GEO-CLOUD: MODELLING AND IMPLEMENTATION OF THE GEO-CLOUD EXPERIMENT FOR THE FED4FIRE EUROPEAN PROJECT

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA
EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Geo-Cloud: Modelling and Implementation of the Geo-Cloud Experiment for the Fed4FIRE European Project

Rubén Pérez Pascual

Septiembre, 2014

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

Geo-Cloud: Modelling and Implementation of the Geo-Cloud Experiment for the Fed4FIRE European Project

Autor: Rubén Pérez Pascual
Director: Dr. Jonathan Becedas Rodríguez
Tutor: Dr. Carlos González Morcillo

Septiembre, 2014

Rubén Pérez Pascual

Ciudad Real – Spain

E-mail: Ruben.Perez@alu.uclm.es
Web site: `www.linkedin.com/pub/ruben-pérez-pascual/53/403/639`

This work was carried out with the support of the Fed4FIRE-project (“Federation for FIRE"), an Integrated Project receiving funding from the European Union’s Seventh Framework Programme for research, technological development and demonstration under grant agreement no 318389. It does not necessarily reflect the views of the European Commission. The European Commission is not liable for any use that may be made of the information contained herein.

TRIBUNAL:

Presidente:

Secretario:

Vocal:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	**SECRETARIO**	**VOCAL**
Fdo.:	Fdo.:	Fdo.:

Abstract

PROCESSING and distribution of big space data still present a critical challenge: the treatment of massive and large-sized data obtained from Earth Observation (EO) satellite recordings. Remote sensing industries implement on-site conventional infrastructures to acquire, store, process and distribute the geo-information generated. However these solutions do not cover sudden changes in the demand of services and the access to the information presents large latencies.

In this work we present the detailed design, architecture and implementation of the GEO-Cloud experiment to value if future internet technologies can be used to overcome the previously defined limitations.

The whole system implements a complete highly demanding EO system making use of future internet technology and cloud computing to define a framework to make EO industry more competitive and adaptable to the new requirements of massive data processing and instant access to satellite information.

Contents

List of Tables

List of Figures

Listings

List of acronyms

AOI	Area of Interest
API	Application Programming Interface
BSD	Berkeley Software Distribution
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSV	Comma Separated Values
CSW	Web Catalogue Service
DBMS	Database Management System
EaaS	Elasticity as a Service
EO	Earth Observation
FP7	Seventh Framework Programme
FTP	File Transfer Protocol
GNU	GNU's Not Unix
GPL	General Public License
GSD	Ground Sample Distance
GPS	Global Positioning System
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ICE	Internet Communications Engine
ICMP	Internet Control Message Protocol
IP	Internet Protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LAN	Local Area Network
LEO	Low Earth Orbit

LGPL	Lesser General Public License
LTAN	Local Time Ascending Node
LTE	Long Term Evolution
NFS	Network File System
NASA	National Aeronautics and Space Administration
OGC	Open Geospatial Consortium
OMF6	Object Monitoring Framework v6
OML	Object Monitoring Language
OSI	Open System Interconnection
PaaS	Platform as a Service
QoS	Quality of Service
RPC	Remote Procedure Call
RSA	Rivest, Shamir and Adleman
RTT	Round Trip Time
SaaS	Software as a Service
SCP	Secure Copy
SDL	Simple DirectMedia Layer
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SRTM	Shuttle Radar Topography Mission
SSH	Secure SHell
SSL	Secure Sockets Layer
SSO	Sun Synchronous Orbit
STK	Satellite Tool Kit
TCP	Transport Control Protocol
UAV	Unmanned Aerial Vehicles
UDP	User Datagram Protocol
UML	Universal Modelling Language
USGS	U.S. Geological Survey
WCS	Web Coverage Service
WFS	Web Feature Service
WMS	Web Map Service

WPS	Web Processing Service
XML	eXtensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Acknowledgements

No tengo suficientes dedos en las manos, de momento, para agradecer a todas las personas que debería, el que yo haya sido capaz de llegar hasta donde he llegado. Ruego me disculpen si de alguien me olvidé.

Hay muchísimas personas importantes en mi vida, pero destacan por encima de todos mis padres y hermana. No haría falta ni mencionar que sin su apoyo en malos y buenos momentos, han hecho de mí el hombre que siempre he deseado ser con la carrera profesional que siempre deseé tener.

A Ana por supuesto que es la que me ha soportado y apoyado estos, de momento, tres años; aunque aún la física no logre explicarlo, su fuerza y alegría me eran transmitidos.

Cómo olvidarse de los amigos del campus, sobre todo los que prácticamente he convidido con ellos pasando multitud de buenos y malos momentos. Nombres propios como Antonio, Ángel, Jesús Ángel y Javier entre otros, tienen mi respeto y agradecimientos ya que en poco tiempo, se han convertido en importantes personas en mi vida y juntos compartimos el peso que el llegar hasta este punto acarrea.

A todos mis amigos del grupo *ARCO* y antigua *Oreto*, que siempre me han solucionado las pequeñas adversidades tecnológicas y me han enseñado más de lo que ellos creen. Gracias Luismi, Tato y Manu... y demás compañeros.

Especial reconocimiento merecen mis compañeros de trabajo en *Deimos Castilla La Mancha*, por su apoyo y preocupación diaria. En especial Gerardo, compañero y amigo por el que muchos me envidiarán.

Especiales menciones a Carlos González Morcillo y Jonathan Becedas Rodríguez, tutor y director respectivamente. Ambas son dos personas comprometidas y muy profesionales, pero además de eso quiero agradecerles su trabajo y amistad. Carlos, simplemente brillante y mejor persona, me ha brindado una oportunidad por la siempre le estaré agradecido. Jonathan, carismático, de los mejores profesionales que he conocido y con espíritu indomable, siempre apostó por mí y me presentó diversas experiencias investigadoras.

Por todo lo anterior, gracias. Sin vosotros, estos cinco años no habrían sido lo mismo.

Rubén Pérez Pascual

A mi familia, a Carlos que confió en mí y me dió a conocer y a mis compañeros de Deimos Castilla-La Mancha, especialmente a mis compañeros espartanos de I+D+i.

Chapter 1

Introduction

EARTH Observation (EO) commercial data sales have increased a 550% in the last decade. This area is considered a key element in the space industry and an opportunity market for the next years [Eur10].

EO industries implement on-premises conventional infrastructures to acquire, store, process and distribute the geo-information generated.

However these solutions have the risks of over/under size the infrastructure, they are not flexible to cover sudden changes in the demand of services and the access to the information presents large latencies. These aspects limit the use of EO technology for real time use such as to manage crises, natural disasters and civil security among others [Der07].

In addition, new sectors and user typologies are applying for new EO services and there is an increasing demand of this services. These users need more flexible, easy and instant access to EO products and services through the Web. This demand has traditionally been driven through Space Data Infrastructures and heavy standards (ISO TC/211 and Open Geospatial Consortium (OGC)) which are focused on interoperability rather than the real demand from the end-users.

The use of cloud computing technology can overcome the previously defined limitations that present conventional infrastructures because of its elasticity, scalability and on-demand use characteristics [Amb10].

The GEO-Cloud experiment goes beyond conventional data infrastructures used in the EO industry and beyond the implementations of applications running in cloud, to quest which parts of a complete infrastructure of EO are technologically and economically viable to be virtualized to offer basic and high added value services (see Figure 1.1).

The GEO-Cloud experiment is is divided into two sub-experiments [PGB+14]:

- One experiment in a system integrated in *Virtual Wall* and *BonFIRE* cloud emulating the whole EO system. In *Virtual Wall* , the constellation of satellites and the ground stations are simulated. In the *BonFIRE* cloud, the novel architecture for EO images processing is implemented. Both *Virtual Wall* and *BonFIRE* are interconnected for simulating the data transfers between the satellites and ground stations and between

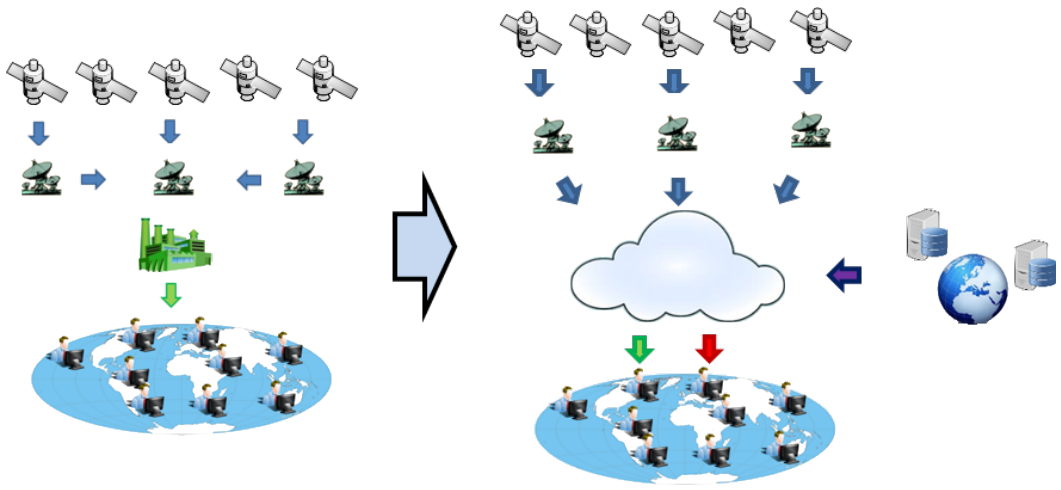


Figure 1.1: The GEO-Cloud Concept. It is based on the use of cloud technology to acquire data, store it, process it, integrate it with other sources and distribute it to end users with the final objective of testing viable solutions for its real implementation.

the ground stations and the cloud platform.

- One experiment based in *PlanetLab* . The experiment consists of real networks interconnected around the world, which are used to emulate the real behaviour of the links between the ground stations and the cloud and from the cloud to the end users. The network bandwidth, loss-rate and delay were measured. Those parameters, once measured were used to update the parameters of the network models implemented in *Virtual Wall* .

GEO-Cloud emulates the remote sensing mission with emulated satellites, the topology network and the communications in the *Virtual Wall* testbed. The data acquired from the emulated satellites is transferred to the *BonFIRE* cloud for storage, processing and distribution to end users. End users accessing and broadcasting will be emulated in another network implemented in *Virtual Wall* . In order to implement realistic impairments in *Virtual Wall* , real networks will be tested in *PlanetLab* . This project was accomplished in *Elecnor Deimos Satellite Systems* located in Puertollano.

1.1 Earth Observation

During Wold War I, the EO was born. Reconnaissance aircrafts flew over enemies in order to follow the army movements [ESA09]. These aircrafts embedded cameras which allowed knowing the position and strength of enemy forces. Also, during the World War II, the technique hugely evolved. This time, aerial photographs were used to map coastal conditions by measuring waves near of the coast. Besides, infrared spectral bands were used in order to perceive green vegetation and distinguish it from camouflage nets. The film sensitive to wavelengths involved a huge advance for EO. Last decade, technological advances in remote

sensing field were carried out. Spacecrafts, airplanes, and several technology elements aggregated by a space mission were designed, built and launched to retrieve information about the Earth, other planets and galaxies.

EO is a multidisciplinary field involving several technical areas such as Computer Science, Optics, Chemistry, Mathematics, Materials, Telecommunications, Aerospace and Physics among others. The physical, chemical and biological information about the Earth are gathered using several manners of obtaining it.

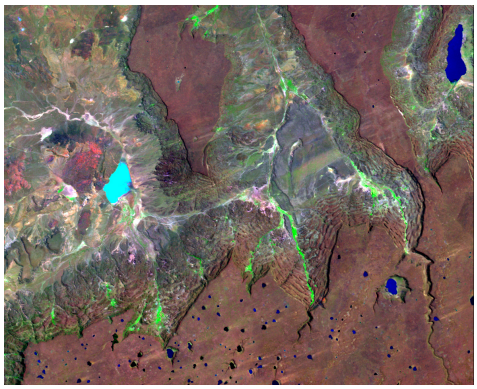
The information from the Earth surface can be obtained by both active and passive [NAS] methods.

- Active remote sensors emit microwaves toward the Earth's surface in order to scan objects and areas. These waves reflect off the surface and return to the sensor. This imaging method is also known as active microwave. There are three types of active remote sensing: imaging radar [NSI], which takes images in the same way that a camera does, depending on the type of surface where the rays reflect; non-imaging radar which measures the amount of reflected energy; and altimetry sensor, which sends microwaves to the Earth's surface and measures the time that these microwaves take to return to the sensor.

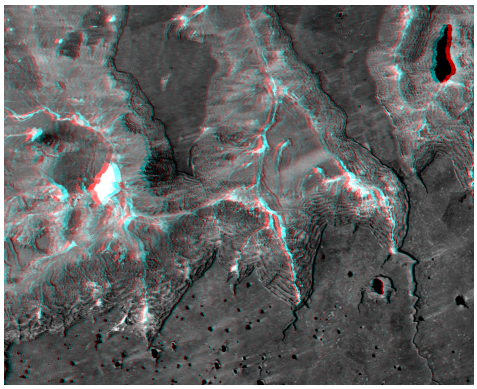
- Passive remote sensors use the radiation provided by some objects and their surrounding areas. The most common source of radiated energy, is the reflected sunlight over the Earth's surface. Passive remote sensors are radiometers, for measuring the electromagnetic radiation; photography, to acquire the light reflected by the chemical elements in the visible spectral band; charge-coupled devices, to acquire ultraviolet spectrum, and infrared sensors, which obtain thermal images (see Figure 1.2). Moreover, the EO acquisition is also categorized by three types of resolution in the images acquired [San09]: spectral , geometric and temporal resolutions. The spectral resolution depends on the object to be sensed. Panchromatic, blue, green , red, near-infrared and thermal infrared are the most used spectral bands.

The geometric resolution is selected depending on how much definition the telescopes or imaging sensors have. Some examples are the acquisition of a cloud for weather applications, about $100\ km$ resolution, or monitoring of roads and highways for traffic monitoring purpose, under $1\ m$ resolution (see Figure 1.3).

The temporal resolution is the related to the visit frequency between two consecutive acquisitions of the same location (see Figure 1.4). This is known in remote sensing as revisit time. Depending on the image application, the revisit time can vary. For example, for environmental monitoring, geology or precision agriculture the revisit time is longer that the required revisit time for military or maritime surveillance [San09].



(a) Image acquired in blue, near infrared and short wave infrared spectral bands.



(b) Image acquired from SRTM conforming a 3D image.

Figure 1.2: Different images acquired by USGS/NASA Landsat.

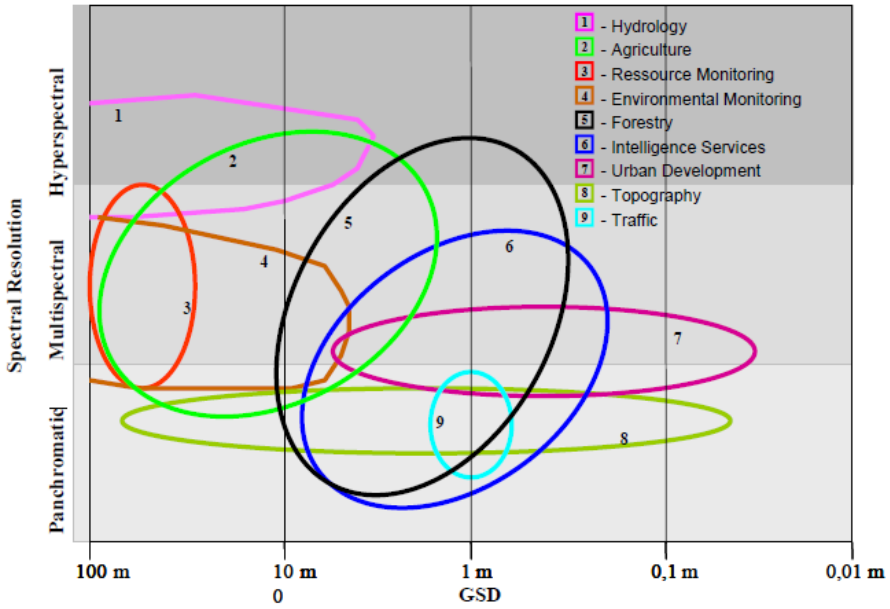


Figure 1.3: Resolutions of different sensing applications.

The main component to carry out EO imaging consists of the platform.There, the payloads for image acquisition are installed. These platforms can be summarized in two groups: spacecrafts and aircrafts. Aircrafts involve aerodines (planes, and helicopters among others) and aerostats (balloons and dirigibles). Spacecrafts are systems designed to fly over the atmosphere and could be used for many different purposes such as communications, remote sensing, meteorology and planetary exploration among others.

1.2 Cloud Computing

Traditional business applications have always been too expensive and complex. The number and variety of necessary hardware and software to run them is overwhelming. A team of

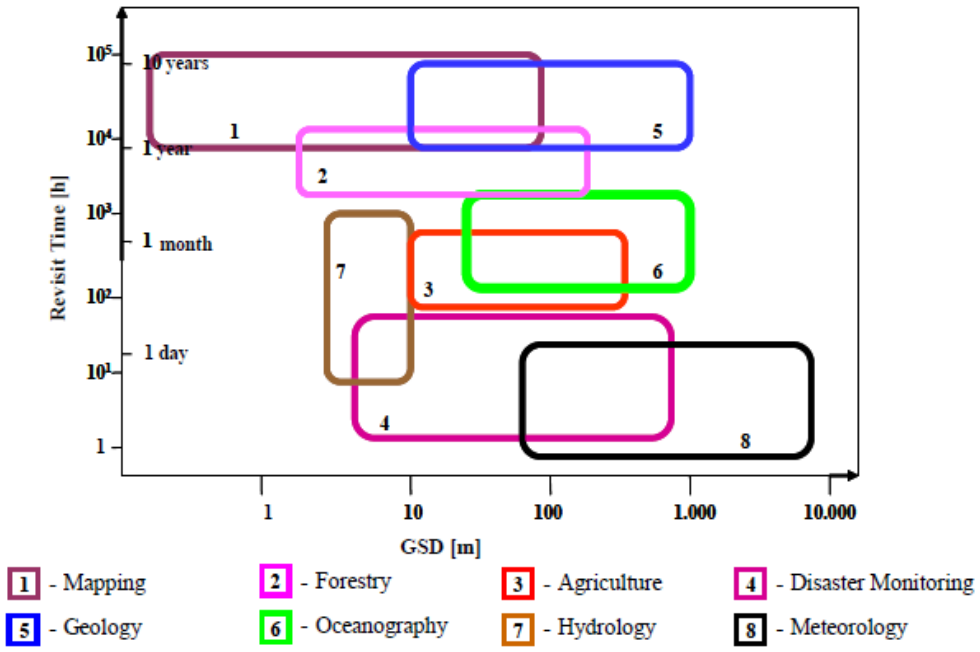


Figure 1.4: GSD vs Revisit Time.

experts that can install, configure, test, run, secure, and update is needed in most of cases. Thanks to Cloud Computing, these complexities do not exist because it is not necessary to manage the hardware and software to run the application: it is the responsibility of an experienced cloud provider. The shared infrastructure makes it work like a utility: You only pay for what you need, upgrades are automatic and the enlargement or reduction of the service comprises a simple process.

The NIST [MG11] defines *Cloud Computing* as *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"*.

Several cloud modalities are implemented at present. These are:

- *Private Cloud:* It is used and operated by a single organization.
- *Community Cloud:* Organizations that have shared subjects create this kind of cloud in which the management and support are carried out by the organizations themselves.
- *Public Cloud:* It is provisioned for general public used. The management of the infrastructure is performed by an academic, companies or public organizations.
- *Hybrid Cloud:* Cloud composed by different infrastructures such as public, community or private in order to achieve some specific objectives.
- *Federated Cloud:* Nowadays, this kind of infrastructures have increasing interest. The

federated architectures are a combination of community, public and hybrid clouds in which each of them offer features that the consumer can use in a transparent manner, i.e. the user can use the whole infrastructures as if it was a single entity that internally manages all the processes and infrastructures that constitute it. In the section 2.4.2 are detailed.

The cloud computing features are summarized as follows:

- *On-demand self-service:* A cloud computing user can provision, deploy and release computing resources as needed automatically without human interaction.
- *Broad network access:* Cloud capabilities can be accessed by standard mechanisms or client platforms.
- *Resource pooling:* Different resources can be assigned such as memory, processing capability and network bandwidth among others.These can be geographically distributed or into several cloud machines.
- *Rapid elasticity:* Resources can be elastically provisioned and automatically released. This features facilitates fast scalability of resources.
- *Measured Service:* Resources can be monitored, controlled and reported in order to achieve transparency for the cloud supporter and consumer.

The cloud computing infrastructure offers different kinds of services that are summarized as follows:

- *Software as a Service (SaaS):* It is the capacity provided to the consumer to use the infrastructure with provider's applications.
- *Platform as a Service (PaaS):* It is the capacity provided to the consumer which consists of the applications created by the consumer using languages, libraries and tools supported by the platform owner, can be deployed on the cloud.
- *Infrastructure as a Service (IaaS):* The capacity provided to the consumer consists of provisioning computing resources in order to facilitate the deployment and run its developed software.

1.3 The Fed4FIRE European Project

The *Fed4FIRE* [Fed14a] is an Integrating Project under the European Union's Seventh Framework Programme (FP7) in the topic *Future Internet Research and Experimentation*. The project is performed by a consortium of 29 partner from different countries. These partners are represented in Table 1.1. The project is coordinated by *iMinds*, Belgium. The *Fed4FIRE* project establishes a common federation framework for experimenters. A large number of European facilities are integrated. Such facilities focus on different areas of networking. Some example domains are wireless networking, cloud computing, smart cities

and grid computing among others. Furthermore, one of the main objectives of the *Fed4FIRE* project is the validation of its infrastructure to perform innovative experiments. Then, one of the objectives of the GEO-Cloud experiment is to test and validate the *Fed4FIRE* infrastructure to carry out large scale, industry driven innovative experiments.

Partner	Country	Partner	Country
iMinds	Belgium	*IT Innovation*	United Kingdom
UPMC	France	*Fraunhofer*	Germany
TUB	Germany	*UEDIN*	United Kingdom
INRIA	France	*NICTA*	Australia
ATOS	Spain	*UHT*	Greece
NTUA	Greece	*UNIVBRIS*	United Kingdom
i2CAT	Spain	*DANTE Limited*	United Kingdom
UC	Spain	*NIA*	Korea
UMA	Spain	*UPC*	Spain
UC3M	Spain	*DEIMOS*	Spain
MTA SZTAKI	Hungary	*NUI Galway*	Ireland
ULANC	United Kingdom	*WOOX*	Belgium
UKent	United Kingdom	*BT*	United Kingdom
TELEVES	Spain		

Table 1.1: Partners of the Fed4FIRE European Project.

1.4 The GEO-Cloud Experiment Overview

1.4.1 Experiment Description

The experiment consists of virtualizing a conventional Earth Observation system to offer on demand services with the objective of validating its viability, find the strengths and weaknesses of using cloud computing technology and establish possible solutions for a future implementation in the market [Bec14]. There are three components:

1. *In-orbit mission:* this component generates the raw data. This consists of un-processed images of the Earth captured by a constellation of satellites and downloaded to a distributed network of ground stations.

2. *Data Centre computed in cloud:* the data has to be stored, processed at different levels based on the services offered and distributed to the clients. The data acquired by the in-orbit mission is integrated with other sources to provide higher quality services.

3. *End-users:* users of the provided services with different levels of remote access rights.

1.4.1.1 Experiment Design

The GEO-Cloud experiment requires emulating a complete realistic Earth Observation mission to provide high added value services such as crisis management. To this complex

situation, the system has to response by processing on demand massive and variable amounts of stored and on line transferred data.

GEO-Cloud makes use of the following *Fed4FIRE* facilities: *PlanetLab*,*Virtual Wall* and *BonFIRE*. *PlanetLab* allows us to measure real network characteristics, geographically distributed, to setup our models. *Virtual Wall* allows us to create any desired network topology and emulate the in-orbit mission and the web service to the users. *BonFIRE* provides us a real cloud infrastructure with observability in all the layers to test our cloud based services.

1.4.1.1.1 Implementation of the acquisition of geo-data in Virtual Wall and PlanetLab

The acquisition of geo-data is obtained from the in-orbit mission. The constellation of satellites and the ground stations are emulated in *Virtual Wall* . A network topology is implemented to communicate the different satellites with the ground stations. Every satellite in its orbit and every ground station models are simulated in a different node.

The satellite models simulate the orbits and the pass of the satellites over the ground stations. The ground stations models simulate the coverage of the antennas and the download of the data. When a satellite is inside this radius, the satellite downloads the data to the ground station. Then, the downloaded data in the ground stations is transferred to the *BonFIRE* cloud.

With the *Virtual Wall* network,the *bandwidths, latencies and loss rates* are controlled.Also a realistic network topology to transfer data between different nodes is created.

In order to determine the correct link characteristics for the connections between the ground and the cloud infrastructure, a profiling tool has been developed for measuring appropriate values for the link impairment between these different geographical locations using the *PlanetLab* testbed.

1.4.1.1.2 Implementation of the cloud based services in BonFIRE

To facilitate offering the previous services we propose to implement a multi-layered cloud model in the *BonFIRE* cloud infrastructure to generate on demand geo-information. The multi-layered cloud model is constituted of two layers:

- *Layer 1:* This layer involves the basic satellite imagery services. It acquires the raw data, stores it, has the first level of processing, distributes the processed data and offers the hosting service.
- *Layer 2:* This layer involves the high added value services. It can use historical processed, real time captured and pre-processed data from Layer 1. This layer processes the information for real time generation of geo-information and offers real time access and distribution to the end-users. Typically, the implementation of high added value

EO services involves the ingestion of the raster imagery from the satellites into a spatial database or storage, where it can be refined, simplified, processed or combined with other data sources in vector or raster format. The products, which can be vector or raster data, are distributed or queried using Internet technologies (OGC standards like Web Map Service (WMS)) or through Web services (tiles, caches, etcetera).

Thus, the whole EO system is completely implemented in *Fed4FIRE* (see Figure 3).

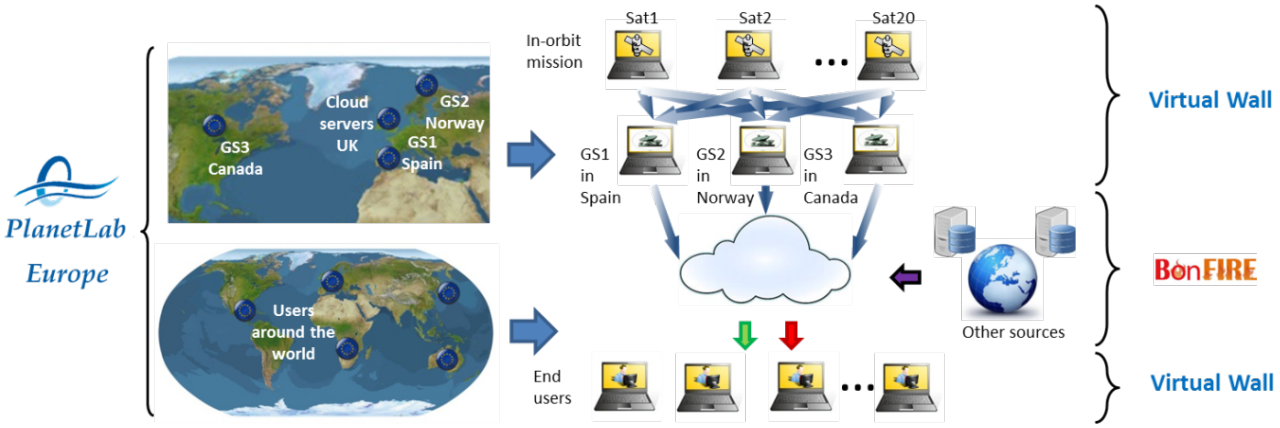


Figure 1.5: GEO-Cloud implementation in Fed4FIRE.

1.4.2 Impact in Fed4FIRE

The experiment will contribute to several objectives of the *Fed4FIRE* project:

- GEO-Cloud will increase trustworthiness of its facilities and support their sustainability.
- GEO-Cloud will test *Fed4FIRE* tools for its use in the industry driven experiments close to market, specifically complex and real time services for Earth Observation industry when critical situations occur.
- GEO-Cloud will validate the tools for monitoring and control of cloud computing and networking in EO services for emergencies and will test the limits of the infrastructure for processing, storing and traffic of massive on-demand data.
- GEO-Cloud will provide feedback to improve the infrastructure during and after the experiment is carried out, sharing our knowledge in traditional infrastructures for EO applications, monitoring, processing and distribution of geospatial data.

