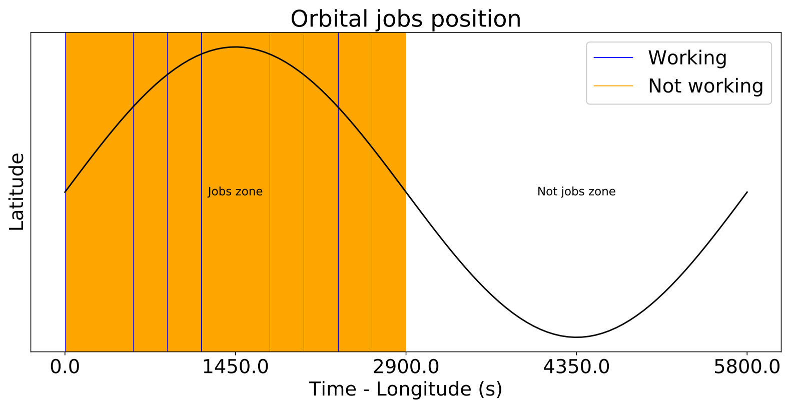
B.4. Jobs



(a) Coverage.



(b) Jobs completed.

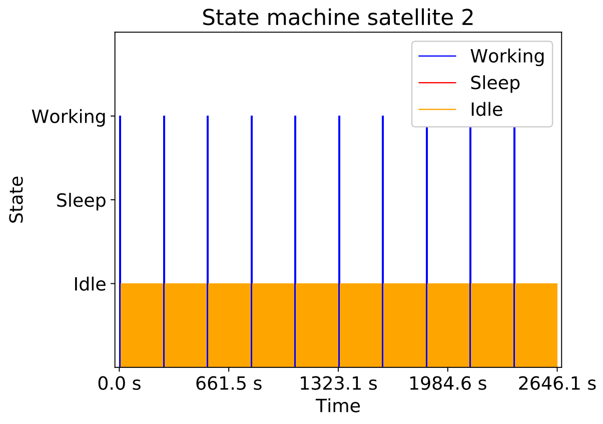


(c) Jobs position.

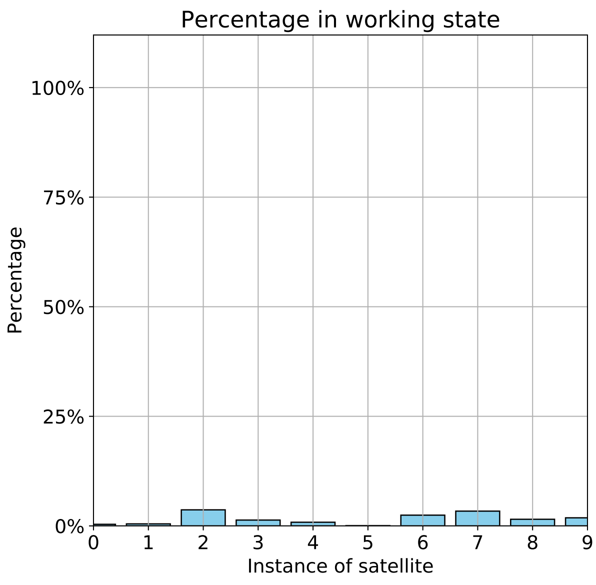
Figure B.4: Job plots generated by our HIL simulation.

These plots show the behavior of the jobs used in the execution of the simulation.
First, the *coverage* plot, shown in Figure B.4a, analyzes the percentage of jobs started for each simulated nano-satellite. We calculate the percentage of jobs because, since the jobs are located along the orbit of the simulated nano-satellites, if the simulation needs to simulate a high amount of jobs, some jobs could have similar orbital positions, and if a job executes for a long time, the simulation skips some consecutive jobs. Furthermore, the *jobs completed* plot, shown in Figure B.4b, examines the number of jobs completed for each simulated nano-satellite. Lastly, the *jobs position* plot, illustrated in Figure B.4c, is a more particular plot. In this plot, we examine how the jobs are accessed and executed along the orbit. In this plot, the sinusoid represents the orbit of the simulated nano-satellite, while the colored areas represent what the simulated nano-satellite was doing in the portion of the orbit. This plot is generated for every simulated nano-satellite.

B.5. Operational State



(a) State.



(b) State percentage.

Figure B.5: Operational state plots generated by our HIL simulation.

These plots show the operational state of the simulated nano-satellites during the simulation. The various operational states we used in our simulation are: "sleep", "idle", and "working". The "sleep" state is not visible because the nano-satellite remains in this state for a really short time. There is an additional state, which is the "off" state, which is not considered an operational state and, therefore is not visualized in these plots.

The former plot, which is the *state* plot shown in Figure B.5a, analyzes the time consumed by every nano-satellite in the different operational states. This plot is composed of different colored areas of different heights, and both the color and the height of an area represent a different operational state. This plot is generated for every instance of nano-satellite. The latter plot, which is the *state percentage* plot shown in Figure B.5b, examines the percentage of time in which the simulated nano-satellites remain in the different operational states, specifically in Figure B.5b we examine the working time. The values in Figure B.5b are extremely low because we need to consider that the orbit of the nano-satellites lasts 2900 [s], and using only 10 jobs the nano-satellites only work for a small fraction of the orbit. By configuring the simulation with a higher number of jobs, the percentages of time working for the nano-satellites are higher. In conclusion, the time in the plots is sometimes different from 2900 [s] because the simulated nano-satellites go into "off" mode sometimes, and that fraction of time is not considered in this plot. This plot is generated for all the operational states used to execute our simulation.