1.4.3 Scientific and Technological Impact

GEO-Cloud will contribute to the development of a common framework to provide worldwide services in the Earth Observation field. It will answer if Future Internet technologies can provide viable solutions for the complex EO market and will find the limitations of the current cloud computing technology for its application in Earth Observation.

1.4.4 Socio-Economic Impact

The GEO-Cloud project is used as a framework to offer services from EO users. The benchmark developed in the experiment allows us to establish the frontiers of viable and not viable cloud solutions in EO depending on the type of demand and service offered. This will establish the basis to satisfy the growing demand of added value EO services.

The reduction of the processing, storage, communications and distribution costs of EO services will facilitate the access to the remote sensing technology to common end users, but also to a more general public. GEO-Cloud will contribute in the definitio of the basis to advance in the use of geospatial information of the nine "Societal Benefit Areas" defined in GEO: disasters, health, energy, climate, water, weather, ecosystems, agriculture and biodiversity; by demonstrating whether or not cloud computing offers technologically and economically viable solutions to offer highly demanding services [GW09].

The results obtained in the experiment will be used by *Elecnor Deimos* to offer new services to the general public, current end users and future potential users. The users will be beneficiated since we will define the framework to offer higher quality services.

1.5 Document Structure

This document has been carried out as Master Thesis rules from the *Escuela Superior de Informática* of the *Universidad de Castilla-La Mancha*. It contains the following sections:

Chapter 2: Project Background

In this chapter an overview of the necessary knowledge areas to do this project is made: cloud computing, distributed middleware, processing premises for EO and several testbeds in *Fed4FIRE*.

Chapter 3: Objectives

In this chapter, the main objectives of this project are depicted and explained.

Chapter 4: Method of work

In this chapter the selected methodology is explained and justified. In addition, the used resources such as *Fed4FIRE's* testbeds, hardware and software are described.

Chapter 5: The Geo-Cloud Experiment

In this chapter, the entire GEO-Cloud project is explained. Furthermore, the design and implementation of the satellite constellation, the design and implementation of the satellites and ground stations simulators over *Virtual Wall* , the design and implementation of the cloud architecture for EO and the *PlanetLab* experiment for acquiring network impairments are detailed.

Chapter 6: Evolution and Costs

In this chapter, the project evolution during the development, detailing the phases and

iterations besides the inconveniences and decisions to solve them, is described. In addition,the costs of project and its schedule are shown.

Chapter 7: Results

In this chapter, the results obtained development of the project are shown.

Chapter 8: Conclusions

In this chapter, the main development conclusions of the project are discussed and the reached objectives summarized. Furthermore, some future work and lines of research are suggested.

Chapter 2

Project Background

THIS chapter summarized the background required to accomplish the GEO-Cloud project. Thus, the study of the state of the art will be done by focusing in some aspects of cloud computing platforms, federated testbeds, networking and satellite systems. As a consequence, this section describes the different thematic areas. The conceptual map showed in Figure 2.1 shows the sections and subsections of this chapter. Thereby in the Earth Observation satellites area, the different concepts for creating and modelling a satellite for remote sensing are studied together with its data management and orbital definitions. In high-level languages, some scripting languages are required. In Networking,the different existing network impairments and the tools required to measurement are explained. In Federated Infrastructures, the testbeds that constitute the *Fed4FIRE* insfrastructure are studied. In Graphical User Interfaces, the several frameworks for developing Graphical User Interfaces (GUIs) are described. In Databases, the database management systems are detailed. In distributed systems, the middlewares such as *Hadoop* or *ZeroC Ice* for developing distributed applications are related. And finally, in the Software Design, the requirements for building multiplatform code and other ancillary libraries required by the detailed objectives of GEO-Cloud are explained.

2.1 Earth Observation Satellites

In this section, different areas of aeronautics are involved: Orbital Mechanics defines and calculates the satellite orbits for rounding around the world; Spatial Telescopes contribute with the payload parameters and the data management processes on the fly, stores and sends the science and ancillary data from satellites to the Ground Segment.

2.1.1 Orbital Mechanics

This field [Bra13] studies the motion of the satellite considered as a point with no mass in space affected by *Newton's Laws of Motion* and *Newton's Law of Universal Gravitation*. *The Newton's Law of Universal Gravitation* dictates that *any two bodies in the universe attract each other with a force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between them.*

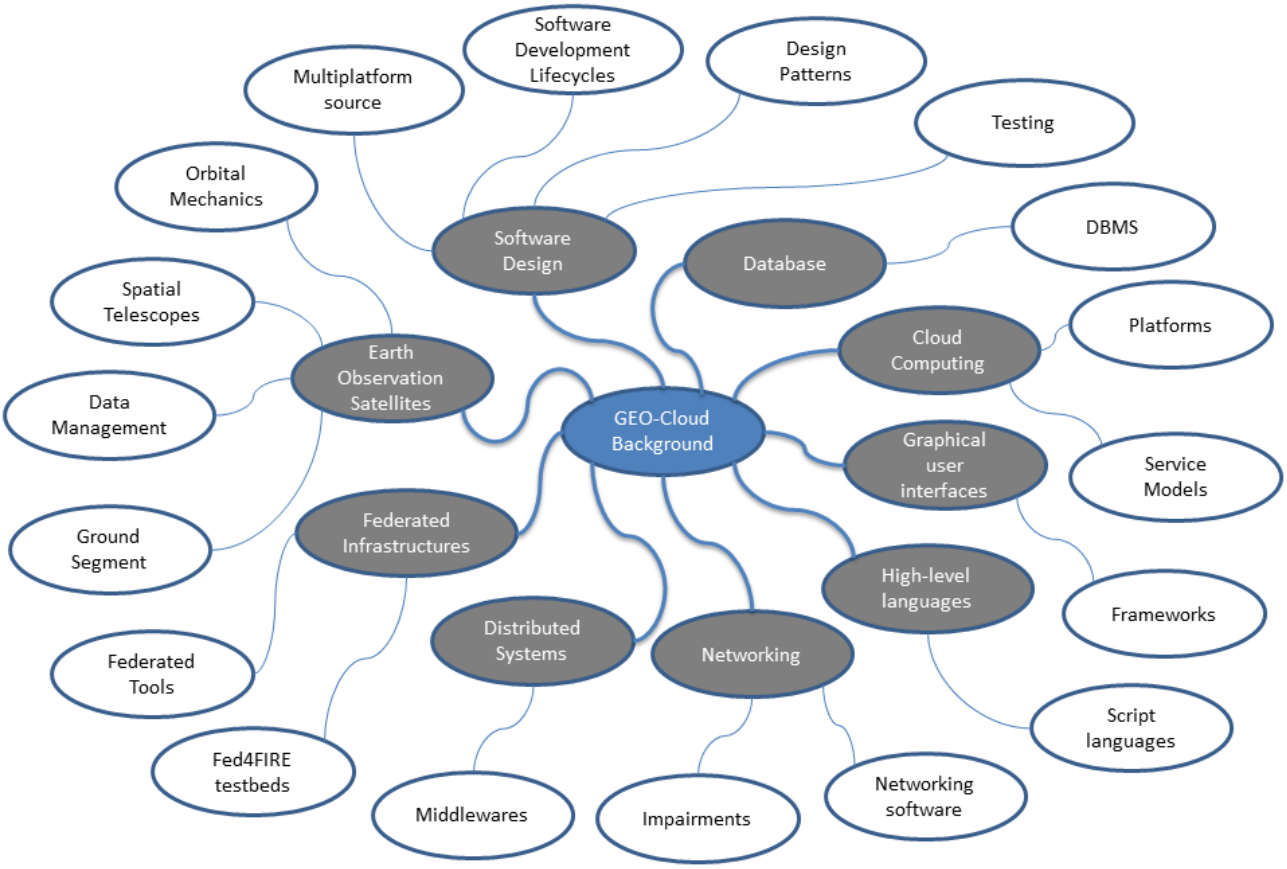


Figure 2.1: Conceptual map of this chapter.

$$F = G * \frac{m_1 * m_2}{r^2} \tag{2.1}$$

where F is the force between the masses, G is the gravitational constant ($6.67\ x\ 10^{-11}\ \frac{Nm^2}{kg^2}$), m_1 and m_2 are the first and second mass in kilograms respectively and r is the distance in meters between the centres of the masses.

Orbital Mechanics focuses on the trajectories of the spacecrafts, manoeuvres for orbital acquisition, orbit maintenance and end of life disposal.

Spacecrafts motion is governed by the *Kepler's Laws of Planetary Motion* [Ste14] which can be derived from *Newton's Laws*. These laws are the following:

1. The orbit of a planet is an ellipse with the Sun at one of the two foci.
2. A line segment joining a planet and the Sun sweeps out equal areas during equal intervals of time.
3. The square of the orbital period of a planet is proportional to the cube of the semi-major axis of its orbit.

2.1.2 Spatial Telescopes

The EO satellites carry on board telescopes pointing to the Earth in order to acquire images of its surface. The information can be obtained in several kinds of spectral bands as visible, near infra-red, thermal, etcetera. This mission uses visible multispectral telescopes distributed in a constellation of satellites to obtain a global map in a daily basis. For selecting the telescopes [Bal13], the swath and the Ground Sample Distance (GSD) have been analysed. The number of satellites is depends on the width of the swath and the telescope resolution is required to achieve quality images for the mission.

Depending on the spectral bands used for imaging, the applications may be different. For example, scientific applications such as soil categorization, vegetation analysis oroceanography use hyperspectral imagers, which offers information about hundreds of bands. Other applications such as traffic monitoring, urban development and surveillance commonly require multispectral imagers with visible bands.

2.1.3 Data Management

Each satellite of the constellation acquires images which have five spectral bands into the visible spectrum. When this spectral data is obtained by telescopes, it is necessary to convert from analogical to digital data. This process is carried out using 12 bits per pixel to codify the intensity signal level with a digital value. Then, the digital data is stored and pre-processed on board.

During the pre-processing activity, some meta-data also known as ancillary data, is added. The ancillary data enriches the acquired sector of image by adding some geolocated information and transmission auxiliary meta-data for communication protocols.

All the pre-processed data is aggregated in the internal storage for downloading when the satellite comes into a visibility zone of a ground station. In the very moment that the satellite enters into the footprint of a ground station, the acquired images are downloaded.

The data download is done by multiplexing the communication channel within two ways. A section of total bandwidth is used for downloading the images that are being acquired in real time and the rest of the bandwidth performs the download of the stored images located in internal storage units.

2.1.4 Ground Segment

The Ground Segment is composed by the antennas, the control centre and the infrastructures for processing and distribution of images namely processing data centre.

The current industries of EO imaging perform the processing, storage and distribution on premises. The steps are the following:

1. First, the antennas receive all the data from the satellites and it is stored.

2. Then, the data centre obtains the images from the ground stations.

3. The data centre starts to process available images.

4. When an image has been processed it is archived and catalogued.

5. Finally, the image can be distributed to end users.

The proposed architecture on-cloud in this project, reduces the delivery time making shorter the cataloguing and archiving times.

In order to implement the archive and catalogue module in cloud, several platforms for sharing geospatial data were studied. These software servers are *GeoServer*[Geo14] and *GeoNetwork* [Com14] among others.

- *GeoServer:* It is a Java-based software server for cataloguing and archiving geospatial images among others functions, that uses OGC standards and allows users to view geospatial images. By default it offers Web Feature Service (WFS) services and Web Processing Service (WPS). In GEO-Cloud, the Archive and Catalogue module has been implemented using *GeoServer* in assembly with its Web Catalogue Service (CSW) extension, because it is flexible, simply, multiplatform, open source and it can receive data from other sources. It can also be integrated with *GeoNetwork*.

- *GeoNetwork:* It is a Java-based and platform independent catalogue application to manage geospatial images. It uses OGC standards such as CSW, WMS and Web Coverage Service (WCS) among other interchanges protocols. It catalogues data from sources but it does not maintains its data. For this reason *GeoServer* was selected.

2.2 High-level languages

For the development of the project, eXtensible Markup Language (XML), JavaScript Object Notation (JSON), *Python* and *Bash* languages has been used. The first, XML, has been used for building the configuration file of the Orchestrator component and to obtain the selected nodes by *JFed* application. JSON was used to create the experiment descriptor for the *BonFIRE* platform. The source of the components of the cloud has been developed using *Python* and the interconnections between modules and others secondary functionalities, in *Bash script*.

2.2.1 XML

XML [Lau98] is a mark-up language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. Most of actual software use it for configuring or for updating its configuration on fly.

2.2.2 JSON

JSON [Org] is a lightweight language for data interchange between applications. The simplicity of JSON results very simple and human-readable way to transmit data objects consisting of attribute-value pairs. Web applications are using this language substituting XML because parsing and generating JSON is more efficient and quick. The *BonFIRE* interface uses it for creating experiment descriptors ought to the resources provided by the platform can be translated into objects and its features.

2.2.3 Scripting languages

The Scripting technics consist of using an interpreted programming language in order to provide advanced mechanisms to specify the functionalities of an application. The most important features of a scripting language are that it is not compiled and it permits effortless development. Furthermore, some interpreted languages are multiplatform because they do not require the compilation of the source for running the application in a target platform.

2.2.3.1 Python

Python [Pyt14] is an object oriented language, although permits imperative and functional programming. It does not need to compile the source because it is interpreted. It permits a flexible development because it is dynamically typed and multiplatform. With this feature, the software can be executed in any platform. The current version is 2.7.4 and it offers several useful and multipourpose libraries such as *Matlib, Pthread, Mysqllib and Pdb*.

2.2.3.2 Bash Script

Bash [Coo14] is a command language interpreter for the GNU operative system. *Bash* is fully compatible with other command language interpreters like *sh* or *ksh*, so the source developed by *Bash* is portable. The Bash scripts contain commands to be executed by the interpreter. It provides a direct line to communicate with the operative system and makes some functionalities ad-hoc. The *Bash* language consists of several sentences that the operative system can to read and translate to play a specific action. All current *Linux* distributions contain a Bash interpreter.

2.2.3.3 Ruby

Ruby [Com] is an object-oriented, cross-platform, general-purpose and dynamic programming language. It was designed and developed by *Yukihiro Matsumoto*. This language has a dynamic type influenced by Groovy, Falcon and other ones. It permits an easy, flexible and agile development.

2.2.3.4 Lua

Lua is a lightweight, cross-platform, prototype-based, object-oriented programming language. It was designed and developed by *Roberto Lerusalimschy*. This language is influenced by C++, Modula and Scheme among others. It provides native data structures as tables, records and associative arrays which this structure perform high throughput [Org14].

2.3 Networking

For simulating an environment as realisticallyl as possible in GEO-Cloud, the impairments of the networks between both Ground Stations and Cloud Platform and between both Cloud Platform and end-users had to be acquired. In addition, features like the bandwidth is obtained in order to stablish the maximum throughput of channel over *Virtual Wall*.

There are several simulators that calculates the value of these network features roughly but the obtained results may be wrong. Therefore, a sub-experiment explained in Section 5.5 for acquiring these values was implemented.

2.3.1 Impairments

The communication through the Internet is not perfect [Tan03] because there are many sources or random events that affect traffic packets that are traversing a different network at any given moment. This is a main issue to take into account when modelling a network. The main impairments of a network are the following:

- *Packet Loss:* this is the disappearance of a packet that was transmitted.
- *Packet Delay:* also known as "Latency" is the amount of time that elapses between the time a packet is transmitted to a physical environment until the packet is received by the target. This delay is composed by three types of delay times: *Propagation delay*, which is the time in which the packet arrives its destination; *Routing/Switching* delay, which is the time in which the routers or switches process the packet; and finally the *Queuing* delays, which are the time the packet is queued in any intermediate hardware of the entire network.
- *Jitter:* jitter is a measure of the variation in the packet delay experienced by a number of packets.
- *Packet Duplication:* may occurs when one packet becomes two or more identical packets.
- *Packet Corruption:* it occurs when the payload of the packet (even a bit) is damaged but the packet continues to flow towards the destination instead of being discarded.

In the GEO-Cloud project, the impairments considered are *Packet Loss* and *Packet Delay*, which permit modelling a simulated and close to reality network.

2.3.2 Networking Software

In the networking subject, there are lots of tools that facilitate the obtantion of the features values for the metrics defined in a network. For measuring the throughput, the bandwidth and the loss-rate the following tools can be used:

- *Microsoft's NTttcp [Mic13a]:* this tool is optimized for Windows environments and it similar to *Iperf*. Testing impariments in networks can be done using this tool.
- *NetCPS: [Aas98]* The most basick network testing can be performed to give a good baseline of network performance. This tool is optimized for Windows enviroments.
- *Iperf [Ipe14]:* the most popular tool to measure maximum Transport Control Protocol (TCP) bandwidth, allowing the tuning of various parameters and User Datagram Protocol (UDP) characteristics. *Iperf* reports bandwidth, delay, jitter and datagram loss. *JPerf* is a Java-extension for providing a graphical user interface.
- *Uperf [Mic13b]:* is a network performance measurement tool that supports the execution of workload profiles. It is more complex and complete than *Iperf* or *NetPerf* allowing the user to model an application using a very high level language and running this over the network. It allows the experimenter to use multiple protocols, by varying message sizes to collect statistics.
- *NetPerf [Com12]:* is a benchmark that can be used to measure the performance of many different types of networking. It provides tests for both unidirectional throughput, and end-to-end latency.

For measuring the delay time the following tools are depicted:

- *Ping:* it is the most used tool for testing the reachability of a host on an Internet Protocol network and for measuring the round-trip-time for packets sent from a source host to a target host. Internet Control Message Protocol (ICMP) protocol is used for it.
- *Traceroute:* is a networking tool for displaying the path and measuring transit delays of packets across a network over the Internet Protocol. This command is available on a number of modern operative systems. By default it works in network layer of the OSI model, sending UDP packets but it can be customized for sending ICMP packets.

2.4 Federated Infrastructures

In this section, some federated infrastructures are described. This infrastructures are used by experimenters for creating new network topologies, distributed applications and new network protocols.

A federated infrastructure is a set of unified and autonomous platforms which are joined in order to provide interoperability and coordinated information sharing among individual components.

The federated architecture pattern was first used by the *US Federal CIO* in 1990s. Then other organizations adopted this paradigm for its infrastructure technologies. Nowadays, this topology is growing up in business where the technological infrastructure has to be distributed, and using critical information around the world.

The benefits of using this kind of network topology are enumerated as follows:

- Independence: the components of the federation has its own rules, protocols and subcomponents. All of them provide an interface for communication between others federation components.
- For the end-users the federated cloud provides an easily way to host apps and the automatically selection of resources from different federated components.

There are several federated infrastructures for experimentation. These are known as "Federated testbeds". The most important ones are the following:

- *GENI*[1]: The *Global Environment for Networking Innovation*, is a distributed virtual laboratory for the future internet sponsored by the *U.S. National Science Foundation* for experimenting. In this platform developments such as protocol design and evaluation, distributed services or content management may be carried out [BCL+14].
- *Open Cirrus:* The *Open Cirrus* testbed provides a federation in cloud computing for experimentation. In order to support these experiments, global services were added by providing distributed common services [ACG+10]. The experiments that can be carried out in this testbed are large-scale clustering experiments, machine learning and scientific computing. *Open Cirrus* is composed by 10 sites in North America, Europe and Asia and the platform has been developed by the *U.S. National Science Foundation*, *the University of Illinois*, *the Karlsruhe Institute of Technology*, *the Infocomm Development Authority of Singapore*, *the Russian Academy of Sciences*, *the Electronics and Telecommunications Research Institute of South Korea*, *the Malaysian Institute of Microelectronic Systems* and *Carnegie Mellon University*. This project is sponsored by Hewlett-Packard, Intel and Yahoo! [2].
- *Fed4FIRE:* The *Fed4FIRE* is an Integrating Project under the European Union's FP7 addressing the work programme topic Future Internet Research and Experimentation. The project is performed by a consortium of 29 partners organisations from 8 countries. The *Fed4FIRE* proposal consists of the creation of heterogeneous and federated platform with different kinds of services and applications such as cloud computing, grid computing, smart cities, wireless networking and large-scale experiments. The federation is composed by the following testbeds:
  - *Virtual Wall:* for creating network topologies.

[1]For more information, see `http://www.geni.net/`
[2]For more information, see `http://opencirrus.org`

- *PlanetLab Europe:* for experimenting with nodes around the world.
- *Norbit:* for experimenting with Wi-Fi resources.
- *w-iLab.t:* this testbed is intended for Wi-Fi and sensor networking experimentation.
- *NETMODE:* it consists of Wi-Fi nodes connected with some processors for networking experimenting.
- *NITOS:* it is a testbed offered by *NITLab* and consists of wireless nodes base on open-source software.
- *Smart Santander:* this is a large scale smart city deployment in Santander city of Spain to experiment Internet of Things.
- *FuSeCo:* this testbed provides resources to experiment with 2G, 3G and 4G technologies.
- *OFELIA:* for testing and validating research aligned with Future Internet Technologies.
- *KOREN:* it provides programmable virtual network resources with necessary bandwith connecting 6 large cities at the speed of 10 Gbps to 20 Gbps.
- *BonFIRE:* it is a multicloud tesbed for experimenting.
- *performLTE:* it is a realistic environment for creating experiments using the Long Term Evolution (LTE) technology.
- *Community-Lab:* it is a distributed infrastructure for researchers to experiment with community networks for creating digital and social environments.
- *UltraAccess:* it provides several Optical network protocols and resources to experiment with Quality of Service (QoS) features, traffic engineering and virtual Local Area Networks (LANs).

In GEO-Cloud project, the testbeds used for implementing are *Virtual Wall*, *PlanetLab* and *BonFIRE*. These facilities are explained in the next sections.

2.4.1 Fed4FIRE Testbeds

In the previous section the *Fed4FIRE's testbeds* were numerated. Next, the facilities used in GEO-Cloud are detailed.

2.4.1.1 Virtual Wall

Virtual Wall is an emulation environment for experimenting with advances networks, distributed software and service evaluation carried out by the *University of Ghent*. It offers the possibility for experimenters to create any type of network topology, e.g. emulating a large multi-hop topology, client-server topologies among others and to check algorithms or protocols for subsequent marketing. Two *Virtual Wall* testbeds are available: *Virtual Wall*

1 contains 200 servers: 100 quad-cores and 100 eight-cores; *Virtual Wall* 2 contains 100 dodeca-cores.

All servers have a management interface and five Gigabit Ethernet connections interconnected all of them by switches. On each of these links, the network impairments can be configured. There are also some virtual machines for customising the resources. Finally, the network impairments are configurable [Wal14].

2.4.1.2 PlanetLab Europe

PlanetLab Europe is part of the *PlanetLab* global system, the world's largest research networking facility, which gives experimenters access to Internet-connected Linux virtual machines. About 1000 servers conform *PlanetLab* platform located in United States, Europe, Asia and elsewhere as Figure 2.2 depicts. This platform can be used by experimenters in order to develop and to check distributed systems, network protocols, peer-to-peer systems, network security and network measurements among others applications [Eur14].

Through *Fed4FIRE*, the experimenters can reserve and deploy some PlanetLab Europe resources and to experiment with them.

Figure 2.2: Geographical distribution of PlanetLab Europe.

2.4.1.3 BonFIRE

BonFIRE [HAHB+12] is a multi-cloud testbed based on an Infrastructure as a Service (IAAS) delivery model with guidelines, policies and best practices for experimenting. Currently, *BonFIRE* is composed by 7 geographically distributed testbeds, which offer heterogeneous cloud services, compute resources and storage resources [Bon14]. These testbeds are *EPCC*, *INRIA*,*Wellness*, *HLRS*, *iMinds* and *PSNC*. The Figure 2.3 shows these testbeds

and its interconnections. The compute resources that *BonFIRE* offers are summarized in Table 2.1.

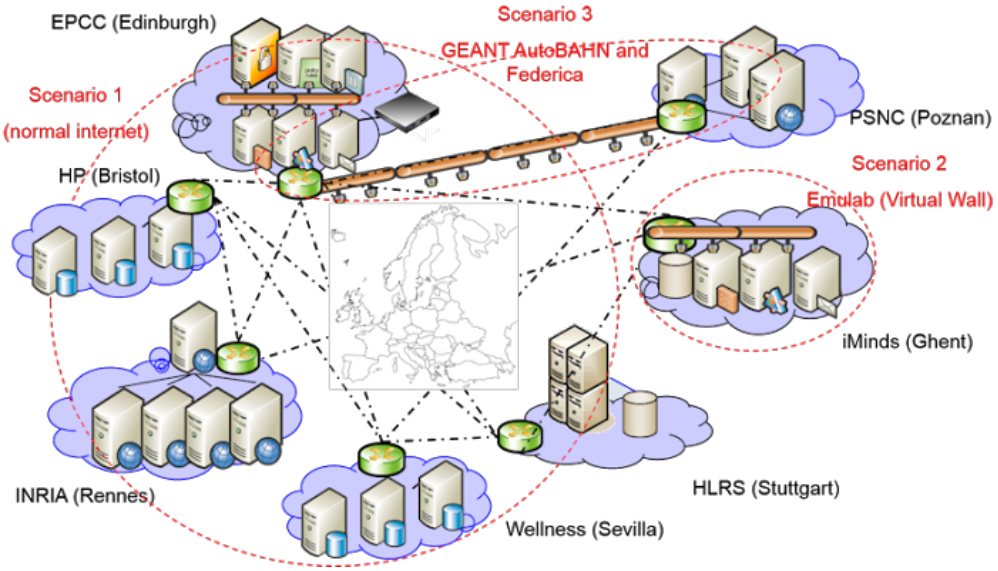


Figure 2.3: BonFIRE testbeds.

The setup of the compute resources can be done by using contextualization variables in order to provide important information for software applications in the virtual machines. This testbed also offers elasticity resources, that are dynamically created, updated and destroyed according to the execution environment.

Name	CPU cores	Memory	Features
Lite	0.5	256 MB	-
Small	1	1 GB	-
Medium	2	2 GB	-
Large	2	4 GB	-
Large+	2	4 GB	Higher CPU clock speed
Large-en	4	4 GB	-
Xlarge	4	8 GB	-
Xlarge+	4	8 GB	Higher CPU clock speed
Custom	User defined	User defined	VCPU must be an integer

Table 2.1: Instance types of BonFIRE.

2.4.2 Federated Tools

In the *Fed4FIRE* project, some tools for deploying, controlling and monitoring the experiments have been developed. Some of these tools are used only by developers to deploy, to provision, to make reservations and to discover resources [Fed14b]. These tools are *Flack, Omni and SFI*, which have a common interface named *SFA*.

The experimenters utilize the tools for controlling the experiments. These tools are described as follows:

- *NEPI:* the Network Experimentation Programming Interface, is a life-cycle management tool for network experiments. It is developed in Python and it provides a high-level interface to describe experiments, to provision resources, to control experiments and to collect all the results of the experiment. To highlight that this implementation provides an important way to manage and to make a workflow for an experiment [INR14].
- *Object Monitoring Framework v6 (*OMF6*):* is a generic framework that facilitates the definition and orchestration of the experiments. It can be used for controling the experiment through a distributed infrastructure based on Extensible Messaging and Presence Protocol (XMPP) and an Aggregate Manager that manages all the information flows. It can be integrated with OML for provisioning, control and collection of all the information about the experiment.
- *Object Monitoring Language (*OML*):* this tool provides an Application Programming Interface (API) to collect the status of any sensor or device. It implements a reporting protocol composed by a client and a collection server. On the client side, any application can be implemented using the OML API for collecting any information about the status of the devices concerning the experiment.

2.5 Graphical User Interfaces

A GUI is a software that facilitates the human-machine interaction using images and graphic items. Normally, the interactions are produced by user actions directly and these actions corresponds to events. The GUI events are usually produced by the mouse, touchpad, keyboard or nowadays, a touchscreen. When an event is located, a specific action is performed changing the interface itself, or creating, updating or deleting data or to accomplish a specific proceeding.

2.5.1 Frameworks

There are lots of graphical user interface engines or frameworks available for Python [Ath14]. Because the project is developed in Python, the user interface was also developed in Python. In addition, the searched engines are multiplatform. Finally, considering these criteria, some frameworks and libraries such as *PyGUI, PyGtk, PyGame, PyQt, PyKDE and WxPython*

were studied:

- *PyGUI:* it is an API for building graphical user interfaces easily. It can be used in any platform or operative system. Furthermore, it use PyOpenGL libraries and it is distributed under General Public License (GPL) v3.
- *PyGtk:* It is a library set of Python with which the programmer can develop programs with a graphical user interface easily. It is multiplatform and it is distributed under Lesser General Public License (LGPL) licence with few restrictions.
- *PyGame:* is a set of Python modules facilitates the creation of games and multimedia software such as graphical user interfaces. *Pygame* is fully portable and runs on nearly every platform and operative system. This engine evolves funcionality of Simple DirectMedia Layer (SDL) library, OpenGL and provides Blender integration. It is distributed under GPL v3.
- *PyQt:* is a Python binding for Qt development. Qt is a cross-platform and GUI framework for developing graphical user interfaces. Qt is distributed under GPL v3 and LGPL license. It is developed by Nokia and it permits the development of software for mobile, embedded platforms, desktop platforms and any operative system.
- *WxPython:* is a GUI toolkit for Python that allows us to create robust, highly functional graphical user interfaces easily and simply. It is implemented in Python and it is distributed under GPL v3 license.

2.6 Database

Since computer manage the users information, humans had tried to sort these data and collect it from machines effectively. For this purpose, the databases were created. The databases are conceptual models to orderly store the information. Another step forward was the creation of the Database Management System (DBMS). The DBMS is a software to manage the information contained into database to concentrate, to arrange and to provide lots of mechanisms to manage the information contained into database repository [Gro10].

2.6.1 DBMS

In order to simulate the constellation of satellites for GEO-Cloud project, some free DBMS were studied. The most interesting are the following:

- **MySQL:** open source relational database management system developed by *MySQL AB*. It is the most used database system in web applications. *MySQL* provides triggers, cursors, sub-selects, stored procedures, an embedded database library and efficiently works with distributed systems. This DBMS is multiplatform and it works in any operative system [3].

[3] For more information, see `http://www.mysql.com/`

- **MonetDB:** open source column-oriented database management system developed by *Centrum Wiskunde & Informatica* in Netherlands. It offers high performance on complex queries in large databases combining tables with lots of columns and multi-million rows. This DBMS is performed in data mining and geographic information systems [4].

- **PostgreSQL:** *PostgreSQL* is an object-oriented relational distributed database management system developed by *PostgreSQL Global Development Team*. It is distributed under Berkeley Software Distribution (BSD) license. This database uses a client-server model based on multi-processing for granting system stability [5].

MySQL was selected in the project development because there are quite documentation and the syntax is similar to Structured Query Language (SQL).

2.7 Distributed Systems

Nowadays, most of applications need to obtain information from other sources or to communicate with another hardware either other computers or devices. As a result, some programming paradigms like client-server, remote procedure call and remote object invocation were born [TvS08].

- **Client-Server paradigm**
  This architecture is composed by two parts: the first one is the server component and the second one is the client. The server is listening request from clients and when an application comes the server decodes, processes it and sends back the reply to the client.

  Remote Procedure Call
  Remote Procedure Call uses the same principle as Client-Server but in a more complex way. Basically, in remote procedure call paradigm, a client calls a function as if it were a local function on the client machine, but this called function is located in another machine which performs the function and returns the results.

  Remote Object Invocation
  This paradigm was born when the object-oriented programming paradigm was developed. There are distributed objects and these objects are not necessary fixed in a host. When a distributed-object is created, it is bound to a set of machines as slave, so when a client application recovers that object using a distributed-object registry or another one like that, the provided operations by the distributed object can be remotely performed as if it was a operation call in applicant machine.

[4] For more information, see `https://www.monetdb.org/Home`
[5] For more information, see `http://www.postgresql.org.es/`

2.7.1 Middleware

For client-server paradigm there are not any middleware for developing applications. Normally these applications are developed using the standard oriented socket libraries. For Remote Procedure Call (RPC) development, there are several implementations such as Simple Object Access Protocol (SOAP), Common Object Request Broker Architecture (CORBA), UNIX RPC and Pyro.

For Remote Object Invocation paradigm, there are several novel middlewares. The most used are Hadoop and ZeroC Internet Communications Engine (ICE) and its features are as follows:

2.7.1.1 Hadoop

Hadoop [Fou08] is a framework for developing reliable and scalable distributed applications. It is based on the architecture used by *Google* named "MapReduce" and other components like *Google File System*. Some of its components are the following:

- *Hadoop Common:* the common utilities for other Hadoop modules.
- *Hadoop Distributed File System:* distributed file system that provides high-throughput access.
- *Hadoop YARN:* a framework for job scheduling and resource management.
- *Hadoop MapReduce:* a YARN-based system for parallel processing of large data sets.

It is designed for scaling up thousands of machines. It is maintained and distributed by *Apache Software Foundation* under GPL v.3 license.

2.7.1.2 ZeroC ICE

ZeroC ICE, the internet communications engine, is a modern distributed computing platform. It supports languages such as C++,Python, Java and Ruby among others. It allows developers to create new powerful, efficient and simple applications with minimal effort [Zer13]. The main features of this middleware are the following:

- It is a general-purpose distributed computing platform.
- It uses a modern and flexible specification language.
- Dynamic invocation and dispatch are supported.
- Request Forwarding is provided.
- Asynchronous and synchronous invocation and dispatch.
- One-way, datagram and batched invocation are also supported.
- Multi-threading software developments are fully thread-safe.

- Several transport protocols can be used as TCP,Secure Sockets Layer (SSL),UDP and also Internet Protocol (IP)v4 and IPv6 are supported.

ICE provides the following services:

- *IceGrid:* it is a service for large-scale grid computing applications.
- *IceStorm:* it is a sophisticated event distribution service.
- *Freeze:* it provides object persistence and the possibility to migrate these objects to a database.
- *Glacier2:* it is a firewall service.
- *IcePatch2:* it is a software and files distribution service.

2.8 Software Design

In this section, the software engineering knowledge and the software portability required by this project are studied. These areas are described in the following subsections.

2.8.1 Multiplatform source

One of the main objectives of GEO-Cloud consists of the software portability. As this project is an experiment, it has to be checked in any platform with any operative system, so the developed source was written in Python. Among the huge variety of Python interpreters, the used version in this project source was Python 2.7.

GNU/Linux was the platform in which the development of all project files were carried out. Besides the libraries distributed within Python, *Paramiko* Python library [Tea13] was used. This library implements Secure SHell (SSH) protocol operations for remotely managing and controlling machines. This library is distributed under GPL v3. http://www.lag.net/paramiko/

2.8.2 Software Development Life Cycles

Historically, the software was developed without any methodology, so there were lots of bugs in the code. As a result, programs did not work correctly or they were inefficient or inclusive, they stopped unexpectedly. Then the saviour programmer tried to solve the trouble wasting lot of time. Thus, the software development was very expensive [GGV+13]. At present, the software development is guided by life cycles. In this cycles, stages such as planning, designing, implementation, testing, documenting, deployment and maintenance are basics for software development. Depending on how these stages are done, there are several life cycles. Among others, the most important and more used are the following:

- *Waterfall model:* strict model in which the developers have to follow the following stages in order: analisys, design, implementation, testing, deployment and maintenance.

- *Spiral model:* this life cycle combines the waterfall model and rapid prototyping.
- *Iterative and incremental cycle:* it is based on the development of small but ever-larger portions of a software. During software development, several iterations may be in progress at the same time. Using this cycle, the end-users can check the no-ended product for updating requirements or for giving feedback to end-user.
- *Agile development:* this model is very used nowadays. Its principles are the iterative development and the incorporation of continuous feedback and staging iterations.

2.8.3 Design Patterns

The software programming patterns that were studied in the development of GEO-Cloud experiment are the following:

- Singleton: only a single instance of a class can exist.
- Composite: a tree structure of simple and composite objects.
- Proxy: an object that represents another one.

2.8.4 Testing

While the GEO-Cloud software was being developed, the testing process was done. The testing was done as follows: hand testing and test-case oriented testing. In test-case oriented testing, sundry testing frameworks were studied. The selected framework for testing in Python was *UnitTest* and its extension *Nose*. It is included in Python standard library, it is easy to use and it has lots of plugins [Fou14a].

The developed test-cases were black-box essentially. Some white box test-cases were developed but the code coverage was not thorough.

Chapter 3

Objectives

THE main objective of the GEO-Cloud project is the modelling and implementation of a close to real world Earth Observation System in *Fed4FIRE* cloud.

3.1 Specific Objectives

The main aim of this project is defined regarding a series of functional objectives as follows:

- **To implement a data centre in the multicloud BonFIRE testbed**: This infrastructure for EO provides the ingestion of raw data, on demand processing the raw data to obtain images, and later storing and cataloguing transparently, dynamically and automatically. For this purpose this components are needed:
  - **Cloud Orchestrator**: To manage the processing of images obtained from satellites and to control all the stages (ingestion of images, processing, storing and cataloguing).
  - **Archive and Catalogue Subsystem**: To archive, to catalogue and to provide to end-users, the results obtained from the processing stage.
  - **Processors Module**: To process the raw data sent from the Orchestrator component obtaining processed images from the Earth. The product processors are provided and owned by *Elecnor Deimos* company, so their implementation are treated as a black box.
- **To implement a Space Simulator in Virtual Wall testbed:** This software simulates: the satellite constellation getting images and downloading them into the ground stations; and ground stations receiving the raw data sent by the satellites and sending it to cloud infrastructure. In addition, the connections between the satellite constellation and ground stations and between the ground stations and cloud infrastructure are also simulated.
- **To obtain the network impairments values using PlanetLab testbed**: In order to implement a closest to reality experiment, the real values of loss-rate, bandwidth and Round Trip Time (RTT) of both the satellites-ground stations and ground stations-cloud infrastructure networks, are obtained. Then, the impairments of the networks

implemented using the *Virtual Wall* testbed, are updated.

- **To create the experiment**: By combining and integrating the developed in the three testbeds mentioned before.
- **To create a Graphical User Interface**: This GUI provides the deploy,execution and stop, easily and automatically.
- **To validate the experiment**: Simulating of a scenario to validate the entire system obtaining, downloading, processing, storing and cataloguing EO images. In addition, to evaluate if future internet cloud computing and networks provide viable solutions for conventional EO Systems to establish the basis for the implementation of EO infrastructures in cloud.

Chapter 4

Method of work

In this chapter, the development software methodology applied for carrying out this project is described. Furthermore, the tools such as hardware and software used are enumerated.

4.1 Development Methodology

The selected methodology is based in the *Iterative and Incremental* model. This model provides numerous advantages such as early adaptability, continuously testing and has an implementation implementation close to the final product. A scheme of the model is shown in figure 4.1.

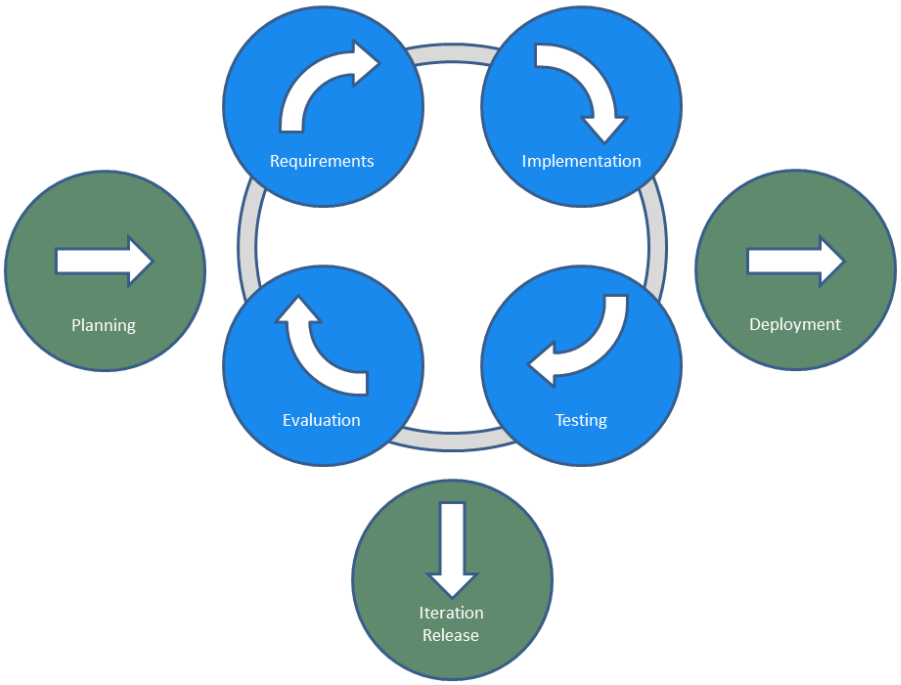


Figure 4.1: Iterative-Incremental model scheme.

This methodology enforces a strategy based in small iterations. In each iteration, the analisys of the requirements, implementation, testing and evaluation of the product is done. This permits to generate a prototype, iteratively to approach the final version validating the

initial requirements. Thus, if the advisor requires any changes, they can be easily done.

As the project has several parts, it is divided in modules and each module, in iterations. These iterations are explained in Section 6.1.

4.2 Tools used in the project

In this section the tools and resources used to develop the project are described.

4.2.1 Programming Languages

Several programming languages were used for the Geo-Cloud implementation. These languages were described in Section 2.2. They can be summarized as follows:

- **Python:** it is the main language used in the project. It is a multiplatform, multipurpose object oriented language, although it permits imperative and functional programming. It does not require to compile the source because it is interpreted.
- **XML:** it is a mark-up language that defines a set of rules for encoding documents in a format. It is used in some configuration files thath both the *Orchestrator* and the User Interface tool use.
- **JSON:** it is a lightweight language to interchange data between applications. The *BonFIRE* experiment descriptor is written in this language, indicating the virtual machines instances, physical machines and the network resources instantiated.
- **Bash:** it consists of several sentences that the operative system can read and translate to play a specific action. All current *Linux* distributions contain a *Bash* interpreter.

4.2.2 Hardware

The development of the GEO-Cloud project was carried out in a PC provided by *Elecnor Deimos* with the following features:

- *Intel Core i5 3450 3.1 GHz*
- *8 GB RAM*

For the Version Control, Bitbucket [1] server and the *Git* version control system were used.

The used resources provided by the *Fed4FIRE* testbeds are the following:

- **PlanetLab**: It currently provides 1204 nodes at 593 sites around the world. The selected nodes for carrying out this project are shown in Table B.1 and Table B.2.
- **Virtual Wall**: It consists of 100 nodes (dual processor o dual core servers) interconnected via a non-blocking 1.5 Tb/s Ethernet switch. 29 nodes have been used to simulate the satellite constellation and ground stations.

[1] Official Webpage: http:www.bitbucket.org

- **BonFIRE**: Nodes from diferent testbeds were selected. The features the several types of nodes are summarized in Table 2.1.
  - From INRIA: 2 nodes Medium.
  - From EPCC: 2 node Xlarge for Orchestrator and for Processing Chain.
  - From IBBT: 1 Shared Storage.

4.2.3 Software

The software used in the development of this project can be grouped in: Operative Systems, where the software was developed and tested; Software development tools, with which the source was performed; graphics and documentation software, in which the documentation was carried out; and software libraries, which provide specific functionalities.

- **Operative Systems**
  - *Ubuntu*: UNIX based operative system in which the development was done. It is based in Debian and distributed under GPL v3.
  - *Debian*: UNIX based operative system distributed under GPL v3. It provides more than 37500 precompiled software packets which can be easily installed. The *BonFIRE* , *Virtual Wall* and *PlanetLab* virtual machines have this operative system.
- **Software Development Tools**
  - *Emacs*: versatile and powerful text editor developed by Richard Stallman. This tool is used with *Pymacs* together. Version used 23.2.
  - *JFed*: tool developed by *iMinds*[2] (Ghent) for deploying nodes in *Fed4FIRE* testbeds.
  - *BonFIRE Web Interface*: *BonFIRE* tool that provides the experiment deployment. It can be done manually or automatically using a experiment descriptor.
  - *Ipython*: command shell for interactive computing in Python. It allows us to check some code statements before adding them in a software.
  - *GeoServer*: is an open source software server licensed under the GNU General Public License, written in *Java* that allows us to share and edit geospatial data. Designed for interoperability, it publishes data from any major spatial data source using open standards.
  - *Apache Tomcat*: is an open source software implementation of the *Java Servlet* and *JavaServer Pages* technologies from *Sun Microsystems*, and provides an Hypertext Transfer Protocol (HTTP) web server environment for *Java* code to run in, being one of the most popular *Java* application servers [Fou14b].

[2]Official Webpage: http:www.jfed.iminds.be

- **Graphics and documentation**
  - *LaTeX*:is a document mark-up language widely used for the communication and publication of scientific documents [OPS14]. It is multiplatform and is distributed under GPL v3.
  - *GIMP*: is the *GNU Image Manipulation Program*. It provides photo retouching, image composition and image authoring among others. It is multiplatform and is distributed under GPL v3.
  - *Dia*: multiplatform software for drawing many types of schemes. Some design schemes of this document were performed on it.
  - *Libre Office:* free open source office suite, developed by *The Document Foundation*. This suite provides such software as *Draw* and *Writer* among other programs.
- **Software Libraries**
  - *Python-mysql*: *Python* library that provides a high-level interface *MySQL* programming.
  - *Python-mathplotlib*: *Python* library that provides some tools and functions for plotting and representing data.
  - *Python-xml*:*Python* library used to manage XML files.
  - *Paramiko*: *Python* library that provides high-level interfaces for connecting through SSH to other network host.
  - *PyQt:* Python binding for the *Qt* cross-platform framework. *PyQt* is available in two versions, *PyQt4* and *PyQt5*. *PyQt5* was used for developing the GUI of the project.
  - *Phonon:* multimedia API provided by Qt for handling multimedia streams. It was used in the development of the GUI for experimentation.

Chapter 5

The Geo-Cloud Experiment

THE GEO-Cloud experiment consists of the emulation of a realistic and complete Earth Observation System that provides services using cloud technology. For that purpose a constellation of satellites, ground stations, a cloud architecture, use cases and end users' models were designed.

The EO system was computed in *Fed4FIRE*: A constellation of satellites record images of the Earth in a daily basis. The images are transferred to a network of ground stations that ingest the data into a cloud computing infrastructure. The data is processed and distributed to end users.

The GEO-Cloud experiment uses in three testbeds: *Virtual Wall*, *BonFIRE* and *PlanetLab*. GEO-Cloud is divided into two sub-experiments:

- One experiment in a system integrated in *Virtual Wall* and *BonFIRE* .
- One experiment in *PlanetLab* .

The experiment in *Virtual Wall* and *BonFIRE* emulates the whole EO system. The system is constituted by:

- A *Space System Simulator* including models of satellite constellation and a ground stations network.
- A cloud computing system that implements a novel architecture that integrates an *Orchestrator*, a *Processing Chain* component based on *EaaS* with image processors to transform the raw data acquired by the satellites in orthorectified images, and finally an *Archive and Catalogue* module for storing, cataloging and allocating images.

The experiment in *PlanetLab* consists of the emulation of the networks that constitute the links between the ground stations to the cloud and from the cloud to the end users. There, the network features such as bandwidth, loss-rate and latency are monitored and measured. Those parameters once measured are used to update the network models implemented in *Virtual Wall* .

In the next sections, the detailed design and implementation in testbeds of the GEO-Cloud experiment is fully described.

5.1 Satellite System Design

5.1.1 Design of the flight and ground segments

In this subsection, the main characteristics of the system are presented. Objectives, constraints, previous estimations and possible modifications and their effects in the system are exposed.

In a wide vision, it is an EO system consisting of a constellation of satellites equally spaced in a Low Earth Orbit (LEO) orbit with the aim of achieving daily coverage of the entire Earth surface. These conditions imply a very sophisticated handle of a huge quantity of data.

The following requirements have been fulfilled to design the system:

- Swath: 160 km (based on state of the art cameras).
- Resolution: 6.7 m (based on state of the art cameras).
- Low Earth Orbits.
- Sun Synchronous orbits.
- Download data rate: 160 $Mbps$ (based on Deimos-2 satellite characteristics).
- Optical Bands: 5 Multispectral (based on state of the art cameras).

5.1.1.0.1 Global Daily Coverage

The objective of this system is the acquisition of images of the total Earth surface in a daily basis. Global coverage is considered to include the land surface that is shown in Figure 5.1.

Figure 5.1: Land surface to be acquired in a daily basis.

5.1.1.0.2 Flight Segment

The satellites are selected according to the current state of the art. However, some enhancements can be assumed as a way to adjust the analysis to the short term future.

5.1.1.0.2.1 Satellite performances

As a first iteration for the system, small platforms of less of $500\ kg$ are supposed according to the size and payload bay characteristics of representative platforms in this category. This study is based on the bus platforms currently offered by Surrey Satellite Technology Ltd, Satrec Initiative and Sierra Nevada Corporation.

In order to reduce the amount of satellites orbiting the Earth, a design with two payloads was assumed. Thus, the swath (the width of the field of view of each satellite in the surface of the Earth) can be duplicated without decreasing the resolution.

The satellites shall be pointing to nadir, acquiring images without maneuvering because of the acquisition plan for covering the Earth daily. Simultaneously, when a satellite gets into the field of view of a Ground Station, the download of the acquired images starts and at the same time the satellite continues recording the surface of the Earth.

The main specifications of the satellites in the constellation are shown in Table 5.1. The values of each parameter were selected according to the state of the art and the desired quality of the images in the mission.

Specification	Value
GSD	$6.7\ m$
Swath	$160\ km$
Number of bands	5
Digitalization	$12\ bits$
Download Data Rate	$160\ Mbps$
Compression Rate	$2:1$

Table 5.1: Main performance parameters of the satellites.

5.1.1.0.2.2 Orbit definition

The use of *sun-synchronous* orbits in EO satellite missions is common. These orbits guarantee that the lighting conditions of the imaged places are the same during the mission, which is a very desirable characteristic.

Local Time Ascending Node (LTAN) is also a desired condition of the orbit very related to the lighting and weather conditions of those places that the satellite overflies (LTAN is selected according to the desired local time of the overflown places and the cloud formation during the day). The use of *LTAN 10:30h* is commmon for Earth Observation because at that time the best conditions of cloudiness, reflectivity and luminosity are given.

Other of the main parameters of an orbit is the altitude. Altitude has effects in the resolution and the swath of the satellites, which has impact in the number of satellites required to achieve the coverage objective. According to the value of those parameters in the payloads

included in the satellites, the reference altitude for the system was found to be $646\ km$. Sun Synchronous Orbit (SSO) condition implies a relation between the altitude and the inclination of the orbit of $97.97\ ^{o}$ in this case.

5.1.1.0.2.3 Number of satellites in the constellation

The number of satellites is calculated by using the altitude ($646\ km$), the inclination ($97.97\ ^{o}$) and the swath ($160\ km$). As a result, 17 satellites are required to carry out this mission. In Figure 5.2 the whole constellation is shown.

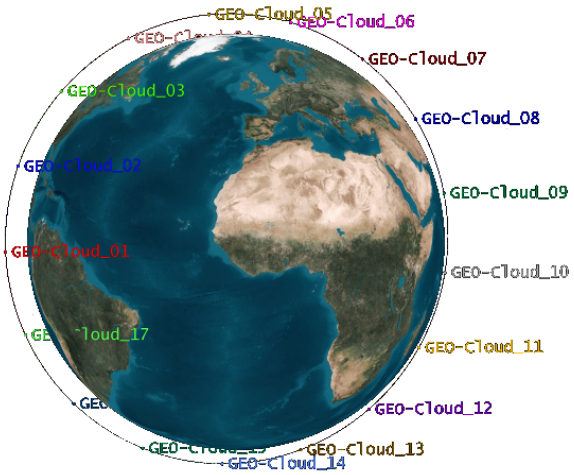


Figure 5.2: Constellation of 17 satellites in a SSO at $646\ km$.

5.1.1.0.3 Ground Stations design

When the satellite acquires the data, it has to be downloaded to the Ground Stations. Due to the huge quantity of data and the limitations to download data rate sets, several stations distributed over the surface of the Earth are required. They will allow the satellites to communicate with them and to download the images. Figure 5.3 shows how the Ground Stations and their footprints (area in which the satellites can communicate with the Ground Station) are distributed.

5.1.2 Generated Data Volume

Specifically, $135,698,500\ km^2$ of ground surface are daily acquired. With the following expression can be estimated the data volume generated:

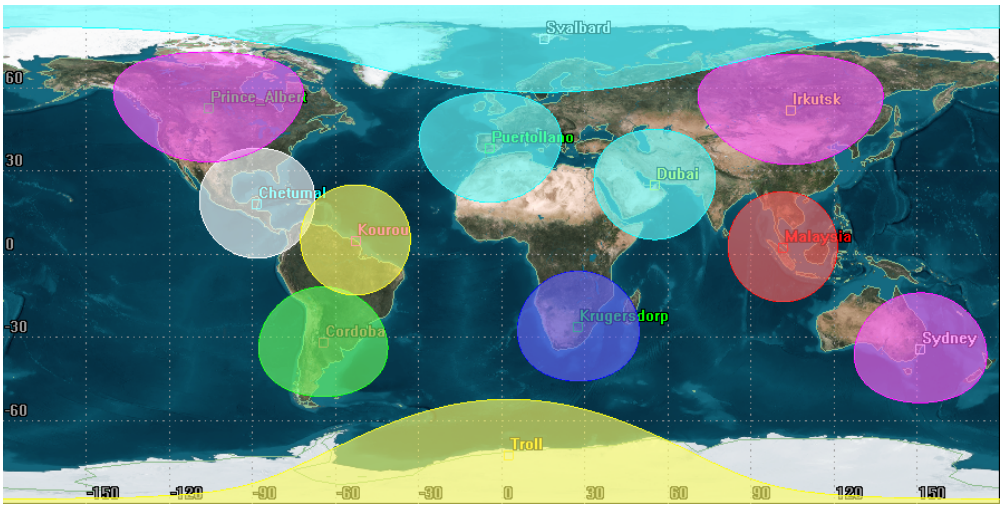


Figure 5.3: Footprints of the selected Ground Stations.

$$Acquired\ Data\ Volume = \frac{Acquired\ Surface}{GSD^2} * N^{o}bands * Digitalization \quad (5.1)$$

Before downloading the images they are compressed. The ancillary data is included in the process (auxiliary information useful for the geolocation of the images, satellite status and communication protocol among others). In this case, the ancillary data is estimated to be 12% of the acquired data (based on Deimos 2 satellite measurements), which is added and then compressed. With the values depicted in Table 5.1, the ancillary and the daily ground surface adquired, the data on ground can be estimated. As a result, including ancillary data before compression, $11.55\ TBytes$ shall be daily downloaded.

5.2 Satellite System Development

This section presents the *Space System Simulator* that emulates the real behaviour of a satellite constellation of 17 satellites that download images to a network of 12 ground stations connected with the *BonFIRE* cloud. The simulator is implemented in *Virtual Wall* .

The *Space System Simulator* is constituted of three components:

- *Satellite System Simulator* : it simulates the dynamics and communications of the constellation of 17 Earth observation satellites.
- *Ground Station System Simulator* : it simulates the dynamics and communications of the network of 12 ground stations distributed around the World.
- *Distributed Database:* it contains all the required information and parameters to initialize the simulators and make them run in every specific simulator. While the *Satellite System Simulator* and the *Ground Station System Simulator* are implemented in

Virtual Wall , the distributed database is computed in the *BonFIRE* cloud.

To adapt the *Space System Simulator* to the Fed4FIRE testbeds, some located scenarios were designed in order to reduce the amount of data to process, store and distribute during the simulations. Thus the simulation is shortened to a specific time required to acquire and download certain areas of interest *(AOI)*.

The detailed design of the *Space System Simulator* and its implementation in *Virtual Wall* are next presented.

5.2.1 Image Acquisition

The first step is to implement the acquisition of images by the satellites of the six pre-defined scenarios (for an extended description on the scenarios see Appendix D) with the satellite constellation:

1. Emergencies – Lorca Earthquake (Spain)
2. Infrastructure monitoring. Affection in railway infrastructures by sand movement in desert areas (Spain)
3. Land Management – South West of England
4. Precision Agriculture – Argentina
5. Basemaps – Worldwide

In each scenario, an Area of Interest *(AOI)* is defined to be acquired during the simulation. The in orbit satellites are nadir pointing in order to acquire images of the sub satellite point over the Earth surface as shown in Figure 5.4.

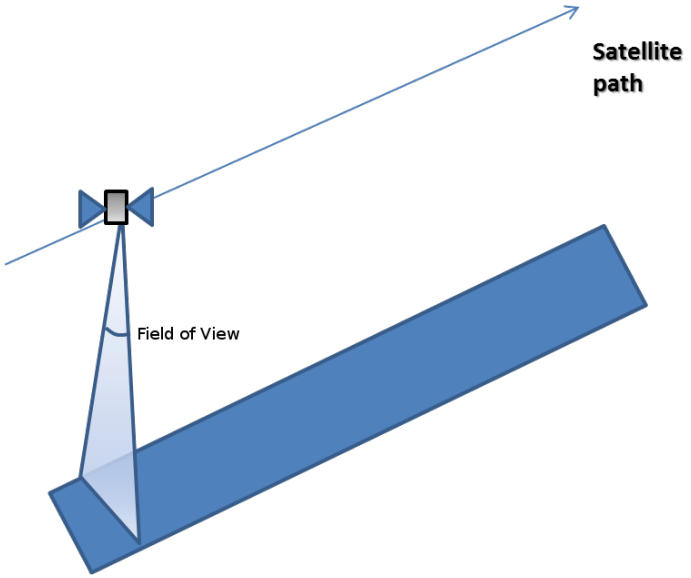


Figure 5.4: Example of strip imaging.

In the next subsection the assumptions to simulate the system as realistically as possible are described for the scenarios. Note that the *AOI* of the scenario 5 is the whole land mass on Earth. This involves that because of the huge amount of data that has to be recorded, specific assumptions to the scenario 5 was made according to the Fed4FIRE testbeds' limitations.

5.2.1.1 Assumptions in satellite image acquisition

The *AOI* in each scenario can be acquired by one or more satellites depending on the size of the *AOI* relatively to the scene size (note that the GEO-Cloud satellites have a swath of 160 km, thus we divide the acquisition into scenes of 160 km x 160 km. In each scenario we call *main satellites* to the satellites with the task of acquiring the *AOI* (note that all satellites are *main satellites* in the model for the scenario 5).

Along the duration of the scenario other satellites acquire images of the areas of the Earth surface they are passing over. Those images are not in the area of interest.

During the experiments in *BonFIRE* , they will be processed but not stored into the system, since the focus of the mission in every scenario has to be in the defined area of interest. Those non *AOI* images allow us to emulate a real system, since we take into account all the possible inputs to the system (note that in the scenarios 5 and 6 all the images created are *AOI*).

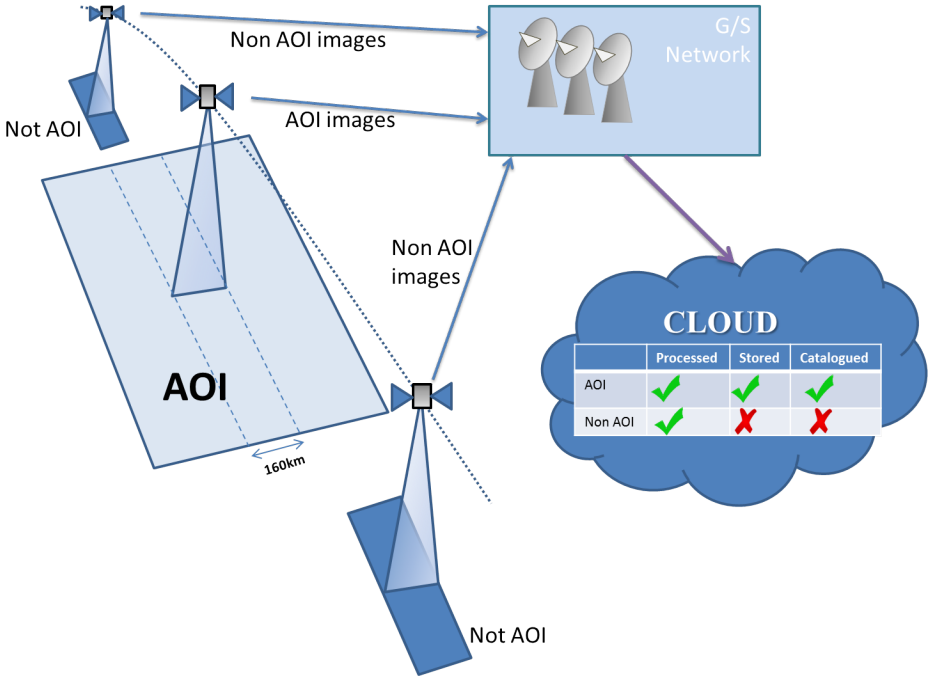
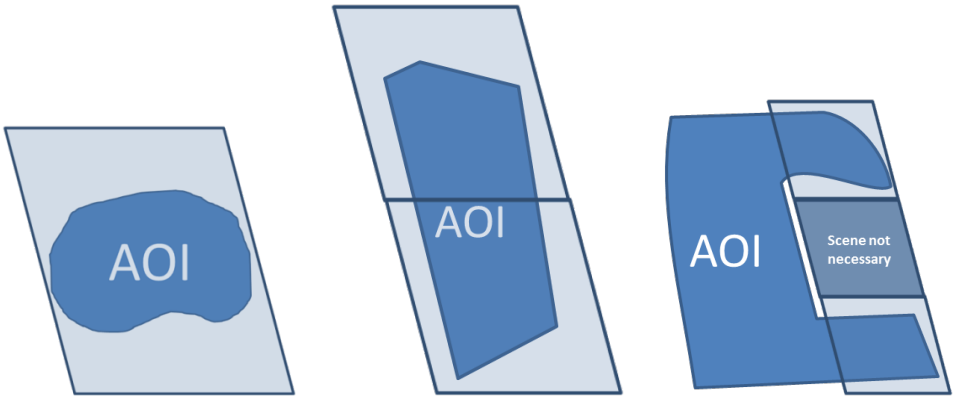


Figure 5.5: Diagram of images adquisition.

Thus, those non *AOI* images, during the simulation will be acquired, downloaded to the ground stations, transferred to the cloud and processed, but not stored, neither catalogued. In addition, it has to be taken into account that the main satellites can also acquire some images out of the *AOI* during the duration of each scenario. Those acquired images are also



(a) Simple acquisition. (b) Multiple consecutive acquisitions. (c) Multiple non-consecutive acquisitions.

Figure 5.6: Different types of the AOI acquisitions.

considered non *AOI* images (see Figure 5.5).

In every scenario the time is set to 0 when the simulation starts in the Fed4FIRE environment.

These assumptions are summarized as follows:

1. Only the scenes acquired by a satellite that include the *AOI* are considered to carry out the complete simulation.
2. Images taken by the satellites that are out of the *AOI* are processed but not stored into the system to adjust the experiment execution to the resources provided by *BonFIRE* .

5.2.1.2 Types of acquisition of the AOI by a single satellite

Depending on the relative sizes of both the *AOI* and the scenes, three different situations can occur when a single satellite is acquiring images (see Figure 5.6):

- Simple acquisition: the *AOI* (at least the part to be imaged by the satellite) fits into just one scene.
- Multiple consecutive acquisitions: the *AOI* to be acquired by a satellite fits in a strip with several scenes.
- Multiple non-consecutive acquisitions: the *AOI* to be acquired by a satellite fits in a strip but there are some scenes between the acquisitions that have to be acquired in order to complete the general mission (world map daily) but it is not necessary for the scenario.

5.2.1.3 Parameters required for the simulation of the scenarios

For each scenario, the following parameters and assumptions are considered:

1. *S: Scenario number.* It indicates the scenario that is being simulated (from 1 to 6).
2. *Main satellites:* The main satellites are those that acquire images of the AOI in each scenario. Notice that all the satellites are numbered from 1 to 17.
3. *ADR:* Acquisition Data Rate. Rate at which the images are acquired by the satellite.

$$ADR = 1395\ Mbps \tag{5.2}$$

4. *CR:* Compression Rate. Rate at which the images are compressed in the satellite for their download.

$$CR = 14.1 \tag{5.3}$$

5. $Bandwidth_{sat}$*:* Download rate between the satellites and the ground stations.

$$Bandwidth_{sat} = 160\ Mbps \tag{5.4}$$

The previous bandwidth is used as follows:

- The satellite is downloading images in memory and at the same time is acquiring images and downloading them:

$$Bandwidth_{sat} = Bandwidth_{(sim\_acq)} + Bandwidth_{mem} \tag{5.5}$$

where $Bandwidth_{(sim\_acq)}$ is the bandwidth used for downloading images that are being acquiring at the same time and $Bandwidth_{mem}$ is the bandwidth used to download images in the satellite memory. $Bandwidth_{(sim\_acq)}$ can be obtained as follows:

$$Bandwidth_{(sim\_acq)} = \frac{ADR}{CR} = 98.9\ Mbps \tag{5.6}$$

where *ADR* is the Acquisition Data Rate (Mbps) and *CR* the data compression rate. And $Bandwidth_{mem}$ can be obtained as follows:

$$Bandwidth_{mem} = Bandwidth_{sat} - Bandwidth_{(sim\_acq)} = 61.1\ Mbps \tag{5.7}$$

6. T_0*:* Start time. It is the start time of the scenario. The scenario starts when the first main satellite begins the acquisition of the first portion of *AOI*.
7. T_f*:* End time. It is the time when the last main satellite finishes the downloading of the *AOI* taken images.
8. $[T_{AOI}]_{0i}$*:* Time a main satellite starts acquiring the *AOI*. i stands for the number of the main satellite.
9. $[T_{AOI}]_{fi}$*:* This is the time a main satellite finishes acquiring a piece of *AOI*.

10. $[\Delta T_{AOI}]_i$*:* This is the difference $[T_{AOI}]_{fi} - [T_{AOI}]_{0i}$. Note: Some satellites (all of them in the model for scenario 5) can require more than one orbit to acquire the corresponding *AOI*, and then these satellites will have a different $[\Delta T_{AOI}]_i$ in each acquisition of *AOI*. $[\Delta T_{AOI}]_i$ is a multiple of $T_{scene} = 23.4\ s$, which is the needed time to acquire one scene of $160\ km$x $160\ km$, considering that satellite velocity on ground is $v_{Gsat} = 6.84\ km/s$:

$$T_{scene} = L_{scene}/v_{Gsat} \tag{5.8}$$

$[\Delta T_{AOI}]_i$ can be calculated as follows:

$$[\Delta T_{AOI}]_i = nL_{scene}/v_{Gsat} = nT_{scene} = [T_{AOI}]_{fi} - [T_{AOI}]_{0i} \tag{5.9}$$

where n is the number of scenes, L_{scene} is the length of the scene (in this case $160\ km$), and v_{Gsat} the velocity of the satellite on ground (in this case $6.84\ km/s$).

11. $[T_{GS}]_{0ij}$*:* Time a main satellite i enters into the visibility cone of a *Ground Station* j.
12. $[T_{GS}]_{fij}$*:* Time a main satellite i leaves the visibility cone of a Ground Station j.
13. $[\Delta T_{GS}]_{ij}$*:* This is the difference $[T_{GS}]_{fij} - [T_{GS}]_{0ij}$.
14. T_{start}*:* This is the start time of the simulation. We chose to be 5 seconds in order to synchronize all the satellites.

As an example, the Table 5.2 shows the data was obtained from "Scenario 2: Infrastructure monitoring affection in railway infrastructures by sand movement in desert areas":

Scenario	**Start Time** T_0	**End Time** T_f	**Main Satellite**	**Start Acquisition** $[T_{AOI}]_{0i}$	**End Acquisition** $[T_{AOI}]_{fi}$	**Number of *AOI* scenes**
2	54060	54753.4	4	54060	54083.4	1
			3	54390	54413.4	
			2	54730	54753.4	

Table 5.2: Example of data of image acquisition for Scenario 2.

5.2.2 Image Downloading

The acquired images by the satellites shall be downloaded to the emulated ground stations through the antennas network designed (see Section 5.1). All the scenarios finish at T_f, when the last main satellite completely downloads the last acquired portion of the *AOI*. During the simulation, the rest of the satellites not overflying the *AOI* would be imaging other places; these images will also be downloaded and processed in parallel to the *AOI* images to simulate a realistic case, but as previously explained those images will not be stored nor catalogued.

Because of the limitations in the testbeds (limited storage and compute resources in *BonFIRE* cloud and multiplexed channel over virtual networks in *Virtual Wall*) it is necessary to scale the data involved in these simulations from the original design of the GEO-Cloud experiment. It has been decided to change the designed compression rate from lossless compression $2 : 1$ to a rate compression $14 : 1$. With this change, the data volume has been reduced and the size of every $160km$ x $160km$ scene after compression is now $288\ MB$ instead of the almost $2\ GB$ size of the original images.

In order to download the data, the satellites have to be inside the visibility cone of a ground station. During these accesses, the satellites download images at a rate of $160\ Mbps$. Part of these images are in memory before entering the visibility cone and others are being acquired simultaneously during the downloading task (see section 5.2.1.3 for a wider explanation of the download parameters and Figure 5.7).

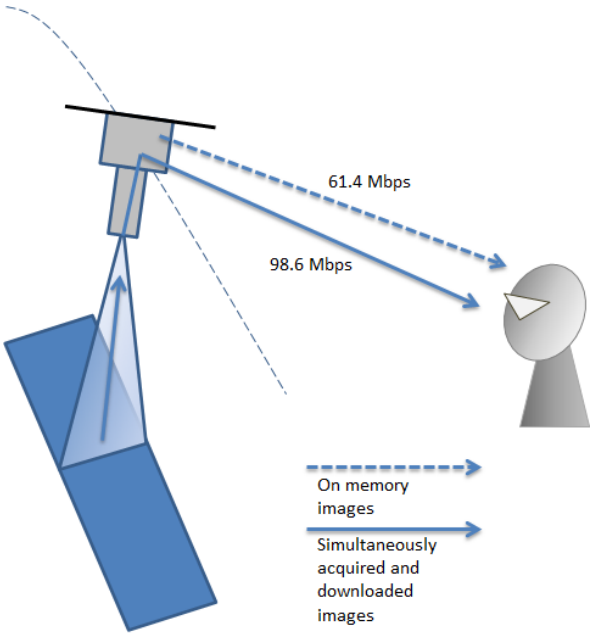


Figure 5.7: Multiplexed images downloading.

It is possible that the duration of the accesses and the time to download an image are not multiples of $T_{scene} = 23.4\ s$. In this case, an image would be partially downloaded at the end of the access but it would be supposed that this image is completely downloaded in the following access in other ground station.

For the memory management, a *LIFO* (last in, first out) procedure is applied. This implies that the latest acquired images will be downloaded as soon as possible. This imposes that when a satellite is inside the visibility cone of a ground station both the images latest acquired and the images that are being acquired of the *AOI* will be simultaneously downloaded by multiplexing them. Note that because of differences between the rate of acquisition and downloading, and also because of the compression process, the downloading of images that

are being simultaneously acquired do not occupy all the bandwidth of $160\ Mbps$; the portion not occupied is used to download on board storage also with a *LIFO* process (See section 5.2.1.3 numbers 3 to 5).

As an example, the data of the accesses for the Scenario 2 are included in Table 5.3.

Sat. Ident. Number	Ground Station	Access Start Time (s)	Access Stop Time (s)	Duration of the passes (s)
1	Krugersdorp	54184.2	54516	331.8
2	Dubai	54184.2	54516	331.8
2	Krugersdorp	54549.4	54753.4	204
3	Dubai	54060	54753.4	204
4	Dubai	54060	54337.5	277.5
4	Svalvard	54660	54753.4	93.4
5	Svalvard	54316.3	54753.4	437.1
6	Prince Albert	54708.1	54753.4	45.3
6	Svalvard	54060	54639.4	579.4
7	Prince Albert	54362	54753.4	391.4
7	Svalvard	54060	54296.7	236.7
8	Prince Albert	54060	54577.9	517.9
9	Prince Albert	54060	54246.8	186.8
15	Troll	54419	54670.3	251.3
16	Troll	54085.6	54303.4	217.8
17	Krugersdorp	54495.4	54753.4	258

Table 5.3: Example of data of accesses for Scenario 2.

Next, a summary of the assumptions made for the image downloading from the satellites to the ground stations are described:

1. Time to process the images on board before downloading is considered to be instantaneous. Then, the simultaneous acquisition and download of the *AOI* inside the visibility cone is done without latency.

2. The ground stations network was designed to guarantee the download of a complete world map in a daily basis. With this assumption into account, the management of the memory on board of each satellite does not require an additional design. This involves that we do not know in which ground station all the areas acquired out of a visibility cone are downloaded.

3. Because of the small gap between the satellites, which are in the same orbit, more than one antenna in each ground station is required to be available. This provides the possibility of having simultaneous contacts with all the satellites inside the visibility cone.

5.2.3 Getting the satellite data

The steps to obtain and simulate the realistic behaviour of the satellites are next described.

5.2.3.1 Extraction of the real behaviour of the satellite constellation in the scenarios

The first point is to obtain the realistic behaviour of the constellation of satellites designed in section 5.1. For that purpose, the constellation of satellites and the ground stations were implemented in the $SystemsToolKit^{®}$ (STK) software.

The next steps were followed to obtain the values of the parameters required for the simulations of the defined scenarios:

- Commonly for all the scenarios:
  1. Implementation of the satellite constellation.
  2. Implementation of the ground station network.
  3. Calculation of the access time for each satellite communicating with each ground station during 24 hours.
- For each different scenario:
  4. Identification of the main satellites.
  5. Extraction of the times each satellite is acquiring the AOI: $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$.
  6. Calculation of the number of scenes acquired: n.
  7. Extraction of the access duration of each satellite with the ground stations it communicates within the scenario duration: $[T_{GS}]_{0ij}$ and $[T_{GS}]_{fij}$.
  8. Start time of the scenario: T_0.
  9. End time of the scenario: T_f.

5.2.3.2 Exportation of data to the simulator

Once obtained the previous data, it is exported to different Comma Separated Values (CSV) format files:

- The *Scenario_<NUM>_<SCENE>.csv* files: it contains the information of the $[T_{GS}]_{0ij}$ and $[T_{GS}]_{fij}$ for each scenario. *<NUM>* indicates the number of the scenario and *<SCENE>* the name of the scenario. In the GEO-Cloud experiment we simulate 6 scenarios. The previous parameters are defined in the following table:

  For example, the format of the resulting *Scenario_1_ Emergencies_Lorca_Earthquake.csv* file with the $[T_{GS}]_{0ij}$ and $[T_{GS}]_{fij}$ information for the Lorca scenario, looks like the code extract shown in Listing 5.1.

```
"GEO-Cloud_005-To-Troll - Access","Start Time (EpSec)","Start Time (
    UTCG)","Stop Time (EpSec)","Stop Time (UTCG)","Duration (sec)"
1,63638.000,11 Aug 2014 10:10:38.000,63661.400,11 Aug 2014
    10:11:01.400,23.400

"GEO-Cloud_006-To-Troll - Access","Start Time (EpSec)","Start Time (
    UTCG)","Stop Time (EpSec)","Stop Time (UTCG)","Duration (sec)"
7,63638.000,11 Aug 2014 10:10:38.000,63661.400,11 Aug 2014 10:11:01
```

Listing 5.1: Extract of the *Scenario_1_Emergencies_Lorca_Earthquake.csv* of the Lorca scenario

In the code, a heading is separated into columns. It is divided as the table 5.4 shows.

- The *All_Scenarios.csv* file: it contains the information of the $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$ for all the scenarios. A piece of the *All_Scenarios.csv* file that contains the $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$ of all the scenarios is shown in Listing 5.2. It also contains T_0 ,T_f and the number n of *AOI* obtained images.

```
Scenario,Start Sce,End Sce,Sat,Start sat,End Sat,Images
1,63638,63661.4,11,63638,63661.4,1
,,,,,,
2,54060,54753.4,4,54060,54083.4,1
,,,3,54390,54413.4,1
,,,2,54730,54753.4,1
,,,,,,
3,62480,63865.4,15,62480,62503.4,1
,,,14,62821,62844.4,1
,,,13,63161,63184.4,1
,,,12,63501,63524.4,1
,,,11,63842,63865.4,1
```

Listing 5.2: Extract of the *All_Scenarios.csv* code of the Lorca scenario.

This file was manually computed. The required fields to describe the behaviour of the satellites acquiring the *AOI* are the depicted in Table 5.5.

5.2.3.3 Processing of the Scenario_<NUM>_<SCENE>.csv and All_Scenarios.csv

A script is designed and developed to get and merge the data from the *Scenario_<NUM>_<SCENE>.csv* and the *All_Scenarios.csv* files, and to store the information in the database: *setDatabase.py*. It was developed in Python 2.7 but it is also compatible with all Python versions.

This has to be done because as previously explained *Scenario_<NUM>_<SCENE>.csv* contains the $[T_{GS}]_{0ij}$ and $[T_{GS}]_{fij}$ parameters of each single scenario and *All_Scenarios.csv* the $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$ of all the scenarios. Thus the script matches the $[T_{GS}]_{0ij}$ and $[T_{GS}]_{fij}$ of one scenario with the $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$ of such a scenario.

To execute *setDatabase.py* the arguments depicted in Table 5.6 are required.

Column Title	Function
GEO-Cloud_sat-To-station	Represents the access of a satellite to the specific ground station. sat represents the satellite number and station the ground station name.
Start Time (EpSec)	Means $[T_{GS}]_{0ij}$.
Start Time (UTCG)	Means $[T_{GS}]_{0ij}$ in UTCG format
Stop Time (EpSec)	Means $[T_{GS}]_{fij}$
Stop Time (UTCG)	Means $[T_{GS}]_{fij}$ in UTCG format.
Duration (sec)	Means $[\Delta T_{GS}]_{ij}$.

Table 5.4: Columns headings *Scenario_<NUM>_<SCENE>.csv* files.

An example of the execution of the programme is the following:

> python setDatabase.py 192.168.0.2 All_scenarios_file.csv Scenario_1_Example1.csv Scenario_2_Example2.csv Scenario_3_Example3.csv

5.2.4 Space System Simulator

The *Space System Simulator* is constituted by the following modules:

- The Database
- The *Satellite System Simulator*
- The *Ground Station System Simulator*

To simulate every scenario, on the one hand, the *Satellite System Simulator* is executed. It is constituted by 17 *Satellite Simulators* of individual satellites; each one reproduces the characteristic behaviour of every single satellite in the constellation. On the other hand the *Ground Station System Simulator* is also executed. As in the case of the *Satellite System Simulator* , the *Ground Station System Simulator* is constituted of 12 *Ground Station Simulators* of individual ground stations reproducing the behaviour of every single ground station of the designed network.

The diagram in Figure 5.8 depicts a scheme representing the *Space System Simulator*.

The Space System Simulator sequentially follows the next steps during the execution:

- First, *setDatabase.py* is executed. The script fills the database fields with the information of every scenario.
- Second, the *Ground Station System Simulator* is executed. The *Ground Station System Simulator* can execute all the *Ground Station Simulators* or some of them individually

Column Title	Function
Scenario	Scenario number. Indicates the scenario executed
Start Sce	Indicates the start of the scenario. It is T_0
End Scen	Indicates when the scenario finishes. It is T_f
Sat	Indicates the number of the *main satellite*.
Star Sat	The time when the satellite starts acquiring the *AOI*. It is $[T_{AOI}]_{0i}$.
End Sat	The time when the satellite finishes acquiring the *AOI*. It is $[T_{AOI}]_{fi}$
Images	This is the number of *AOI* images. It is n.

Table 5.5: Columns headings of the *All_Scenarios.csv* file.

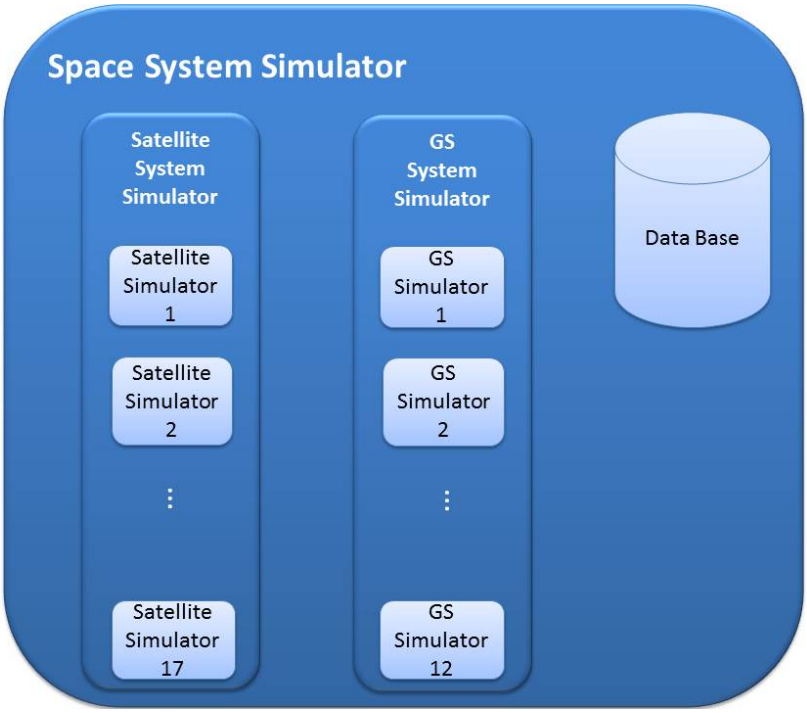


Figure 5.8: Space System Simulator's Architecture.

Argument Position	Meaning
1	IP address of the host which contains the MySQL database
2	The relative path or absolute path of the file that contains the information of all scenarios (see Listing 5.2)
3...	The *Scenario_-<NUM>_-<SCENE>.csv* files that contain the events occurred in a particular scenario. At least one file must be introduced, otherwise an execution error is produced.

Table 5.6: Arguments of *setDatabase.py*.

to simulate faults in the ground stations as it can occur in reality. Thus, when a satellite needs to download data into a ground station that is offline, it will receive an exception so the downloading of images does not start.

- Third, the *Satellite System Simulator* is executed. As in the previous case, the *Satellite System Simulator* can execute all the *Satellites Simulators* or some of them individually for the same reason.
- Finally, when the scenario finishes, the simulation is stopped.

The *Database*, the *Satellite System Simulator* and the *Ground Station System Simulator* are thoroughly described in the next subsections.

5.2.4.1 Database

A database containing all the required data for the simulations was implemented. The selected database management system *(DBMS)* is *MySQL*. It was selected because it is friendly usable, works in multiple platforms, provides transactional and nontransactional storage engines and the server is a separate program for use in a *client/server* networked environment.

The design of the database architecture is shown in Figure 5.9.

The database is constituted by three tables: *Scenarios table*, *Satellites table* and *Ground-*

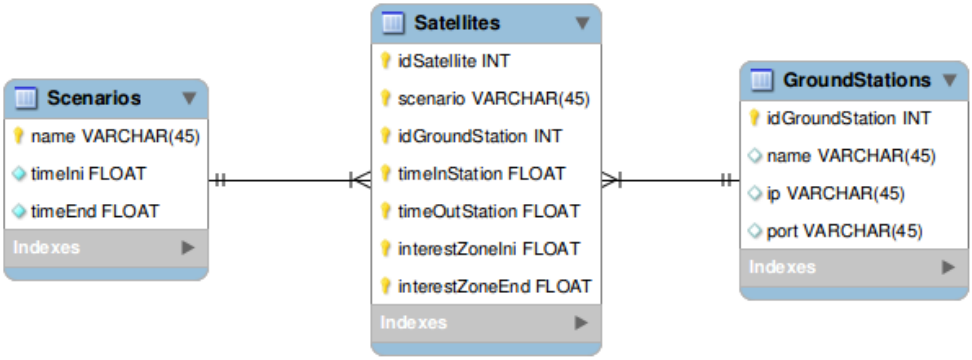


Figure 5.9: Database architecture.

Stations table. The functionalities of the tables are described in Table 5.7.

Table	Function	Columns	Relationship
Scenarios table	It contains the name of the scenario, the start time T_0 and the end time T_f. Its primary key[1] (represented with a key symbol in Figure 5.9) is the name column.	name timeIni timeEnd	None
GroundStations table	It contains the ID of the Ground Station and its name. Its *primary key* is the idGroundStation.	idSatellite scenario idGroundStation timeInStation timeOutStation interestZoneIni interestZoneEnd	None

[1]The *primary key* identifies an object; for example a scenario, a ground station or a satellite

Table	Function	Columns	Relationship
Satellites table	It contains the ID of the satellites *idSatellite*, the scenario and the ground station id, *idGroundStation*, in which the events occur, the time in which the satellite enters into the visibility cone $[T_{GS}]_{0ij}$, *timeInStation*, the time when it leaves the visibility cone $[T_{GS}]_{fij}$, *timeOutStation*, the time when the satellite enters into the *AIO* $[T_{AOI}]_{0i}$, *interestZoneIni*, and the time when the satellite leaves the *AOI* $[T_{AOI}]_{fi}$,*interest ZoneEnd*. All of them are primary keys because a satellite can download images of several *AOIs* in a single ground station.	idGroundStation name ip port	This table relates the *Scenarios* table and the *GroundStations* table. This is because the *scenario* and *idGroundStation* columns are foreign keys[2]. The characteristics of these columns are *On delete cascade* and *On update cascade*. This means that when an object contained in a table (*GroundStation* table or *Scenarios* table) is deleted, it is also automatically deleted the object that refers to it, and if it is updated the object is also automatically updated.

Table 5.7: Funcionalities of the database tables for the simulator

5.2.4.2 Database Filling

- **Filling during initialization**

The database is filled during the initialization process by executing *setDatabase.py* as

[2]The *foreign key* binds two tables, linked across a field. For example in this case, the column from the *Satellites* table which is named *Scenario* refers to an object that exists in the *Scenarios* table.

previously described in section 5.2.3.3. This execution makes the following sequential actions:

1. All the data that is in the database is cleaned to avoid inconsistencies.
2. The *GroundStation* table is filled with the ground stations information (only name and id).
3. The Scenario table is completely filled with the data of each scenario.
4. Finally, the Satellites table is completely filled. This process joins the information contained in the *Scenario_<NUM>_<SCENE>.csv* with file *All_Scenarios.csv*. This means that for each Scenario the $[T_{GS}]_{0ij}$, $[T_{GS}]_{fij}$, $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$ representative of the simulated scenario are taken. That information is inserted into the database as follows:
   - For each scenario the information regarding the accesses of the satellites to the ground stations are selected, $[T_{GS}]_{0ij}$ and $[T_{GS}]_{fij}$.
   - Those accesses are merged with the *All_scenarios.csv* file in order to obtain the accesses of the satellites in which there are *AOI* images.
   - Then if the satellite acquired an *AOI* in the current scenario, the information of the *AOI* is also included in the database.
   - Otherwise the fields with the information regarding the *AOI* are filled with -1 value.

- **Filling during the execution of the Space System Simulator**

  During the execution of the *Ground Station System Simulator* the ip and port columns in the *GroundStations* table are filled. This is done when a *Ground Station Simulator* starts its execution in a node. The *IP address* and *port* of that node are obtained and included in the database. Once each ground station has an *IP address* and a *port* associated, the *Satellite Simulators* can read both of them and communicate with *Ground Stations Simulators* servers.

5.2.5 Satellite System Simulator

The *Satellite System Simulator* reproduces the behaviour of the 17 satellites constellation by executing 17 individual *Satellite Simulators* characterized by the specific behaviour of each satellite.

The *Satellite System Simulator* is a manager that executes 17 *Satellite Simulators*. The *Satellite System Simulator* requires the scenario number and the *IP adress* of the database as an input and executes the *Satellite Simulators* by providing them the specific identity of the satellite *(idSatellite)*, number of the scenario *(S)* and IP address of the distributed database *(ipDatabase)*.

Next, the architecture of the *Satellite Simulator* for each individual satellite is described.

5.2.5.1 Satellite Simulator

The *Satellite Simulator* is constituted by the following components:

- *Initialization Module:* This module obtains the following parameters and initializes the Satellite Simulator:
  - *S:* the number of the scenario to simulate.
  - *idSatellite:* identification number of the satellite (1 to 17).
  - *ipDatabase: IP address* of the database.

  Once obtained the previous parameters, the Initialization Module connects with the database and fetches the $[T_{GS}]_{0ij}$, $[T_{GS}]_{fij}$, $[T_{AOI}]_{0i}$, $[T_{AOI}]_{fi}$, the *IP addresses* of the *Ground Station Simulators* servers "ipGroundStations" and the ports of the *Ground Station Simulators* servers "portGroundStations" for the simulated satellite in the specific scenario. Those times$[T_{GS}]_{0ij}$, $[T_{GS}]_{fij}$, $[T_{AOI}]_{0i}$, $[T_{AOI}]_{fi}$ are provided to the *Satellite Dynamics Module*.

- *Satellite Dynamics Module:* This module represents the dynamics and associated parameters of the satellite. It requires $[T_{GS}]_{0ij}$, $[T_{GS}]_{fij}$, $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$ of every scenario as inputs, and it provides the acquired images by the satellite in function of the time acquisition (those images are downloaded to the ground stations, simulated by the *Ground Station System Simulator*). Figure 5.10 shows the relation between the previous modules.

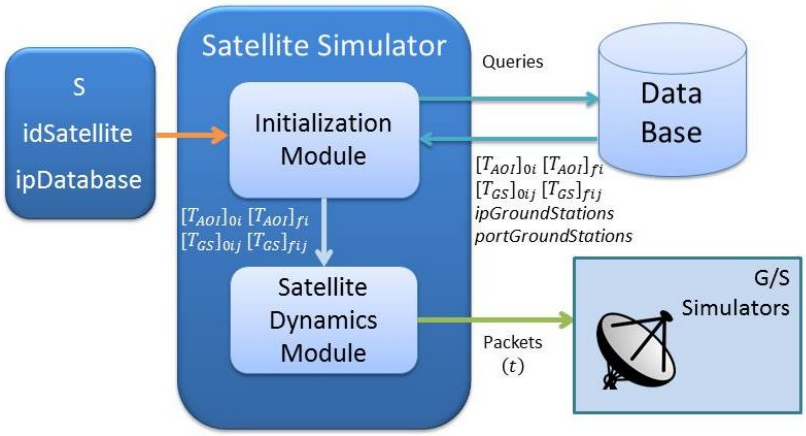


Figure 5.10: Satellite Simulator Architecture.

5.2.5.1.1 Satellite Simulator Workflow

The *Satellite Simulator* starts the execution following these steps:

1. The *Initialization Module* obtains *S*, *idSatellite* and *ipDatabase*.

2. Then, the *Initialization Module* makes queries on the database to obtain all the required satellite data for current scenario: $[T_{GS}]_{0ij}$, $[T_{GS}]_{fij}$, $[T_{AOI}]_{0i}$ and $[T_{AOI}]_{fi}$. The queries made to the database contain the particle "order by timeInZone" column. This means that the database returns the $[T_{GS}]_{0ij}$ times in ascending order. Other query gets the *ipGroundStations* and *portGroundStations* of every *Ground Station Simulator*.

3. Schedule of data download: the *Satellite Simulator* schedules the data download from the *Satellite Simulator* to the *Ground Station Simulators* (see Section 5.2.5.1.1.1).

4. When all the communications between the *Satellite Simulator* and the *Ground Station Simulators* have been scheduled in time, the *Satellite Simulator* starts *Satellite Dynamics Module*.

5. When the last scheduled task has been executed, the *Satellite Simulator* finishes the execution.

Figure 5.11 shows the process explained above. The diagram also shows the interactions with the other subsystems (see also Figure 5.10). Figure 5.11 also shows the inputs for the *Satellite Simulator* and the outputs after its execution. These outputs are the packets in time sent to the ground stations and a log file that contains the information about the execution. In orange the inputs to the *Satellite Simulator* are represented, in green the transitions in the workflow and in blue the outputs.

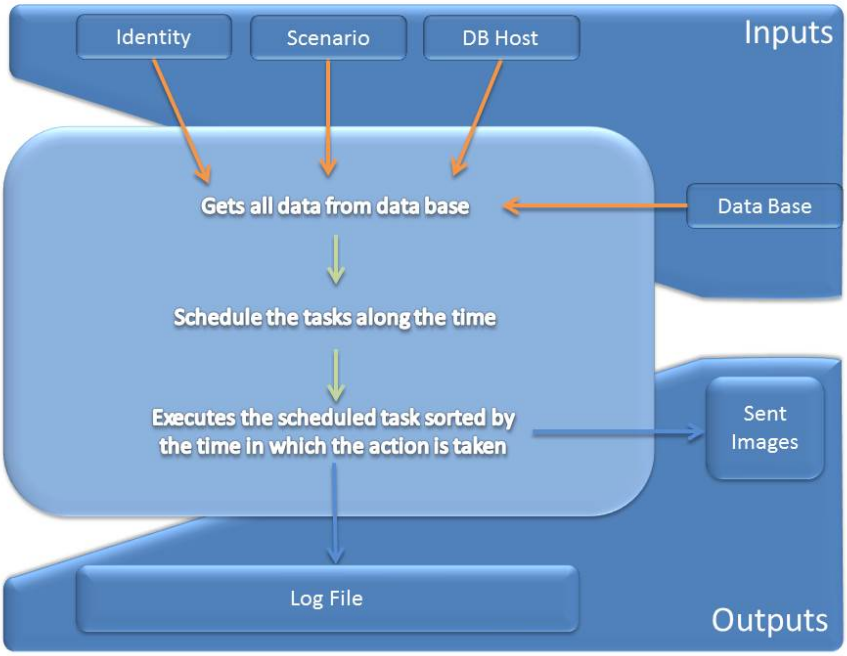


Figure 5.11: Satellite Simulator Workflow.

5.2.5.1.1.1 Schedule of data download process

Definitions:

Regarding the images download process, the satellites should download at $160\ Kbps$ to the ground station. In *Virtual Wall* we have a limited bandwidth at $100\ Kbps$. To solve this problem we designed several types of packets (with one byte size), which are sent between *Virtual Wall* nodes instead of real satellite imagery. The packets contain all the metadata of the images: *AOI*, non *AOI*, area between visibility cones. These packets are the following:

- *Packet "U":* The transmission of this packet means that the satellite sends *AOI* images at $98.6\ Mbps$ and *non AOI* images at $64.1\ Mbps$.
- *Packet "I":* The transmission of this packet means that the satellite sends *non AOI* images at $160\ Mbps$.
- *Packet "B":* The transmission of this packet means that the satellite sends *AOI* images at $160\ Mbps$.

To schedule the data download process the following functions were defined:

- *NotInterestingZone:* it represents the non *AOI* area that is being recorded by the satellite when it is inside a visibility cone. It creates a new connection with the *Ground Station Simulator* and transmits packets type "I". The inputs of this function are the following:
  - *ipGroundStation: IP address* of the *Ground Station Simulator*.
  - *t_ini:* start time in which the function is executed.
  - *t_end:* end time in which the function is finished.

  The function has implemented the pseudo-code in Listing 5.3.

```
function NotInterestingZone (t_ini,t_end,ipGroundStation):
offset <-0 //deviation of the normal time
socket <- connect(ipGroundStation) //connection with the GS
current_t <- time() //get the current system time
penal_times <-0 // times that time function is called
time_penalty <- time that time() costs
while (current_t  <  t_end + offset):
        Socket.send('I')
        Penal_times <- penal_times+2
        Offset = penal_times * time_penalty
        Time.sleep(0.2-(time()-t_temp))
        t_temp <- time()
end while
socket.close()
```

Listing 5.3: Pseudocode of *NotInterestingZone* function.

- *InterestingZone:* It represents the *AOI* area that is being recorded by a satellite when it is inside a visibility cone. It creates a new connection with the *Ground Station Simulator* and transmits packets type "U" and "B". The inputs of this function are the following:
  - *ipGroundStation: IP address* of the *Ground Station Simulator*.
  - *t_ini:* start time in which the function is executed.
  - *t_end:* end time in which the function is finished.
  - *offset_time:* It is the difference $([T_{GS}]_{0ij} - [T_{AOI}]_{0i}) * CR$.

  The function has implemented the pseudo-code in Listing 5.4:

```
function InterestingZone (t_ini,t_end,offset_time,ipGroundStation):
offset <-0 //deviation of the normal time
socket <- connect(ipGroundStation) //connection with the GS
current_t <- time() //get the current system time
penal_times <-0 // times that time function is called
time_penalty <- time that time() costs

while (current_t <  t_ini+offset_time + offset):
        socket.send('B')
        penal_times <- penal_times+2
        offset = penal_times * time_penalty
        time.sleep(0.2-(time()-current_t))
current_t<- time()
end while
while (current_t <  t_end + offset):
        socket.send('U')
        penal_times <- penal_times+2
        offset = penal_times * time_penalty
        time.sleep(0.2-(time()-current_t))
current_t<- time()
end while
socket.close()
```

Listing 5.4: Pseudocode of *InterestingZone* function.

- *OutOfVisibility:* The satellite is not in the visibility cone of a ground station, the satellite follows the orbit until it enters into a visibility cone. It does not transmit any packet. The inputs of this function are the following:
  - *t_ini:* start time in which the function is executed.
  - *t_end:* end time in which the function is finished.

  The function has implemented the pseudo-code in Listing 5.5:

Scheduling process:

This process consists of the scheduling all the satellite interactions with the ground stations. This is done as follows:

```
function OutOfVisibility (t_ini,t_end):
offset <-0 //deviation of the normal time
socket <- connect(ipGroundStation) //connection with the GS
current_t <- time() //get the current system time
penal_times <-0 // times that time function is called
time_penalty <- time that time() costs
while (current_t  <  t_end + offset):
        penal_times <- penal_times+2
        offset = penal_times * time_penalty
        time.sleep(0.2-(time()-t_temp))
        t_temp <- time()
end while
socket.close()
```

Listing 5.5: Pseudocode of *OutOfVisibility* function.

1. If the area the satellite is flying over is not *AOI*, the function *NotInterestingZone* is scheduled to be executed as follows:
   NotInterestingZone(ipGroundStation, $[T_{GS}]_{0ij}$*,* $[T_{GS}]_{fij}$*)*. The function *OutOfVisibility* is scheduled to be executed as follows:
   OutOfVisibility($[T_{GS}]_{fij}$*,* $[T_{GS}]_{(0i(j+1)}$*))*, and the scheduling process finishes.

2. Otherwise three cases can take place:

   - If $[T_{AOI}]_{0i} < [T_{GS}]_{0ij} || [T_{AOI}]_{0i} = [T_{GS}]_{0ij}$ the function *InterestingZone* is scheduled to be executed as follows: *InterestingZone(ipGroundStation,* $[T_{GS}]_{0ij}$*,* $[T_{AOI}]_{fij}$ *, offset_time)*. It is always considered that $[T_{AOI}]_{fi} > [T_{GS}]_{fij}$.
   - If $[T_{AOI}]_{0i} > [T_{GS}]_{0ij}$, first, the function *NotInterestingZone* is scheduled to be executed as follows: *NotInterestingZone(ipGroundStation,* $[T_{GS}]_{0ij}$*,* $[T_{AOI}]_{0ij}$*)*; and second, the function *InterestingZone* is scheduled to be executed as follows: *InterestingZone(ipGroundStation,* $[T_{AOI}]_{0ij}$*,* $[T_{AOI}]_{fij}$*, 0)*.

3. The *NotInterestingZone* function is scheduled to be executed as *NotInterestingZone(ipGroundStation,*$[T_{AOI}]_{fij}$*,* $[T_{GS}]_{fij}$*)*.

4. The function *OutOfVisibility* is scheduled to be executed as follows:
   OutOfVisibility($[T_{GS}]_{fij}$*,* $[T_{GS}]_{(0i(j+1)}$*))*, and the scheduling process finishes.

Figure 5.12 graphically shows the scheduling process explained above.

The activity diagram in *UML* format of the *Satellite Simulator* is depicted in Figure 5.13.

5.2.5.1.2 Satellite Simulator Implementation

The implementation was done in Python 2.7. The python's libraries needed to implement the software are listed in Table 5.8.

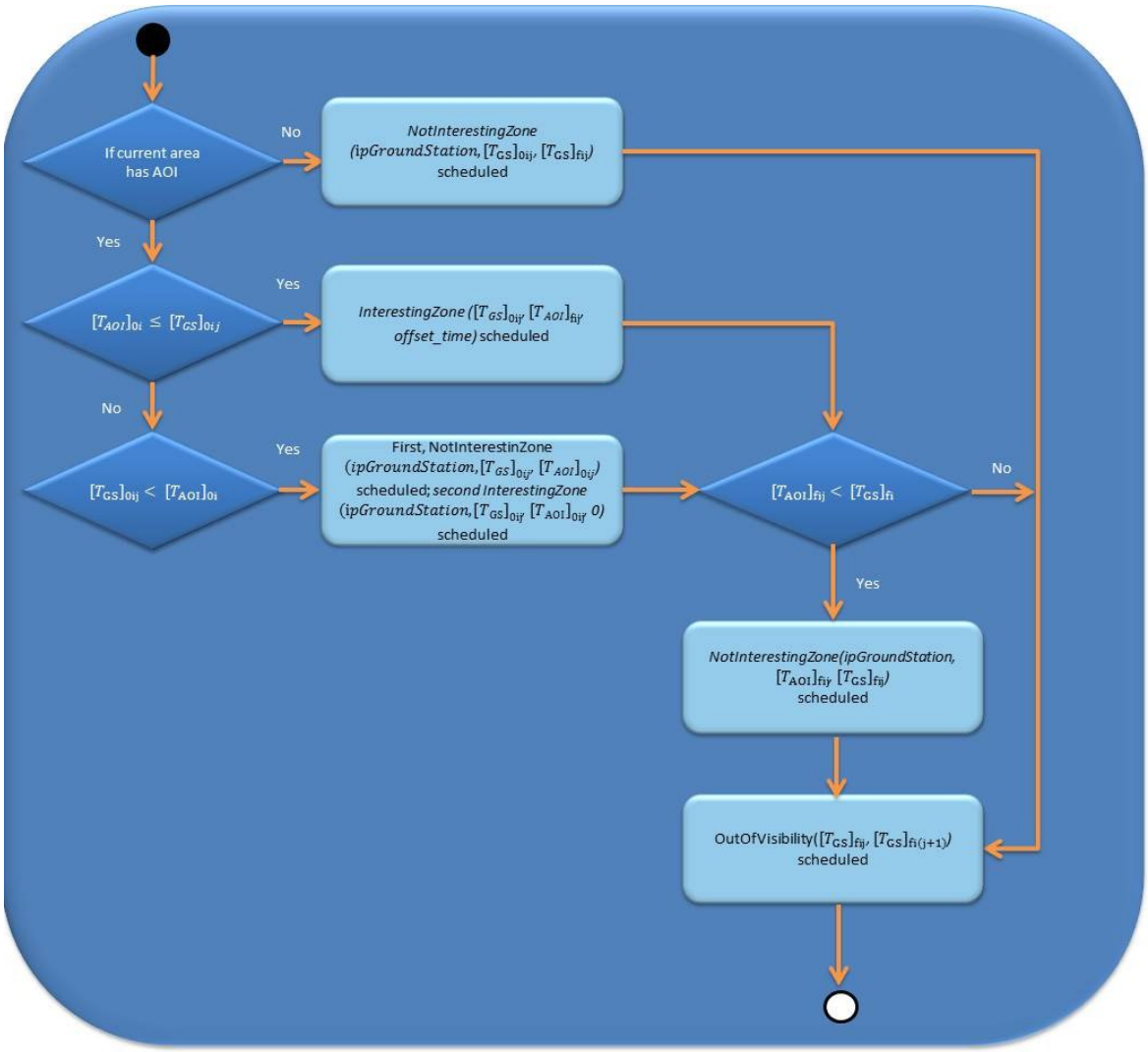


Figure 5.12: Sheduling Process on the Satellite Simulator.

When we installed Python, all the libraries were installed with the exception of "MySQLdb", which was manually installed.

Furthermore, STK provided the time in the next formats:

- *UTCG format:* Universal Time Coordinated in Gregorian format.
- *Format in seconds:* total of seconds

They are the absolute time from the beginning of the constellation simulation in the STK software. In the data base, the stored times are implemented in seconds. However to simulate a scenario in real time is too long. It can be shortened by computing a relative time (T_r):

$$T_r = T/n \tag{5.10}$$

where T is the absolute time and n a factor that scales the absolute time to reduce the

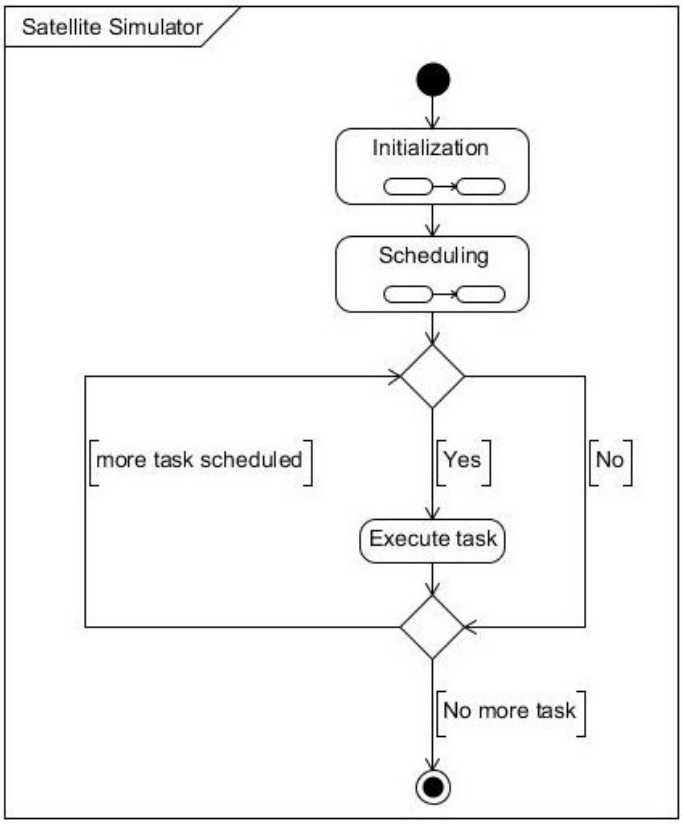


Figure 5.13: Satellite Simulator Activity Diagram.

execution time of the experiment. In preliminary simulations we used $n = 10$, although during the experiment execution stage this will be evaluated and probably changed. The Table 5.9 shows the absolute times and relative times for each scenario duration.

5.2.5.1.3 Execution

To execute the *Satellite Simulator Software* the following dependencies are required:

- Operative System based in *GNU Debian*.
- Python v.2.7
- Python packages (Table 5.8).
- Ethernet interface for the network connection.
- Connectivity with the data base located in *BonFIRE* through the network.

The *Satellite System Simulator* software is developed in a multiplatform language, but it is restricted to be executed in *UNIX* operative systems because there are many dependencies with some packages and the file system.

The execution of the satellite software has to be done as *super-user* in *UNIX*. It is executed with the following command line:

Python Library	Function
Sys	System library
OS	Operative system interactions supply
Sched	Library that allow the Satellite Simulator to schedule the task along the time
Time	For managing the time
Socket	Library for creating and establishing connections with other host
Pdb	Used for debugging the software
Logging	Log

Table 5.8: Satellite Simulator's Python Libraries.

> python satellite.py <IDSAT> <SCENARIO> <DBHOST> [LOGLEVEL]

where:

- *IDSAT* is the satellite identity. This value must be an integer.
- *SCENARIO* is the scenario number to simulate. Must be an integer.
- *DBHOST* is the host where the data base is located. Must be a hostname or an *IP address*.
- *LOGLEVEL* is the level of log that the software will show. The values can be: INFO, DEBUG. This parameter is optative. By default its value is INFO.

5.2.6 Ground Station System Simulator

In this section, the implementation and the architecture of the *Ground Station Simulator* is described. The *Ground Station System Simulator* reproduces the behaviour of the set of 12 ground stations by replicating 12 times a *Ground Station Simulator* that renders individual ground stations behaviour.

The *Ground Station System Simulator* is a manager that executes 12 *Ground Station Simulators*. The *Ground Station System Simulator* requires the scenario number and the *IP address* of the database as an input and executes the *Ground Station Simulators* by providing them the specific identity of the ground station *(idGroundStation)*, number of the scenario *(S)* and *IP address* of the distributed database *(ipDatabase)*.

5.2.6.1 Ground Station Simulator

The functions of the ground stations are the following:

Scenario	Absolute Time (T_0, T_f) (Seconds)	Relative Time (T_{r0}, T_{rf}) (Seconds)
Scenario 1: Emergencies – Lorca Earthquake (Spain)	(63638, 63661)	(6363.8, 6366.1)
Scenario 2: Infrastructure monitoring. Affection in railway infrastructures by sand movement in desert areas (Spain)	(54060, 54753)	(5406.0, 5475.3)
Scenario 3: Land Management – South West of England	(62480, 63865)	(6248.0, 6386.5)
Scenario 4: Precision Agriculture – Argentina	(78461, 84999)	(7846.1, 8499.9)
Scenario 5: Basemaps – Worldwide	(0, 86400)	(0, 8640.0)

Table 5.9: Scenarios relative times.

- To receive the images sent by the satellites.
- To create the files with the raw images that the *Orchestrator* will download for processing and publishing in the *Archive and Catalogue* subsystem.

The *Ground Station Simulator* software is common for all the ground stations, but it is parameterized to define each of them, similarly to the *Satellite Simulator*.

The *Ground Station Simulator* is constituted by the following components:

1. *Initialization Module:* This module obtains the following parameters and initializes the *Ground Station Simulator*:
   - *S:* the number of the scenario to simulate.
   - *idGroundStation:* identification number of the ground station (1 to 12).
   - *ipDatabase: IP address* of the database.

   Once obtained the previous parameters, the *Initialization Module* connects with the database and fetches the name of the ground station. Also, a query to update the *IP address* and port of the *Ground Station Simulator* server is sent to the database.
2. *Ground Station Dynamics Module:* This module represents the dynamics and associated parameters of the ground station. It requires *S*, *ipDatabase* and *idGroundStation*.

Figure 5.14 shows the relation between the previous modules.

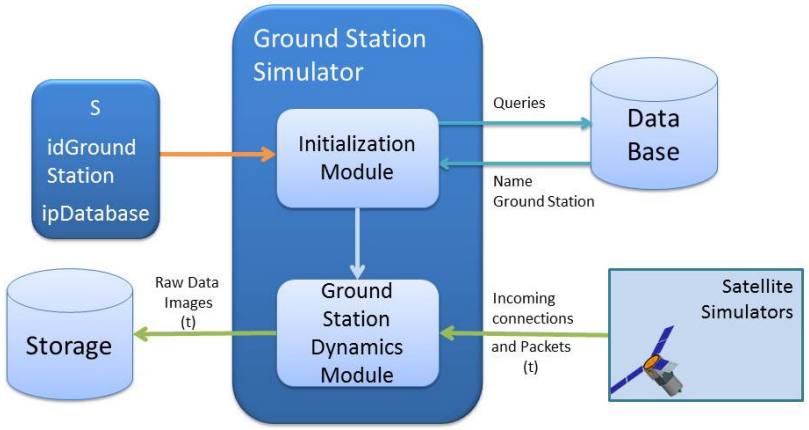


Figure 5.14: Ground Station Simulator Architecture.

5.2.6.2 Ground Station Simulator Workflow

The *Ground Station Simulator* starts the execution following these steps:

1. The *Initialization Module* obtains *S*, *idGroundStation* and *ipDatabase*.
2. Then, the *Initialization Module* queries on the database for the name of the ground station and for updating the *IP address* and *port* of the *Ground Station Simulator*.
3. The *Ground Station Dynamics Module* starts:
   - A network socket[3] is created for listening input connections from *Satellite Simulators*.
   - Every time a satellite enters into the visibility cone of the ground station a connection between a *Satellite Simulator* and the *Ground Station Simulator* is created. Note that if several satellites are in the same visibility cone, the *Ground Station Simulator* opens a connection for each *Satellite Simulator*, and that the *Ground Station Simulator* can listen to new connections if new satellites enter into the visibility cone.
   - The *Ground Station Simulator* creates a new process to keep the previous connection or connections (if there are more than one satellite in the visibility cone) open.
   - The new process or processes receive the packets sent by the *Satellite Simulators*.
   - Once the transmission between the *Satellite Simulators* and the *Ground Station Simulator* finished the connections are closed.
   - Then the received packets are counted and classified by type (see Section 5.2.5.1.1.1):

[3]Network socket is an endpoint of an inter-process communication flow across a computer network. Most of the communications between computers are based on the Internet Protocol using network sockets.

- *Packet "U":* 98.6 Mb are added to the *AOI* buffer and 64.1 Mb are added to the *non AOI* buffer in the *Ground Station Simulator*.
- *Packet "I":* 160 Mb are added to the *non AOI* buffer in the *Ground Station Simulator*.
- *Packet "B":* 160 Mb are added to the *AOI* buffer in the *Ground Station Simulator*.

After classifying the packets, the new processes finish.

- The *AOI* images and *non AOI* images are created in the *Ground Station Simulator*.

4. The *Ground Station Simulator* server ends when it receives the *SIGINT* signal.

Figure 5.15 shows the process explained above. The diagram also shows the interactions with the other subsystems (see Figure 5.14). Figure 5.15 shows the inputs for the *Ground Station Simulator* and the outputs after its execution. These outputs are the Raw Images in time created and a log file that contains the information about the execution.

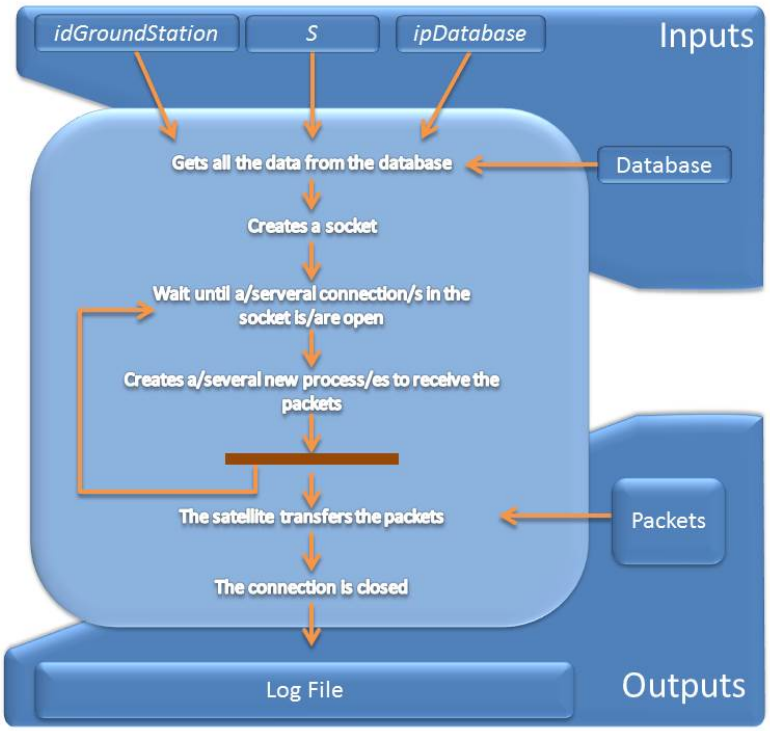


Figure 5.15: Ground Station Simulator Workflow.

The activity diagram of the Ground Station Simulator in UML format is depicted in Figure 5.16.

5.2.6.2.1 Implementation

The implementation was done in Python 2.7, using the libraries described in Table 5.10.

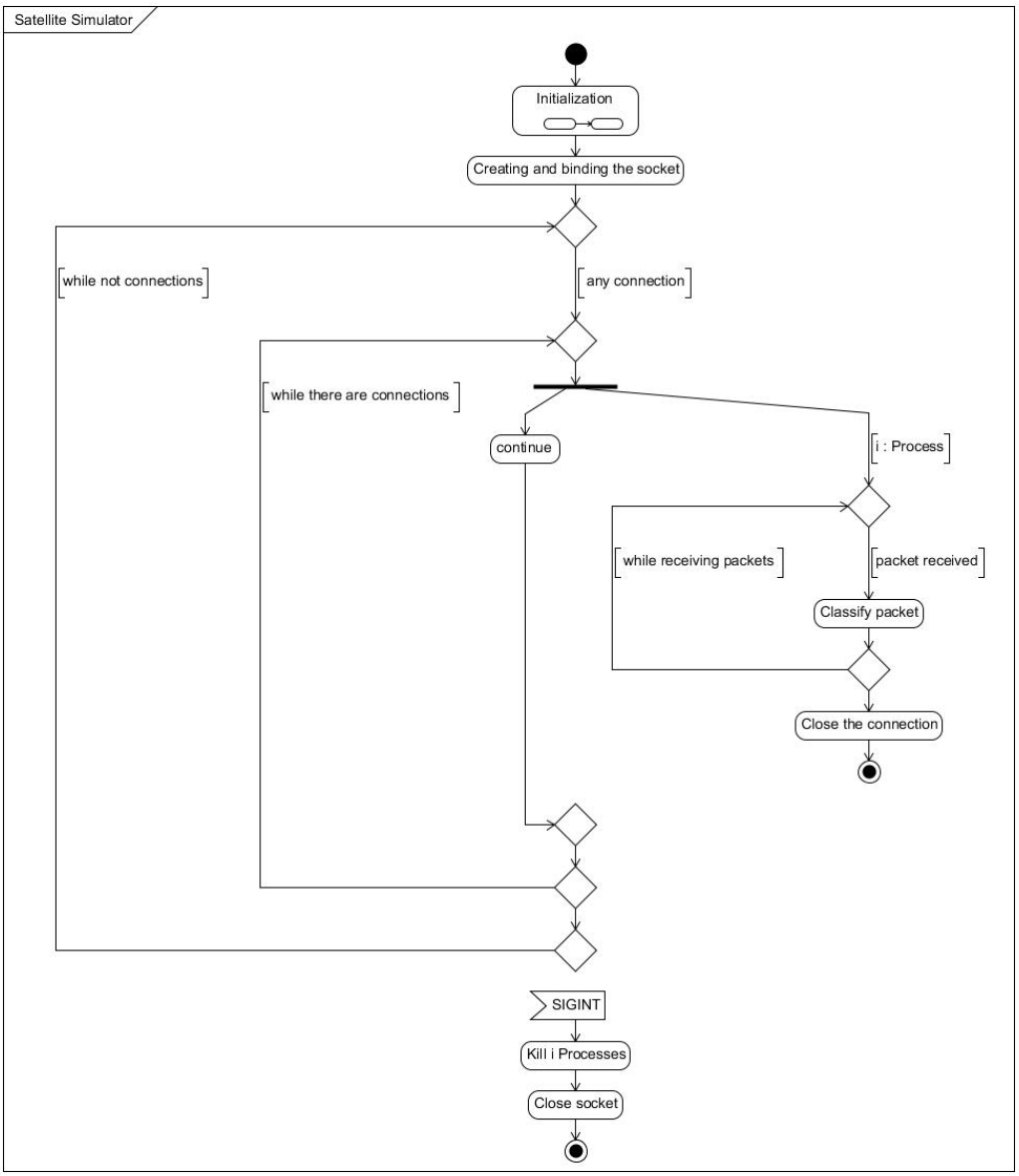


Figure 5.16: Ground Station Simulator Activity Diagram.

5.2.6.2.2 Execution

To execute the Ground Station Simulator Software the following dependencies are required:

- Operative System based in *GNU Debian*.
- Python v.7.
- Python packages: (shows in Table 5.10).
- Ethernet interface for the network connection.
- Connectivity with the data base located in *BonFIRE* through the network.

The *Ground Station Simulator* is developed in a multiplatform language, but it is restricted to be executed in *UNIX* operative systems because there are many dependencies with some

Python Library	Function
Sys	System library
OS	Operative system interactions supply
Sched	Library that allow the Satellite Simulator to schedule the task along the time
Time	For managing the time
Socket	Library for creating and establishing connections with other host
Pdb	Used for debugging the software
Logging	Log

Table 5.10: Ground Station Simulator Python Libraries.

packets and the file system.

The software also needs to know the *IP addresses* assigned to its Ethernet interface. If this interface is not connected, the software looks for the WLAN0 interface. This information is accomplished from the *UNIX* file "/etc/hosts".

The execution of the satellite software must be done as sudo user as follows:

> python groundstation.py <IDSAT> <SCENARIO> <DBHOST> [LOGLEVEL]

where:

- *IDSAT* is the ground station identity. This value must be an integer.
- *SCENARIO* is the scenario to simulate. It must be an integer.
- *DBHOST* is the host where the data base is located. It must be a hostname or an *IP address*.
- *LOGLEVEL* is the level of log that the software will show. The values can be: INFO, DEBUG. This parameter is optative. Its value is INFO by default.

5.3 Implementation in Virtual Wall

In this section, the implementation of the *Space System Simulator* in *Virtual Wall* is explained. First, the designed topology and the new modified topology (different to the design one) are layed out because the handicaps of the hardware during the implementation. Then, nodes reservation is explained and detailed. Finally, the scripts included in each type of node are described before the presentation of the execution.

5.3.1 Topology network

The *Space System Simulator* is comprised of the following modules:

- The *Satellite System Simulator*
- The *Ground Station System Simulator*

The topology network designed is depicted in Figure 5.17.

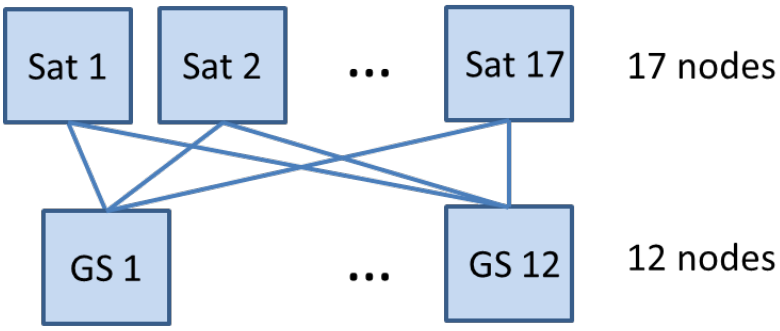


Figure 5.17: Topology Network in *Virtual Wall*

The previous topology network involves Sat nodes to have 12 connections with the GS nodes; and GS nodes 17 connections with Sat nodes plus one connection with the *BonFIRE* cloud.

The nodes in *Virtual Wall* have a limitation of 5 physical connections. To deploy the previous topology we adapted the *Space System Simulator* software to establish those connections by software. This was done by making the *Satellite System Simulator* flexible to connect with any ground station from GS1 to GS12.

The solution was to multiply the *Satellite System Simulator* by 3 and connect 4 ground stations to each *Satellite System Simulator* . This ensures the same performance and dynamics of the previous topology network described in Figure 5.17, and the only thing it changes is the software.

The *Space System Simulator* implemented in *Virtual Wall* is depicted in Figure 5.18.

5.3.1.1 Nodes Reservation and Setup

The *Space System Simulator* is then constituted by 15 "Generic nodes" in *Virtual Wall* 1 (The *Virtual Wall* testbed is composed by two sets of machines, *Virtual Wall* 1 and *Virtual Wall* 2). The configuration of the experiment makes the provisioning of "any available node", which is more flexible in case a node is being used in other experiment (see Figure 5.19). The links between SS *(Satellite Simulators)* and GS *(Ground Stations Simulators)* nodes are TCP. The definition of the dependencies and the commands that will be installed at setup is defined in *JFed* using a *Rspec* specification. This specification includes some bash

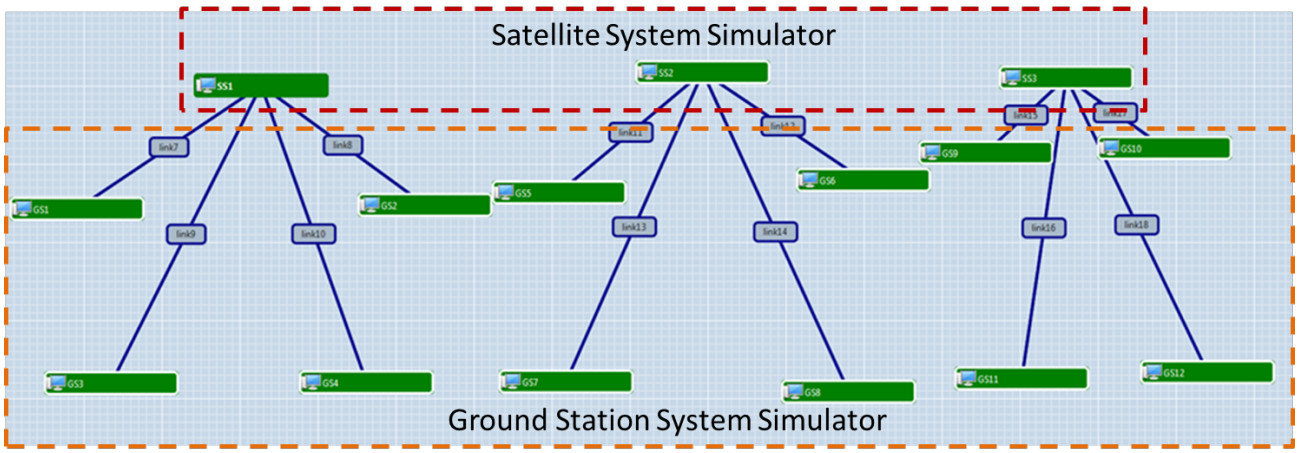


Figure 5.18: Topology Network in *Virtual Wall*

commands or instructions that will execute after reservation automatically.

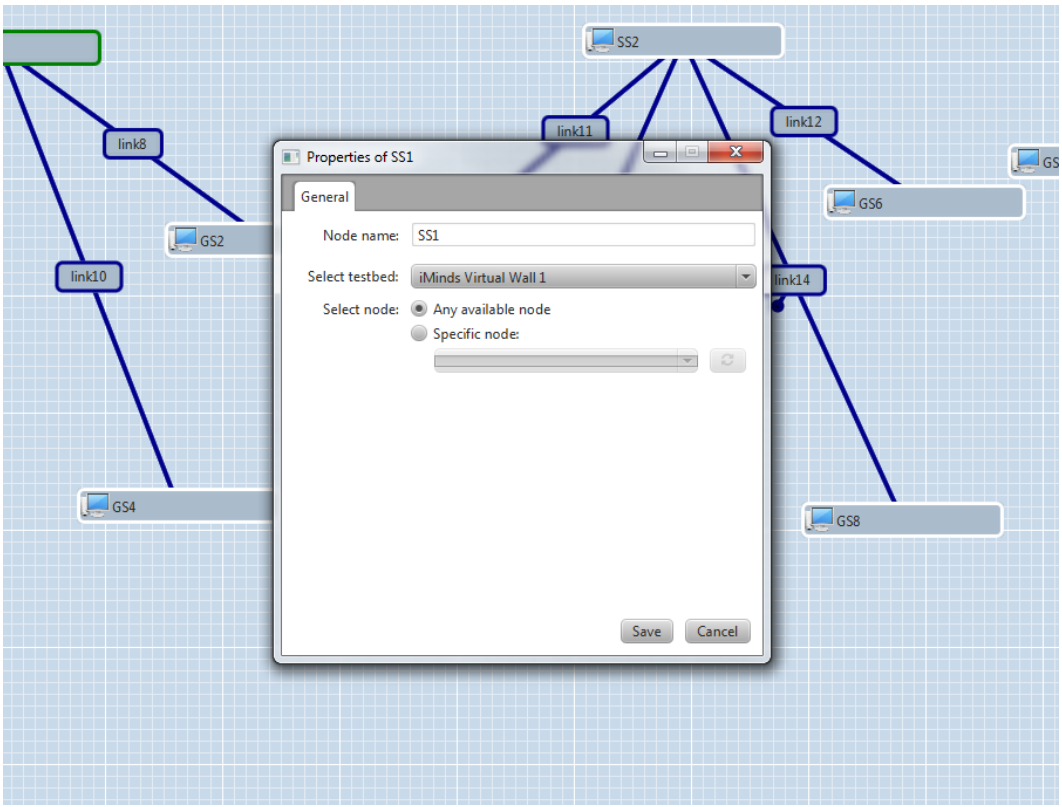


Figure 5.19: Configuration of *Virtual Wall* nodes

5.3.1.1.1 Satellite Simulator Nodes Setup

The configuration of the nodes is done by using *JFed Rspec Experiment*. The setup steps are the following:

- To configurate a gateway for obtaining Internet connectivity and installing the needed

libraries.

- To update the system.
- Once the update is finished, dependencies can be installed.
- As the simulators require a connection with a database, the *MySQL* library is also required.
- Source is downloaded from Google Drive where is located.

The script that perform the above is showed in Listing 5.6.

```
<node client_id="SS1" component_manager_id="urn:publicid:IDN+wall1.ilabt.iminds.be+authority+cm" exclusive="true">
 <sliver_type name="raw-pc"/>
 <services>
  <execute command="sudo route del default gw 10.2.15.254" shell="sh"/>
  <execute command="sudo route add default gw 10.2.15.253" shell="sh"/>
  <execute command="sudo apt-get update" shell="sh"/>
  <execute command="sudo apt-get install python python-mysqldb -y" shell="sh"/>
  <execute command="sudo wget --no-check-certificate https://googledrive.com/host/0B7vtkY3QTQrkZ0RXNm1KcEs0ZW8 -O /users/jbecedas/satellite.py" shell="sh"/>
  <execute command="sudo wget --no-check-certificate https://googledrive.com/host/0B7vtkY3QTQrkXzF4OERoNGJGbzg -O /users/jbecedas/groundstation.py" shell="sh"/>
  <execute command="sudo wget --no-check-certificate https://googledrive.com/host/0B7vtkY3QTQrkYi1GTnZyU0o3Tlk -O /users/jbecedas/push_dbIP.sh" shell="sh"/>
  <execute command="sudo bash /users/jbecedas/add_wanBF.sh" shell="sh"/>
  <execute command="sudo bash /users/jbecedas/push_dbIP.sh 129.215.175.148" shell="sh"/>
 </services>
```

Listing 5.6: Rspec specification for *Satellite Simulators*

Where «google_drive_host_address» is the address of the Google Drive URI in which the scripts are located.

The *Database* is located in *BonFIRE* and it has an *IP address* that can change. In order to obtain that *IP address* and include it in the *Satellite Simulator*, a simple script to acquire the *IP address* of the database was developed *push_dbIP.sh*. Listing 5.7 contains the script.

```
#!/bin/bash
```

```
3 ip=$1

5 touch /users/jbecedas/ipdb
6 echo $ip > /users/jbecedas/ipdb
```

Listing 5.7: Bash script to write the Database's *IP address* on a file

The execution of the *push_dbIP.sh* script acquires the *IP address* of the database and includes it in the file *ipdb*, created by *push_dbIP.sh*. Finally, the node is perfectly configured to proceed with the installation of the *Satellite Simulator*. The software of the simulator has been archived too in *Google Drive*. The software is acquired and it can now be executed in the node.

5.3.1.2 Ground Station Simulator Nodes Setup

The *Ground Station Simulators* software is developed under Python. The steps of setup are the following:

- To configurate a gateway for obtaining Internet connectivity and installing the needed libraries.
- To update the system.
- Once the update is finished, dependencies can be installed.As occurred with the SS nodes, the update and the installation of Python and MySQL are required.
- As the simulators require a connection with a database, the *MySQL* library is also required.
- Source is downloaded from Google Drive where is located.
- The database IP address is added to *ipdb* file.
- The raw data is located in Google Drive too.
- The GS nodes are required to open an *FTP* to be accessed from the *Orchestrator* in *BonFIRE* . The *FTP* is configured by executing the *install_ftp.sh* file. This file content is in Listing 5.9.

The script that perform the above is showed in Listing 5.8.

```
1         <services>
2             <execute command="sudo route del default gw 10.2.15.254"
                 shell="sh"/>
3             <execute command="sudo route add default gw 10.2.15.253"
                 shell="sh"/>
4             <execute command="sudo apt-get update" shell="sh"/>
5             <execute command="sudo apt-get install python python-
                 mysqldb -y" shell="sh"/>
```

```
6               <execute command="sudo wget --no-check-certificate https
                    ://googledrive.com/host/0B7vtkY3QTQrkZ0RXNm1KcEs0ZW8 -
                    O /users/jbecedas/satellite.py" shell="sh"/>
7               <execute command="sudo wget --no-check-certificate https
                    ://googledrive.com/host/0B7vtkY3QTQrkXzF4OERoNGJGbzg -
                    O /users/jbecedas/groundstation.py" shell="sh"/>
8               <execute command="sudo wget --no-check-certificate https
                    ://googledrive.com/host/0B7vtkY3QTQrkcVpzbll3eTk5djg -
                    O /users/jbecedas/add_wanBF.sh" shell="sh"/>
9               <execute command="sudo wget --no-check-certificate https
                    ://googledrive.com/host/0B7vtkY3QTQrkMVVoWWJ6b3FtNG8 -
                    O /users/jbecedas/install_ftp.sh" shell="sh"/>
10              <execute command="sudo bash /users/jbecedas/add_wanBF.sh"
                    shell="sh"/>
11              <execute command="sudo bash /users/jbecedas/install_ftp.sh
                    " shell="sh"/>
12              <execute command="sudo wget --no-check-certificate https
                    ://googledrive.com/host/0B7vtkY3QTQrkYi1GTnZyU0o3Tlk -
                    O /users/jbecedas/push_dbIP.sh" shell="sh"/>
13              <execute command="sudo bash /users/jbecedas/push_dbIP.sh
                    129.215.175.148" shell="sh"/>
14              <execute command="sudo touch /tmp/original.bin" shell="sh
                    "/>
15          </services>
```

Listing 5.8: Rspec specification for *Ground Station Simulators*

```
1  #!/bin/bash
2
3  sudo apt-get update
4  sudo apt-get install mysql-client ftp ftplib3 vsftpd -y
5  sudo sed -i "s/listen=YES/listen=NO/" /etc/vsftpd.conf
6  sudo sed -i "s/#listen_ipv6=NO/listen_ipv6=YES/" /etc/vsftpd.conf
7  sudo sed -i "s/#listen_ipv6=YES/listen_ipv6=YES/" /etc/vsftpd.conf
8  sudo sed -i "s/anonymous_enable=YES/anonymous_enable=NO/" /etc/vsftpd.
       conf
9  sudo sed -i "s/#local_enable=NO/local_enable=YES/" /etc/vsftpd.conf
10 sudo sed -i "s/#local_enable=YES/local_enable=YES/" /etc/vsftpd.conf
11 sudo sed -i "s/#write_enable=NO/write_enable=YES/" /etc/vsftpd.conf
12 sudo sed -i "s/#write_enable=YES/write_enable=YES/" /etc/vsftpd.conf
13 sudo sed -i "s/#local_umask/local_umask/" /etc/vsftpd.conf
14 sudo sed -i "s/#chroot_list_file/chroot_list_file/" /etc/vsftpd.conf
15 sudo mkdir -p /home/ftp/deimos
16 sudo mkdir /home/deimos
17 sudo useradd -d /home/deimos -g ftp  deimos
18 sudo chmod 777 -R /home
19 sudo touch /etc/vsftpd.chroot_list
```

```
20 sudo su
21 echo "deimos:deimos"|chpasswd
22 echo "deimos" >> /etc/vsftpd.chroot_list
23 sudo service vsftpd restart
```

Listing 5.9: FTP server installation

Through the FTP connection the raw data will be transferred from the GS nodes to the BonFIRE cloud.

The execution of the Satellite Simulators and the Ground Station Simulators are described in Section 5.2.5.1.3 and Section 5.2.6.2.2.

5.4 Cloud Architecture

In this section, the EO architecture on cloud is fully described. As mentioned, the cloud infrastructure in which this system is implemented is provided by the *BonFIRE* multi-cloud testbed. The *Orchestrator*, the *Archive and Catalogue* and the *Processing Chain* integrate the cloud components for processing, archiving and cataloging Earth's surface images. This modules are summarized as follows:

- *Orchestrator:* it manages the automatic distribution of the raw data to the processors. It handles the complete automatic processing chain execution. If the processor chain is occupied, the manager replicates the complete chain in a new machine.

- *Processing Chain:* it processes the raw data and converts it in different image products.

- *Archive and catalogue:* it is the place where the processed images are stored and catalogued for its distribution.

There are two implementations of this architecture: the first one uses SSH and Secure Copy (SCP) for communications and the processing is limited to one proccesing chain each time; the second one is based in the distributed middleware ZeroC ICE and it takes advantage of the use of the location transparency, distribution and replication service in order to satisfy all the request for image processing. This last implementation is in a prototype stage. Its final implementation is not objective of this project and it is contemplated in future works.

In the following sections the componets which appear in both architectures such as the *Orchestrator*, the *ProcessingChain* and the *Archive and Catalogue* modules are explained. Then, both architectures and their components are deeply studied. In each architecture, the architecture design is elaborated and then, the communications between the components, their implementation and their executions in *BonFIRE* are described.

5.4.1 Cloud components in GeoCloud

5.4.1.1 Orchestrator

The *Orchestrator* is the component that manages the tasks to be done in the cloud. It is running over the *BonFIRE* Cloud and controls all the interactions between all the components implemented in the *BonFIRE* testbed.

The *Orchestrator* has the following functions:

- To identify which outputs shall be generated by the processors.
- To generate the Job Orders. They contain all the necessary information that the processors need. Furthermore these XML files include the interfaces and addresses of the folders in which the input information to the processors is located and the folders in which the outputs of the processors have to be sent. They also include the format in which the processors generate their output.
- To look for raw data in the ground stations (pooling) to ingest such raw data in a shared storage unit in the cloud for its distribution to the processing chain.
- To control the processing chain by communicating with the product processors.
- To manage the archive and catalogue.

The orchestrator interacts with different modules:

- Ground stations implemented in *Virtual Wall* .
- Processing instances in the cloud.
- Archive and catalogue.

Figure 5.20 depicts the *Orchestrator's* interactions with the other modules of the GEO-Cloud architecture.

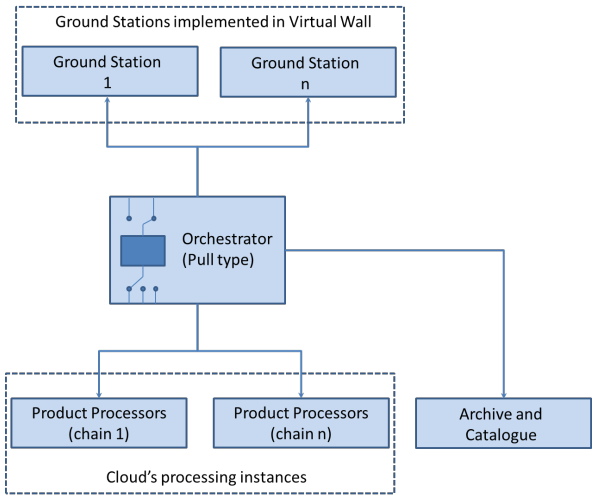


Figure 5.20: *Orchestrator* interactions.

As shown in Figure 5.20, the *Orchestrator* is pooling the Ground Stations frequently. When the *Orchestrator* gets the data, it uses the Product Processor for processing the data to generate the resulting image. When this processing is finished, the *Orchestrator* sends the image to the *Archive and Catalogue* to be available for customers.

5.4.1.2 Processing Chain

The *Processing Chain* is a module which is in charge of the processing of the payload raw data from the satellites to produce image products. The four most important operations that the product processors perform on the input data are the following:

- A calibration, to convert the pixel elements from instrument digital counts into radiance units.
- A geometric correction, to eliminate distortions due to misalignments of the sensors in the focal plane geometry.
- A geolocation, to compute the geodetic coordinates of the input pixels.
- An ortho-rectification, to produce ortho-photos with vertical projection, free of distortions.

The previous steps also generate quality-related figures of merit that are made available in all the products. Moreover, the product processors generate metadata, in line with industry standards, to facilitate the cataloguing, filtering and browsing of the product image collection. These processors are considerated as black boxes because they are owned by Elecnor Deimos and their design and implementation can not be published, but them were studied for carrying out this project.

The output image products are classified into four different levels, according to the degree of processing that they have been subjected to (see Figure 5.21):

- *Level 0* products are unprocessed images, in digital count numbers.
- *Level L1A* products are calibrated products, in units of radiance.
- *Level L1B* products are calibrated and geometrically corrected products (ortho-rectified), blindly geolocated.
- *Level L1C* products are calibrated and geometrically corrected products (ortho-rectified), precisely geolocated using ground control points.

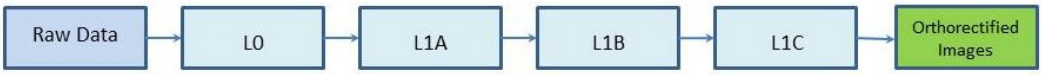


Figure 5.21: Stages of the product processing.

5.4.1.2.1 The L0 Processor

The acquired data is organized into image sectors of predefined size and structure and converted in scenes. Scenes, as defined here, are used throughout the subsequent L1 levels. The size and configuration of the scene is not changed again in the processing chain, for this reason the scene definition is constant for all the L1 levels.

The inputs are the following:

- The Raw Data.
- The configuration database.
- The calibration database.

The outputs are the following:

- The L0 products.

5.4.1.2.2 The L1A Processor

The goal of Level 1A is to calibrate the scenes. The resulting images are given in units of radiance. The L1A component works on the scenes that compound the L0 product, performing different transformations over pixel values to generate radiances.

The inputs to the L1A level are the following:

- One L0 scene.
- The configuration database.
- The calibration database.

The output is the following:

- The L1A product.

5.4.1.2.3 The L1B Processor

Level 1B implements the geolocation, resampling and packing.

The inputs to the L1B level are the following:

- The L1A product.
- The configuration database.
- The calibration database.

The outputs are the following:

- The L1B products.

5.4.1.2.4 The L1C Processor

The L1C processor performs the ortho-rectification of the L1B product using ground control points.

The inputs to the L1C level are the following:

- The L1B product.
- The calibration database.
- The configuration database.

The output is the following:

- Orthorectified Images.

5.4.1.3 Archive and Catalogue

The *Archive and Catalogue* is a shared space of memory between the *Orchestrator*, the product processors and the distribution of data. It has a data acquisition component which manages the input data arriving to the *Archive and Catalogue*. The ingestion of data in this module is automatic.

In the *Archive and Catalogue* module the processed images are stored and catalogued for their distribution.

The *Archive and Catalogue* basically consists of the archive and the catalogue sub-modules:

- The *Archive* is constituted by optimized storages structure allowing managing a big amount of data, efficient storage and retrieval of any kind of file. The *Archive* is organized in hierarchical levels of storage in order to provide a cost effective storage solution.
- The *Catalogue* stores an inventory database with the metadata of archive files. It allows the product process chain easiness to access to the metadata from the processed products.
- For the added value services the catalog will be accessed by a *Web Service*.
- CSW is a module with the CSW standard for the catalogue (based on OGC standard). For more information on CSW, please refer to OGC *OpenGIS Implementation Specification 07-006r1* and the OGC tutorial on CSW. Through this standard the distribution of data is done.

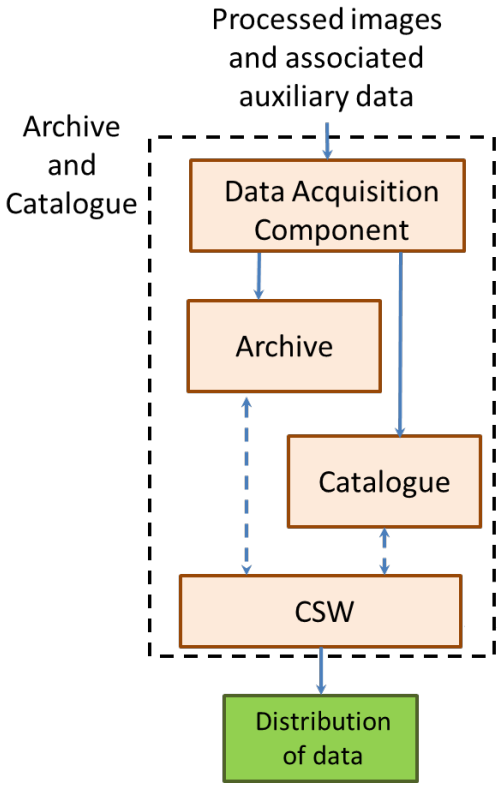


Figure 5.22: Scheme of the *Archive and Catalogue* module.

5.4.2 Implementation of the cloud architecture using SSH and SCP

The cloud implementation using SSH and SCP is based in the following components:

- The *Orchestrator* module.
- The *Processing Chain* module.
- The *Archive and Catalogue* module.

The diagram in Figure 5.23 shows the entire architecture is shown. The communications were done using SSH commands and the sending of the files were performed using the SCP protocol. For this implementation the raw data travels from the *Orchestrator* to the *Processing Chain* module, where it is processed. Finally, the Proccesing Chain module sends the processed image to the Archive and Catalogue module.

It is important to highlight that the *Orchestrator* does not send the processed image to the *Archive and Catalogue* module, the *Processing Chain* module sends it for archiving and cataloguing. Consecuently, the time for archiving and cataloguing for each processed image was reduced.

The different components which compound the cloud architecture are fully explained in the following sections. For each component, the workflow, the design, the implementation of the component and how was implemented in *BonFIRE* are detailed.

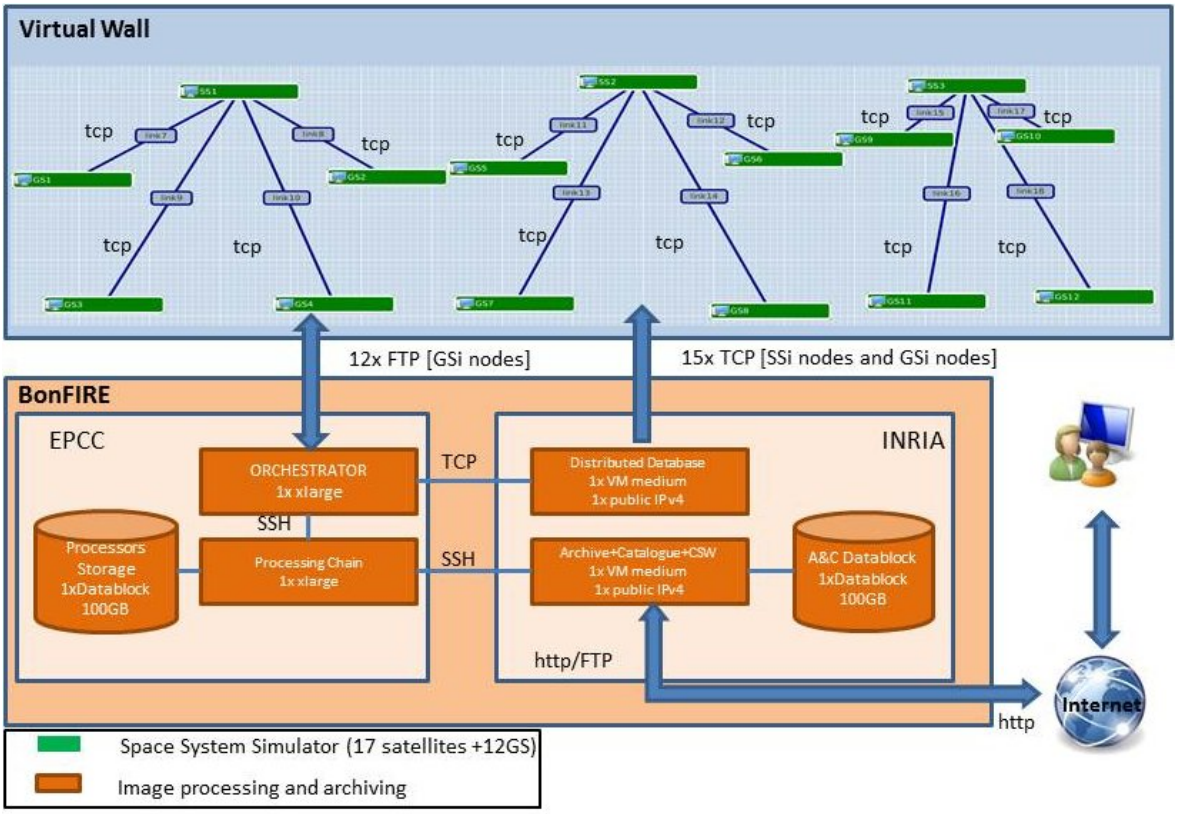


Figure 5.23: First architecture on cloud.

5.4.2.1 Implementation of the Orchestrator

5.4.2.1.1 Orchestrator Workflow

The *Orchestrator* component works by following the next sequence of steps:

1. The *LoadData object* gets all the information about the *Ground Stations Simulators* and localizes them.
2. The *Listener object* pools to the *Ground Stations* and when there are a downloadable raw data, the *New_Data_Event* is launched.
3. When the *New_Data_Event* occurs, the *Orchestrator* downloads the data.
4. The *Orchestrator* moves the raw data to a shared storage.
5. Then, the *Orchestrator* makes different *Job Orders* for the processors. The *Job Order* contains all the useful information for the *Product Processors* to proceed with the image processing.
6. The *Orchestrator* gets the *ProcessorChainController* object (this object was made regarding *Singleton pattern*).
7. The *Orchestrator* instructs the *ProcessorChainController* object to create a new processing chain by sending the *JobOrders* created in step 4. If there were any image

processing, the request is queued.

8. The *ProcessorChain Controller* object creates a new *Processing Chain* to remotely process the data.
9. The *Processing Chain* sequentially executes the L0, L1A, L1B, L1C processors.
10. When the *ProcessingChain* has finished, this notifies the *ProcessorChainController* object that the processing ended.
11. The *ProcessingChainController* alerts the *Orchestrator* that the *Processing Chain* has finished.
12. The *Orchestrator* takes the created image and puts it into the *Archive*. As a improvement, this communication was not done. The sending to archiving and cataloguing is performed when the *Processing Chain* finishes and sends the image to *Archive and Catalogue* module.

Figure 5.24 depicts the workflow of the *Orchestrator*.

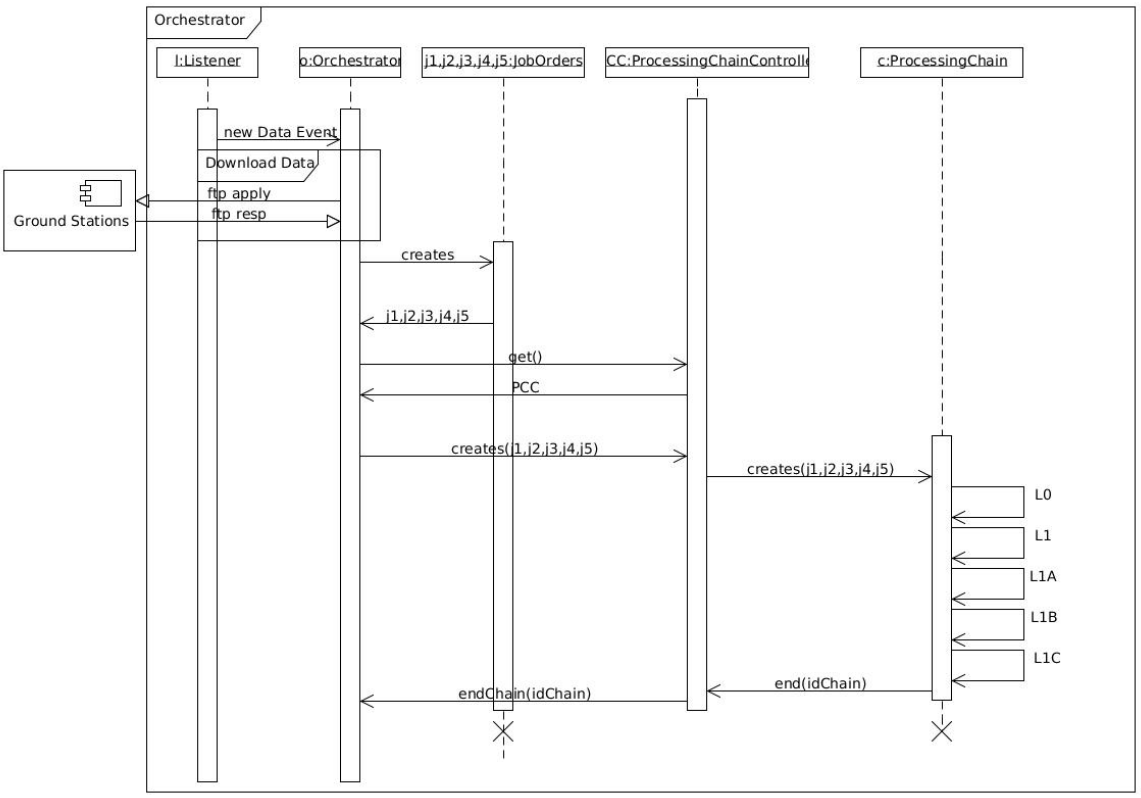


Figure 5.24: *Orchestrator* workflow.

5.4.2.1.2 Orchestrator Interfaces

The *Orchestrator* has interfaces with the Ground Stations implemented in *Virtual Wall*, with the *Product Processors* and with the *Archive and Catalogue*.

5.4.2.1.2.1 Interfaces with the Ground Stations implemented in Virtual Wall

The *Ground Stations* are deployed in some *Virtual Wall* nodes. There, the impairments and features of the network are simulated. Essentially, the *Orchestrator* is pooling those *Ground Stations* over File Transfer Protocol (FTP) connections to know when new raw data is available. So, this Ground Stations are FTP servers in which the *Orchestrator* can get the raw data obtained by the constellation of satellites.

5.4.2.1.2.2 Interfaces with the Product Processors

The *Orchestrator* communicates with the *Product Processors* through the *ProcessingChainController* instance as shown in Figure 5.24. The *Orchestrator* commands via SSH the *ProcessingChainController* to create a new processing chains to process the raw data and sends it through an SCP transmission.
When this process finishes, the *ProcessingChainController* sends a message to indicate the end of the chain to the *Orchestrator*. Thus, the *ProcessingChainController* checks the product processors progress and initiates the next level until the processing chain finishes. Finally, the *Processing Chain* obtains the end product and locates it in the Catalogue service.

5.4.2.1.3 Orchestrator Design

The Orchestrator is formed by the *Listener*, the *JobOrder* and the *ProcessingChainController*, the *Orchestrator* and the *Load* classes.

- *Listener class*: It is responsible of obtaining the *FTP* connections and polling them in order to get the images when they are created in the ground stations. When an image is detected in any ground station, a thread is created to download it. Then, the *Orchestrator* object is signaled in order to begin the processing.
- *JobOrder class*: It perform the creation of the job orders. For simplicity, a specific job order is always used in all processings of scenarios.
- *ProcessingChainController class*: It manages the processing in the product processors via SSH. The processing is remotely executed. When an image is sent by the *Orchestrator* class to the ProcessingChainController class, a new thread is created. Then, this thread remotely processes the image in the ProcessingChain module. If any request comes at same time, it is queued. This is a class based in the Singleton pattern.

- *Orchestrator class*: This class is designed by following a *Controller* pattern. It manages the interactions between all above classes and integrates them.
- *LoadData class*: it parsers the configuration file for obtaining the IP addresses of Database node, *Archive and Catalogue* node and the *Processing Chain* module. Furthermore, the user and password for FTP connections are included into the configuration file.

5.4.2.1.4 Orchestrator Implementation

The implementation of this module was done in Python 2.7. The libraries needed to implement the software are listed in Table 5.11.

Python Library	Function
Threading	System library for creating, syncronizing and managing threads
OS	Library which provides operative system interactions
Collections	Library that contains the data structure "deque" used for queueing the request which can not be processed.
Time	For managing the time
Socket	Library for creating and establishing connections with other host
Pdb	Used for debugging the software
FtpLib	Library used to obtain and manage the Ground Stations' FTP connections
MySQLlib	Library used to manage the MySQL databases

Table 5.11: Orchestrator's Python Libraries.

Moreover, it requires the SCP and SSH clients for communicating with other modules in the cloud. In addition, a script was developed in order to obtain the workload of the *Orchestrator*

machine. That file uses the "Python-psutil" library in order to obtain the CPU times such as user, idle, nice and iowait cycles. This file is used by the graphical interface for plotting the workload of the *Orchestrator* at real time.

5.4.2.1.5 Orchestrator Execution

To execute the *Orchestrator* module the following dependencies are required:

- Python v.2.7. For previous versions it has not been tested.
- Python packages listed in Table 5.11.
- Ethernet interface for the network connection in the *BonFIRE WAN*.
- Connectivity with the database located in *BonFIRE* through the network.
- To setup the "orchestrator.conf.xml" with the correct IP addresses of *Archive and Catalogue* node, Chain Processing node and database node.

The execution of the satellite software it is executed inside the "source/bonfire/orchestrator" with the following command line:

```
> python main.py
```

5.4.2.1.6 Implementation in BonFIRE

For implementing of the *Orchestrator* the following steps were done:

- Reservation of a *Xlarge* machine in *EPCC BonFIRE* platform. This machine belongs the *BonFIRE WAN* so it can communicate with all machines in that network.
- Instalation of the necessary libraries (see Table 5.11).
- To upload the *Orchestrator*' source to the *Orchestrator* machine.
- To generate a pair of Rivest, Shamir and Adleman (RSA) keys for enabling the connections to *Procesing Chain* node.

5.4.2.2 Implementation of the Processing Chain

The implementation of the *Processing Chain* was performed in a bash script. This script is remotely executed by the *Orchestrator* for processing the images. It was made in this manner because this is an implementation for testing the processing behaviour on cloud. The elastic service provided by the *BonFIRE* platform was not available, so as a first approach, a machine was fixed for testing the architecture.
Furthermore, the clustering service provided for *INRIA* did not also work, so the dinamic creation of processes can not be done because a only processing chain needs as minimum 6

GB of RAM memory. At the end, only a Chain Processing at time can be executed, so this implementation is very restricted in performance.

5.4.2.2.1 Processing Chain Workflow

The *Processing Chain* component works by following the next sequence of steps:

1. The *Orchestrator* component sends the image using SCP to the directory "tmp" in the *Processing Chain* machine.
2. The *Orchestrator* remotely executes the script "PP_script.sh" located in the *Processing Chain* machine.
3. The image processing start. The processors are sequentially executed.
4. When all the product processors have finished, the *Processing Chain* sends the orthorectified image to *Archive and Catalogue* using SCP.
5. The *Processing Chain* orders the *Archive and Catalogue* module to archive and catalogue the image received. In the first development, the *Processing Chain* returned the results to the *Orchestrator* and the orchestrator sent the images to the *Archive and Catalogue* module for cataloguing and archiving. But that implementation was inefficient because there were two sending files more (the first one is between the *Processing Chain* to the *Orchestrator* and the second one, between the *Orchestrator* and *Archive and Catalogue*. Finally, the solution was to accomplish only a sending. The *Processing Chain* after processing sends the results for archiving and cataloguing.

5.4.2.2.2 Processing Chain Interfaces

The *Processing Chain* module has interfaces with both the *Orchestrator* and the *Archive and Catalogue* modules.

5.4.2.2.2.1 Interfaces with the Orchestrator

The *Orchestrator* communicates with the *Processing Chain* for processing new incoming raw data. This communication is carried out using SCP for file sending and for remote commands, (SSH). This provides a manner of creating processes on-demand. If the *Processing Chain* machine was running in a cluster or under an Elasticity as a Service (EAAS), the resources would be dinamically requested.

5.4.2.2.2.2 Interfaces with the Archive and Catalogue

The *Archive and Catalogue* receives the images which the *Processing Chain* has processed. The transfer of the images is performed using SCP. Then, the *Processing Chain* remotely executes a Python script in the *Archive and Catalogue* machine for archiving and cataloguing the sent image.

5.4.2.2.3 Processing Chain Design

The *Processing Chain* is formed by the "PP_script.sh" bash script. Originally, this module will be located in an *EaaS*. The *BonFIRE* platform has not available this feature yet, so it was decided to push into a cluster provided by the *INRIA* testbed. The cluster platform was not also available, so it was necessary to create and fix a machine which plays the *Processing Chain* role with the restriction that there can only be one instance.

5.4.2.2.4 Processing Chain Implementation

The implementation of this module was done using a bash script. This script needs the IP address of the *Archive and Catalogue* module. In addition, it requires the SCP and SSH clients for its interfaces with the other modules. The image which is sent to the *Archive and Catalogue* when the process is finished, it is always the same because for the experiment is more important the processing on cloud than the resulting image.

In addition, a script was developed in order to obtain the workload of the *Orchestrator* machine. That file uses the "Python-psutil" library in order to obtain the CPU times such as user, idle, nice and iowait cycles. This file is used by the graphical user interface for plotting the workload of the *Orchestrator* at real time.

Finally, shared storage was used to store the images instead of the local storage. The shared storage is provided by the *IBBT* testbed. This storage is implemented using Network File System (NFS) protocol. Ideally, the NFS protocol using a network with a large bandwidth and links with 10 GB Ethernet is more efficient than using a local hard disk. However the NFS server is not conformed by disks located in the *BonFIRE* platform. The physical disks where the information is saved are in *IBBT* located in Ghent (Belgium). Thus, the read and write accesses from any *BonFIRE* testbed (INRIA and EPCC in this case) travel from *BonFIRE* to *IBBT* through the Internet. This involves high latencies, large waiting times and it implies that the CPU is idle large time waiting for I/O operations. These results are shown in Section 7.2.

The implementation of this shared storage was performed as follows:

- In the *BonFIRE* web interface, a shared storage was created.

- The folders structure for processing were done. This structure is composed by (the local structure has the same distribution):
  - Job orders folder: it contains the job orders necessary for processing all stages.
  - l0_input folder: it contains the neccesary inputs for L0 processor.
  - l0r_input folder: it contains the neccesary inputs for L0R processor.
  - l1a_input folder: it contains the neccesary inputs for L1A processor.
  - l1br_input folder: it contains the neccesary inputs for L1BR processor.
  - l1bc_input folder: it contains the neccesary inputs for L1BC processor.
  - l1cr_input folder: it contains the neccesary inputs for L1CR processor.
  - l1ct_input folder: it contains the neccesary inputs for L1CT processor.
- Processors outputs folder: contains the outputs and temporaly files created by the processors during their actions.

5.4.2.2.5 Processing Chain Execution

To execute the *Processing Chain* module the following dependencies are required:

- To know the IP address of the *Archive and Catalogue*.
- Ethernet interface for the network connection in the *BonFIRE WAN*.
- The geolocated image for sending to *Archive and Catalogue* module once the processing is finished.
- The Product Processors owned by *Elecnor Deimos* installed in the machine.

The script is located inside "source/bonfire/ProcessingChain" folder. The execution of the software is executed with the following command line:

> bash PPscript.sh «arg1» «arg2»

where *arg1* is the outfile name in which the *Archive and Catalogue* module saves the image and the *arg2* is the simulated scenario.

5.4.2.2.6 Implementation in BonFIRE

To implement the *Processing Chain* module the following steps were done:

- Reservation of a *Xlarge* machine in *EPCC BonFIRE* platform. This machine belongs the *BonFIRE WAN* so it can communicate with all the machines in that network.
- Installation of the product processors.
- Uploading the geolocated image.

- Uploading the *ProcessingChain* source to the *ProcessingChain* machine.
- Appending the public key from *Processing Chain* machine to the *" /.ssh/authorized_-host"* to allow the incoming connections.

5.4.2.3 Archive and Catalogue

The first implementation of the *Archive and Catalogue* component was performed for archiving and cataloguing the geolocated images which were processed in the simulation of the defined scenarios. It is based on the *GeoServer* software. The inputs are given by the *Processing Chain* module when it finishes the processing of an image. It sends the image using SCP and then, it remotely executes the Python script for archiving and cataloguing the image. Once the image is catalogued, the end user can access it with the *IP* address of the node using a web-browser. Then *GeoServer* is showed and the end user can navigate between the different scenarios and files generated in the execution of the experiment.

5.4.2.3.1 Archive and Catalogue Workflow

The *Archive and Catalogue* component works by following the next sequence of steps:

1. The *Processing Chain* ends its processing and sends the resulting image via SCP to the *Archive and Catalogue*
2. The Python script is remotely executed by the *Processing Chain* by using SSH.
3. The script obtains the image for archiving and copies it into the *GeoServer* data directory.
4. The image is catalogued by using the API provided by *GeoServer*. If the scenario was not created as a workspace, the workspace is created.
5. A store is created in order to house the image.
6. The image is catalogued in the workspace and stored in the created store.
7. The *GeoServer* module publishes the catalogued image using the CSW protocol. Any end user connectig the web interface can access the catalogued and published images of the scenarios.

5.4.2.3.2 Archive and Catalogue Interfaces

The *Archive and Catalogue* has interfaces with the *Processing Chain* and the *GeoServer* software.

5.4.2.3.2.1 Interfaces with the Processing Chain

When the *Processing Chain* module has finished the processing of an image, it sends the image to the *Archive and Catalogue* module via SCP protocol to proceed with its archive and catalogue. The *Archive and Catalogue* recives the image and stores it into the data directory of *GeoServer*. Then, the *Processing Chain* remotely orders to catalogue this image.

5.4.2.3.2.2 Interfaces with the GeoServer software

The *GeoServer* software provides a Python library namely *Gsconfig*. The official page of this library is `https://github.com/boundlessgeo/gsconfig`. Using this library, operations with layers, images, tilesets or storages can be done. The provided operations for cataloguing "tiff" images (the default format for geodata images) were used in this project.

5.4.2.3.2.3 Interfaces with CSW clients

The *GeoServer* software implements a CSW plugin to provide the end users a CSW interface instead of a web-browser for accesing the catalogue. This feature provides a new connectivity channel for data interchange between modules of others projects, business or end users.

5.4.2.3.3 Archive and Catalogue Design

The *Archive and Catalogue* is formed by the *GeoServer* software and a Python script for communicating with *GeoServer* in order to catalogue and store the processed images.

The script communicates with the *GeoServer* software for creating a workspace, for creating data stores and for cataloguing the images were sended by the *Processing Chain*.

incluir esquema script -> geoserver -> csw o http

5.4.2.3.4 Archive and Catalogue Implementation

The implementation of this module was done in Python 2.7. The python's libraries needed to implement the software are listed in Table 5.12.

The *Gsconfig* library was manually installed as follows:

- The library obtained from the following url: `https://github.com/boundlessgeo/gsconfig`.

- Firstly, the "Python-pip" package is installed.
- Then just execute "pip install gsconfig".

Then, the *GeoServer* software was required to be installed. For that purpose, the *Apache Tomcat* server was installed. Then, the "war" file which contains the *GeoServer* software was downloaded from the official webpage and it was installed into the *Tomcat* server. In Section 5.4.2.3.6 this process is explained in detail.

5.4.2.3.5 Archive and Catalogue Execution

To execute the *Archive and Catalogue* module the following dependencies are required:

- *Apache Tomcat*
- *Geoserver with the* CSW *plugin*
- *Gsconfig library installed*
- *E*thernet interface for the network connection in the *BonFIRE WAN*.
- *E*thernet interface for the network connection with a public IP address. In this project an IPv4 was used.

The developed source for archiving and cataloguing is located in "source/bonfire/geoserver". The execution of the *Archive and Catalogue* module namely "catalog_pp.py" is done as follows:

```
> python catalog_pp.py «name_file» «scenario» «nameStore»
```

where «name_file» is the absolute path of the image to the *Archive and Catalogue*, «scenario» is the scenario in which the image is catalogued and the «nameStore» is the name of the created store for housing the image.

5.4.2.3.6 Implementation in BonFIRE

For implementing the *Archive and Catalogue* module the following steps were done:

- Reservation of a *Medium* machine in *INRIA BonFIRE* platform. This machine belongs to the *BonFIRE WAN* so it can communicate with all the machines in that network. Moreover it has a public IP address in order to be accesible outside the *BonFIRE* network by end users.
- Installation of the Java version 6.
- Installation of the *Apache Tomcat* server and its customization in order to listen in port 80 and to reserve the necessary requirements for Java.

- Installation of the *GeoServer* and the *CSW* plugin in *Apache Tomcat*. In the CD-ROM attached the setup script for this step is included.
- Uploading of the geolocated image.
- Uploading of the *ProcessingChain* source to the *ProcessingChain* machine.
- Appending the public key from *Processing Chain* machine to the *" /.ssh/authorized_-host"* for allowing the incoming connections.

5.4.3 Implementation of the cloud architecture using ZeroC ICE

ZeroC ICE provides multiple services to create distributed architectures. By using its replication service, load balancing and location transparency the following architecture can be obtained. The components of the architecture implemented with ZeroC ICE and their interrelations are represented in Figure 5.25. The layers of the cloud architecture are the following:

- Layer 1: the ZeroC ICE distributed platform. It provides the runtime environment where distributed components can be deployed conforming a distributed architecture.
- Layer 2: the cloud architecture. It defines the logical nodes in which the servers (components of the system) are deployed.
  - *Orchestrator* node: it is the node in which the *Orchestrator* server is deployed.
  - *Broker node*: it is the node in which the *Broker* server is deployed.
  - *Processing Chain node*: it is the node in which the *Processing Chain* server is deployed.
  - *Archive and Catalogue* node: it is the node in which the *Archive and Catalogue* is deployed.
- Layer 3: the architecture servers (also namely component). It defines the basics features such as their relations and interfaces that the components require. In this architecture, the following servers were defined:
  - *Client/Ground Stations*: it is the client of the architecture but it is not included in it. This component is used by the experimenter to initialize the experiment. Moreover, in the client attached in the CD-ROM, it also says to the *Orchestrator* that new raw data is available. In this project the client implementation was done to check the funcionality of the architecture. The implementation of the client funcionality in the Ground Stations implemented in *Virtual Wall* may be future work.
  - *Broker* server: it acts as an intermediary between the client and the cloud architecture components.

- *Orchestrator* server: this component manages the ingestion and the processing the raw data and archiving and cataloguing the images obtained. It was explained in Section 5.4.1.1.
- *Processing Chain* server: this component proccesses the images downloaded by the *Orchestrator* module and it notifies the *Orchestrator* when it finishes. All *Processing Chains* servers are joined into a Replica Group namely *ProcessingChainReplica module*. It provides the replication service and the load balancing. When it receives a request for processing, it selects one of the *Processing Chain* servers which less workload has.
- *Archive and Catalogue* server: it archives and catalogues the images. It was explained in Section 5.4.1.3.

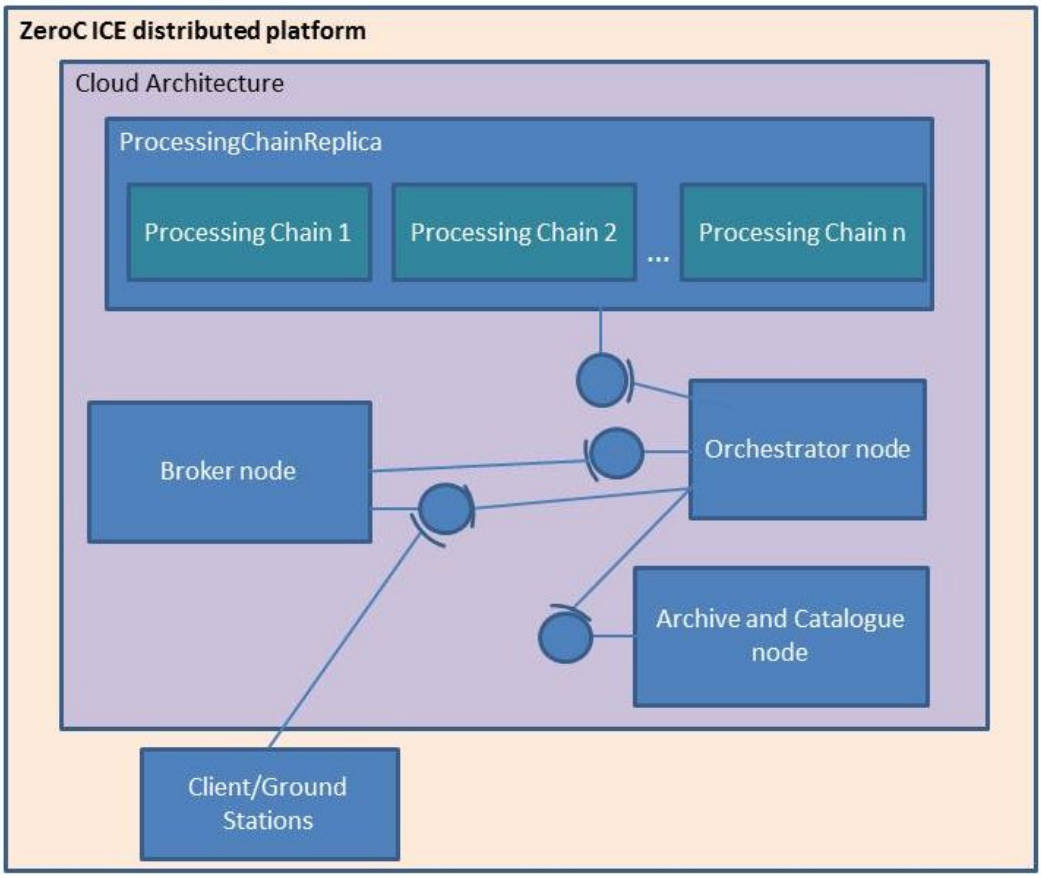


Figure 5.25: Cloud architecture using ZeroC ICE.

This architecture is independent of the platform when it is executed. There are logical nodes in which ICE deploys the servers. These nodes may match with physical nodes or may be several logical nodes in the same physical node. The ICE runtime manages the location of the servers and their execution.

Furthermore, the data store is shared between all the servers. This means that the components such as the *Orchestrator*, the *Archive and Catalogue* and the *Processing Chains* share the same logical memory space and they can write or read data at the same time. This implementation avoids the file transfers and it does the cloud infrastructure clearer, more dynamic and scalable.

In the following sections the ICE interfaces between these components are shown and the previous modules modules are explained. Their design and interfaces with other modules are described. Finally, the deployment of the software in each module is explained in detail.

5.4.3.1 ICE interfaces

The distributed application using ZeroC ICE was made using a slice file namely "Geocloud.ice"for specifying the interfaces between components. These interfaces allows the transparently communication between the components. The slice file is listed in Listing 5.10.

5.4.3.2 Implementation of the Orchestrator

The *Orchestrator* performs the same task that in the architecture based in SSH and SCP (see Section 5.4.2). It connects with the ground stations over FTP connections and then, the communications between the components of the cloud are done by using the ICE interfaces of each component. ICE uses TCP protocol by default.

5.4.3.2.1 Orchestrator Workflow

The *Orchestrator* component works by following the next sequence of steps:

1. The ICE core application is initialized and the ICE communicator is created.
2. The *Orchestrator Object Adapter* is instantiated and the *Orchestrator* servant is added to it.
3. In the *Orchestrator* initialization, the *Listener* process is spawned.
4. The *Listener* process creates the *LoadData* object for loading the ground stations IP addresses and the FTP credentials.
5. The *LoadData object* gets all the information about the *Ground Stations Simulators* and localizes them.
6. The *Listener* pools the *Ground Stations* and when there are a downloadable raw data, the Orchestrator's function *donwloadedImage* is called passing the absolute path of the image as parameter.
7. When the *downloadedImage* is called, the *Orchestrator* obtains the data.

```
2  module geocloud {
3      exception AlreadyExists { string key; };
4      exception NoSuchKey { string key; };
5      exception CreationScenarioException{};
6      exception StartScenarioException{};
7      exception StopScenarioException{};
8      exception DeleteScenarioException{};
9      exception ArchiveNotAvailableException{};
10     exception OrchestratorNotAvailableException{};
11     exception ProcessingException{};
12     exception CataloguingException{};

14     interface Orchestrator{
15         void initScenario(int scen) throws StartScenarioException,
               ArchiveNotAvailableException;
16         void downloadedImage(string path);//the ground station calls
               this operation passing the path
17         void imageProcessed(string path);
18         void imageCatalogued(string path);
19         void stopScenario() throws StopScenarioException;
20     };

22     interface Broker{
23         void startScenario(int scen) throws
               OrchestratorNotAvailableException, StartScenarioException;
24         void appendLog(string newLog);
25         void stopScenario(int scen);
26         void setOrchestrator( Orchestrator * orch);
27         string getLastLogs();
28     };


31  interface Processor{
32         //int init( Broker * log);
33         void processImage(string path) throws ProcessingException;
34         void shutdown();
35         void setOrchestrator(Orchestrator * orch);
36     };



40     interface ArchiveAndCatalogue{
41         void createScenario(string scenario) throws
               CreationScenarioException;
42         void catalogue(string path,string storage,string scenario)
               throws CataloguingException;
43         void deleteScenario(int scenario) throws
               DeleteScenarioException;
44     };
45 };
```

Listing 5.10: Slice of the ICE application.

8. The *Orchestrator* moves the raw data to a shared storage.

9. Then, the *Orchestrator* makes different *Job Orders* for the Processing Chains. The

Job Order contains all the required information by the *Product Processing Chains* to proceed with the image processing.

10. The *Orchestrator* gets the *ProcessorChainReplica* proxy (this replica group contains all the Processing Chains in the cloud) and calls the *processImage* operation of the proxy.
11. When the selected *Processing Chain* finishes, it calls the *processedImage* operation of the *Orchestrator*.
12. The *Orchestrator* obtains the *Archive and Catalogue* proxy and sends the absolute path of the image for archiving and cataloguing.

5.4.3.2.2 Orchestrator Interfaces

The *Orchestrator* has interfaces with the Ground Stations implemented in *Virtual Wall*. Moreover this component communicates with the *Product Processing Chains* through the Replica Group and with the *Archive and Catalogue*.

5.4.3.2.2.1 Interfaces with the Ground Stations implemented in Virtuall Wall

The *Ground Stations* are deployed in some *Virtual Wall* nodes. In those, the impairments and features of the network are simulated. Essentially, the *Orchestrator* is pooling those *Ground Stations* over FTP connections to know when new raw data is available. So, this Ground Stations are FTP servers in which the *Orchestrator* can get the raw data recorded by the constellation of satellites. As the ICE implementation could not be deployed in the *BonFIRE* platform, the checking and testing of funcionalities were locally done. Thus, a client playing as a Ground Station was developed in order to check and validate the architecture.

5.4.3.2.2.2 Interfaces with the Processing Chains

The interfaces which the *Processing Chains* provides are listed in Listing 5.10. The operations that the *Orchestrator* uses are *processImage* and *setOrchestrator* operations.

5.4.3.2.2.3 Interfaces with the Archive and Catalogue

The interfaces which the *Archive and Catalogue* provides are listed in Listing 5.10. The operation that the *Orchestrator* uses is the *catalogue* operation.

5.4.3.2.3 Orchestrator Design

The *Orchestrator* module implements the *Orchestrator* interface which contains the following operations:

- *void initScenario(int scen)*: it initializes the scenario in the module.
- *void downloadedImage(string path)*: it indicates to the *Orchestrator* that an image is available to be processed in the specified path.
- *void imageProcessed(string path)*: the Processing Chains call this function to indicate to the *Orchestrator* that the processing of the image has finished, so the *Orchestrator* can archive and catalogue it.
- *void imageCatalogued(string path)*: the catalogue module uses this function to notify the *Orchestrator* that the image located in "path" was archived and catalogued.
- *void stopScenario(int scen)*: it stops the scenario in the *Orchestrator*. Furthermore, it stops all the existing processing chains working.

The requests are served when they come to the *Orchestrator* module. If other request comes and the processing chains are busy, they are automatically queued by the ICE execution core. Thus, it is not necessary to maintain a queuing algorithm in the *Orchestrator*. However it is necessary to have some data structures, in this case dictionaries to temporaly store the images which are in different stages.

Furthermore, when the *Orchestrator* is initialized the *Listener* process is accomplished. It listens the FTP connections with the ground stations to detect and download new raw data. This process is always pooling the ground stations until the *Orchestrator* execution ends.

5.4.3.2.4 Orchestrator Implementation

The implementation of this module was done in Python 2.7 using ZeroC ICE 3.4. The python's libraries needed to implement the software are listed in Table 5.13

5.4.3.2.5 OrchestratorExecution

To execute the *Orchestrator* module the following dependencies are required:

- *ZeroC ICE 3.4* installed.
- *E*thernet interface for connecting with the *IceGrid* locator (see Section 5.4.3.7).
- *C*onfiguration file for the node.

In Section 5.4.3.7 the automatic deployment and execution of this module is explained.

Python Library	Function
Time	For managing the time
Pdb	Used for debugging the software
Gsconfig	Library used to communicates with *GeoServer* software
OS	Library which provides operative system interactions
Sys	System library

Table 5.12: ICE Archive and Catalogue Python Libraries.

Python Library	Function
Threading	System library for creating ,syncronizing and managing threads
OS	Library which provides operative system interactions
Collections	Library that contains the data structure "deque" used for queueing the request which can not be processed.
Time	For managing the time
Pdb	Used for debugging the software
FtpLib	Library used to obtain and manage the Ground Stations' FTP connections
Ice	ICE general purpose library
IceGrid	ICE library to use the IceGrid service

Table 5.13: ICE *Orchestrator* Python Libraries.

5.4.3.3 Implementation of the Processing Chain

The *Processing Chain* component processes the raw data to obtain a calibrated, geolocated and orthorectified image. It executes all the product processor stages and notifies the *Orchestrator* server The background filesystem, where the cloud based in ICE are implemented and executed, is a shared storage, so all the necessary files are located in that shared memory space explained in Section 5.4.2.2.4. Moreover, all the instantiated *Processing Chains* form part of the *ProcessingChainReplica* in order to provide load balancing and dynamic replication services.

5.4.3.3.1 Processing Chain Workflow

The *Processing Chain* component works by following the next sequence of steps:

1. The *setOrchestrator* operation is invoked for setting up the *Orchestrator* in the *Processing Chain*.
2. When the *Orchestrator* invokes the *processImage* fuction, the *Processing Chain* starts to transform the raw data located in path.
3. When all the stages have been performed, the *Processing Chain* calls the *processedImage* of the *Orchestrator*.
4. The *Processing Chain* is ready for processing other image.
5. If the *shutdown* function is called by the *Orchestrator*, the *Processing Chain* is turned off.

5.4.3.3.2 Processing Chain Interfaces

The *Processing Chain* component has interfaces with the *Orchestrator*.

5.4.3.3.2.1 Interfaces with the Orchestrator

The *Processing Chain* server receives the order to process an image. When the processing is finished, the *Processing Chain* notifies the end of the processing to the *Orchestrator* by calling the *imageProcessed* function.

5.4.3.3.3 Processing Chain Design

The *Processing Chain* module implements the *Processor* interface, which contains the following operations:

- *void processImage(string path)*: it is invoked by the *Orchestrator* in order to process the image located in path. Thus, all the product Processing Chains are sequentially executed and the *processedImage* of the *Orchestrator* is called.
- *void shutdown()*: this funcion turns off the proccesor.
- *setOrchestrator()*: it sets up the *Orchestrator* proxy to be used for the *Processing Chain*.

5.4.3.3.3.1 ProcessingChainReplica

This component involves all the *Processing Chains* in the ICE application. It provides two services: dynamic replication and load balancing. The first of service is used for creating new on demand *Processing Chains*. The second one service servers the incoming petitions searching the less overloaded *Processing Chain* and returning its proxy for carrying out the request.

5.4.3.3.4 Processing Chain Implementation

The implementation of this module was done in Python 2.7 using ZeroC ICE 3.4. The python's libraries needed to implement the software are listed in Table 5.14.

Python Library	Function
Threading	System library to create, syncronize and manage threads
OS	Library which provides operative system interactions
Pdb	Used for debugging the software
Ice	ICE general purpose library
IceGrid	ICE library to use the IceGrid service

Table 5.14: ICE Processor Python Libraries.

5.4.3.3.5 Processing Chain Execution

To execute the *Processing Chain* module the following dependencies are required:

- *ZeroC ICE 3.4* installed.

- *E*thernet interface for connecting with *IceGrid* locator.
- *C*onfiguration file for the node. In Section 5.4.3.7 the automatic deployment of the *Processing Chain* and its execution are explained.

5.4.3.4 Implementation of the Archive and Catalogue

The implementation of the *Archive and Catalogue* component using ZeroC ICE was performed to archive and catalogue the processed images. It uses the *GeoServer* software and the Python library (Gsconfig) to catalogue and archive the images.

5.4.3.4.1 Archive and Catalogue Workflow

The *Archive and Catalogue* component works by following the next sequence of steps:

1. The ICE core application is initialized and the ICE communicator is created.
2. The *Archive and Catalogue Object Adapter* is instantiated and the *A&C* servant is added to it.
3. When the initialization of a scenario is called, a workspace is created.
4. Then, when a image is requested for cataloguing, the module communicates with the *GeoServer* software by the library to catalogue.
5. The images are catalogued in the same workspace.
6. When the scenario ends, *deleteScenario* is invoked and the workspace is deleted.

5.4.3.4.2 Archive and Catalogue Interfaces

The *Archive and Catalogue* module has interfaces with the *Orchestrator* and the *GeoServer* software.

5.4.3.4.2.1 Interfaces with the Orchestrator

The *Archive and Catalogue* module notifies a images was catalogue to the *Orchestrator* by using the *imageCatalogued* call when a image is catalogued.

5.4.3.4.2.2 Interfaces with the GeoServer software

The *GeoServer* software provides a Python library named *Gsconfig*. The official page of this library is `https://github.com/boundlessgeo/gsconfig`. Using this library, op-

erations with layers, images, tilesets or storages can be done. The provided operations for cataloguing "tiff" images (the default format for geodata images) were used.

5.4.3.4.3 Archive and Catalogue Design

The *Archive and Catalogue* module implements the *Archive and Catalogue* interface, which contains the following operations:

- *void createScenario(string scenario)*: it is invoked by the *Orchestrator* and creates the workspace into the *GeoServer* platform for storing the images.
- *void catalogue(string path, string storage,string scenario)*: It catalogues the image located in path, it creates the storage where the image is stored and it creates the workspace named scenario.
- *void deleteScenario(string scenario)*: it deletes the workspace created to carry out the experiment.

5.4.3.4.4 Archive and Catalogue Implementation

The implementation of this module was done in Python 2.7 using ZeroC ICE 3.4. The python's libraries needed to implement the software are listed in Table 5.15.

Python Library	Function
Time	To manage the time
Pdb	Used for debugging the software
Gsconfig	Library used to communicates with *GeoServer* software
Ice	ICE general purpose library
IceGrid	ICE library to use the IceGrid service

Table 5.15: ICE *Archive and Catalogue* Python Libraries.

The *Gsconfig* library was manually installed as follows:

- The library is obtained from the following url: `https://github.com/boundlessgeo/gsconfig`.
- Firstly, the "Python-pip" package is installed.
- Then just execute "pip install gsconfig".

The *GeoServer* software was required to be installed. For that purpose, the *Apache Tomcat* server was installed. Then, the "war" file which contains the *GeoServer* software was downloaded from the official webpage and it was installed into the *Tomcat* server.

5.4.3.4.5 Archive and Catalogue Execution

To execute the *Orchestrator* module, the following dependencies are required:

- *ZeroC ICE 3.4* installed.
- *E*thernet interface for connecting with the *IceGrid* locator (see Section 5.4.3.7).
- *Apache Tomcat*.
- *Geoserver with the* CSW *plugin*.
- *Gsconfig library installed*.
- *C*onfiguration file for the node.

In Section 5.4.3.7 the automatic deployment and execution of the *Archive and Catalogue* is explained.

5.4.3.5 Implementation of the Broker

The *Broker* component performs the initialization and the stop of the experiments. In addition, it collects the which are accesible by the experimenters.

5.4.3.5.1 Broker Workflow

The *Broker* component works by following sequence of steps:

1. The ICE core application is initialized and the ICE communicator is created.
2. The *Broker Object Adapter* is instantiated and the *Broker* servant is added to it.
3. The *Broker* creates a data structure for storing the last log received.
4. The *Broker* receives a log and it stores it in the data structure.
5. When the *Broker* receives a request applying for the logs, the *Broker* returns them and it cleans the data structure.
6. When the experimenter invokes the *startScenario* of the *Broker*, it obtains the *Orchestrator* proxy and it calls the remote method *initScenario* of the *Orchestrator* interface.
7. When the experimenter invokes the *stopScenario* of the *Broker*, it obtains the *Orchestrator* proxy and it calls the remote method *stopScenario* of the *Orchestrator* interface.

5.4.3.5.2 Broker Interfaces

The *Broker* component has interfaces with the *Orchestrator*, with all the cloud components and with the clients used by experimenters.

5.4.3.5.2.1 Interfaces with the Orchestrator

The *Broker* module communicates with the *Orchestrator* by using the *initScenario* and *stopScenario* calls. The first one is used to initialize the experiment in the *Orchestrator* and the second one is used to stop the experiment in the *Orchestrator*.

5.4.3.5.2.2 Interfaces with all the cloud components

The cloud components communicate with the *Broker* by using the *appendLog* operation. When this operation is invoked, the log sent as parameter is added in the log data structure.

5.4.3.5.2.3 Interfaces with the clients

The clients invokes the *getLastLogs* procedure in order to obtains all the logs collected.

5.4.3.5.3 Broker Design

The *Broker* module implements the *Broker* interface which contains the following operations:

- *void startScenario(int scen)*: it initializes the data structure to store logs and initializes the *Orchestrator* by calling *initScenario*.
- *void stopScenario(int scen)*: it calls the *stopScenario* operation of the *Orchestrator*. The data structure is not cleaned in this method because all the available logs may not be read by the experimenter client.
- *void appendLog(string log)*: this function is called by all components of the cloud and it appends the log as parameter in the data structure of the *Broker*.
- *string getLastLogs()*: this function returns to the applicant client the last available logs in the log data structure. When finished, the log structure is cleaned for appending more log strings.

- *void setOrchestrator(Orchestrator *orch)*: it sets up the *Orchestrator* proxy passed as parameter to be used by the *Broker*.

5.4.3.5.4 Broker Implementation

The implementation of the *Broker* was done in Python 2.7 using ZeroC ICE 3.4. The python's libraries needed to implement the software are listed in Table 5.16

Python Library	Function
Time	For managing the time
Pdb	Used for debugging the software
Ice	ICE general purpose library
IceGrid	ICE library to use the IceGrid service

Table 5.16: ICE Broker Python Libraries.

5.4.3.5.5 Broker Execution

To execute the *Broker* module the following dependencies are required:

- *ZeroC ICE 3.4* installed.
- *E*thernet interface for connecting with *IceGrid* locator.
- *C*onfiguration file for the node. In Section 5.4.3.7 the automatic deployment and its execution of the *Broker* are explained.

5.4.3.6 Implementation of the client

The client developed for testing and checking the funcionality of the cloud architecture were developed in the ICE runtime environment. The functionalities that the client performs are the following:

- Obtaining the *Orchestrator proxy*.
- Sending raw data to *Orchestrator* by calling *downloadedImage*.
- Calling *getLastLogs* function of the *Broker*, periodically.

As future work, the ground stations may be implemented by using the ICE middleware. It would provide more transparency and easiness to deploy the experiments. Furthermore, it would facilitate the integration with the *Orchestrator* by using the *Orchestrator* interface to notify that there are new raw data availble.

The implementation of this module was done in Python 2.7 using ZeroC ICE 3.4. The python's libraries needed to implement the software are listed in Table 5.17

Python Library	Function
Pdb	Used for debugging the software
Ice	ICE general purpose library
IceGrid	ICE library to use the IceGrid service

Table 5.17: ICE Client Python Libraries.

5.4.3.6.1 Client Execution

To execute the client attached in the CD-ROM, the following dependencies are required:

- *ZeroC ICE 3.4* installed.
- *E*thernet interface for connecting with *IceGrid* locator (see Section 5.4.3.7.
- *C*onfiguration file for the locator.

The execution of the client, which is located in "source/ice/src/client.py", is done as follows:

> python client.py –Ice.Config=cfg/locator.cfg

5.4.3.7 Deployment of the ICE architecture

The deployment of the ICE architecture was done using the *IceGrid* and *IcePatch* services. *IceGrid* provides location transparency by using well-known objects and server replication and load balancing services by creating replica groups. *IcePatch* is an efficient file patching service which is easy to configure and use. It provides the automatically source distribution to all nodes.

The deployment was made creating a distributed application (located in "source/ice/app" folder) which is composed by nodes, servers running in the nodes and object adapters created by the servers. The nodes are the logical implementation of a physical node where any kind of server can be executed. The nodes created for this deployment were the following:

- *Orchestrator node*: it contains the *Orchestrator* server.
- *Archive and Catalogue node*: it contains the *Archive and Catalogue* server. The *GeoServer* software is also included.
- *Broker node*: it contains the *Broker* server which acts as an intermediate between the cloud architecture and the client or ground stations.

- *Processing Chain node*: it contains a *Processing Chain* server.

Some scripts for initializing the nodes were developed. This scripts are the following:

- Start.sh: it creates the folder structure and launches the "icegridnode" daemons for each node.
- Stop.sh: it stop the nodes.
- Clean.sh: it cleans the temporaly files and directories created during the execution.

Moreover, the source must be located in "/tmp/ice" because to create an application for deploying in several machines, the most usual is to provide a way for an easy deployment.

For the deployment was effective, the "icegridnode" services must be running in each node. There are several configuration files for the nodes. These files are located in "source/ice/cfg/" directory. The "node$.cfg" files ,where "$" is a number, are the configuration files for the nodes. One of them, in this case the "node1.cfg" contains the configuration for the locator service. The locator service provides to ICE applications the location of all well-known objects registered in the ICE runtime environment. The execution of each node is done as follows:

> icegridnode –nochdir –daemon –Ice.Config=cfg/nodo$.cfg

where $ is the number of node to be executed.

Finally the installation of the application and the source distribution is done by the following command:

> icegridadmin –Ice.Config=cfg/locator.cfg -e "application add app/GeoCloudApp.xml"

where the locator file contains the address of the node where it is located and the *GeoCloudApp.xml* is the application description where the deployment, replication policies and the different services involve in the distributed application are defined. Executing the *icegridadmin* command and installing the application, automatically are deployed and started up the servers in their respective node.

5.5 Profilling Tool in PlanetLab

In this section, the motivation for the use of *PlanetLab* is explained together with the design of the real system. Then, the platform and tools used are described with their roles in the experiment. In addition, the network and experiment design are broadly discussed. The execution of the experiment is also presented. Finally, conclusions of this implementation are included.

5.5.1 Definitions

- *Effective bandwidth (Mbps)* is the actual bandwidth at which the data can be transmitted on a link. The nominal bandwidth cannot be reached due to network congestion,

the distance between nodes, delays, etc. Effective bandwidth is higher when nodes are closer, the congestion is scarce and the delays in the transmission are not long.

- *Bandwidth of the network(Mbps)*, which is the nominal "width" of the channel used, if the bandwidth increases, more data can simultaneously be sent, reducing the necessary time to transfer a packet of data. It is usually confused with the signal velocity, which affects the time the data takes to travel to the receiver (latency) but bandwidth cannot reduce this time.
- *Loss rate* is the fraction of data lost in the communication with respect to all the data sent. It is a value between 0 and 1. It can also be provided in percentage.
- *Latency (ms)* is the time it takes a signal to travel from its source, trough the communication channel, until it reaches the receiver. It is related with the distance between the nodes, the network congestion and the propagation velocity (a fraction of the light speed) among other parameters.

5.5.2 Platform description

The experiment presented was completely carried out in *PlanetLab* , although the results obtained will be used to implement a realistic model of the networks in the communications between the simulators and the cloud system of the GEO-Cloud experiment implemented in *Virtual Wall* and *BonFIRE* respectively.

PlanetLab is a global research network that supports the development of new network services [Eur14]. This testbed currently consists of 1188 nodes at 582 sites, which allows researchers to develop new technologies for distributed storage, network mapping, peer-to-peer systems, distributed hash tables and query processing. Currently it is split into two platforms: *PlanetLab Europe* that contains the European nodes and *PlanetLab Central* which contains the nodes located outside Europe. For the experiment we use nodes from both of them.

5.5.3 Tools description

To implement and execute the experiment we use the following tools:

- *NEPI* [INR14]: it is a Python-based language library used to design and easily run network experiments on network evaluation platforms (e.g. *PlanetLab*, *OMF*, wireless testbeds and network simulators among others). It facilitates the definition of the experiment workflow, the automatic deployment of the experiment, resource control and result collection; and has the functionalities of automatic provisioning of resources and automatic deployment of the experiment. In the experiment *NEPI* is used to provision the nodes and execute the whole experiment.
- *Iperf* [Ipe14]: it is a tool used to measure the maximum *TCP* bandwidth, allowing the tuning of various parameters and *UDP* characteristics. *Iperf* reports *bandwidth*,

delay,*jitter* and *datagram loss*. This software allows any host to play the client and server roles. In the experiment, it was used to obtain the bandwidth with a step of one second when executed in *TCP* mode and the loss rate when executed in *UDP* mode. The nodes in Layer 1 and 2 were configured as clients and the cloud node as server.

- *Ping* [PCVB13]: It is software used to test if a host on an Internet Protocol is reachable. It measures the RTT for messages sent from the originating host to a destination host. In the experiment, it was used to measure the latency delivery of a package over the Internet between the nodes in Layer 1 and 2 and the central node.

5.5.4 PlanetLab Experiment

The objective of the *GEO-Cloud* experiment is to simulate as realistically as possible the behaviour of a complete Earth Observation system [GPB+14]. With this aim, the communication links in the real system have to be modelled to connect the simulators implemented in *Virtual Wall* and *BonFIRE* with the values obtained from the experiment in *PlanetLab*. The experiment then consists of communicating 12 real nodes representing the ground stations (the nearest *PlanetLab* node to the real ground station was selected) and the end users distributed around the world (we selected 31 nodes from different 31 countries) with a node representing the cloud (located in *INRIA*) to measure the real impairments of the networks and to implement a realistic model of the communications. The impairments to be measured and used to model the network are the effective bandwidth, the latency and the loss rate. An equivalence scheme is shown in Figure 5.26 with the correlation between the parameters obtained from the experiment and the inputs to model the links between *Virtual Wall* and *BonFIRE* . There are two networks in the system:

1. The dedicated network connecting the ground stations and the cloud: it is represented by the bandwidth, the latency and the loss rate.
   (a) The bandwidth will be computed as a control variable.
   (b) The latency will be extracted from the latency measured in the *PlanetLab* experiment.
   (c) The loss rate will be extracted from the loss rate measured in the PlanetLab experiment.
2. The Internet network connecting the end users and the cloud: it is represented by the bandwidth, the latency, the loss rate and the background traffic.
   (a) The bandwidth will be computed as a control variable.
   (b) The latency will be extracted from the latency measured in the *PlanetLab* experiment.
   (c) The loss rate will be extracted from the loss rate measured in the *PlanetLab* experiment.

(d) The background traffic is affected by the following parameters:

(e) Throughput: the effective bandwidth measured with the *PlanetLab* experiment will be computed as the throughput parameter in *Virtual Wall* .

3. Packet size: 1500 bytes.

4. Protocol: the protocol used is TCP.

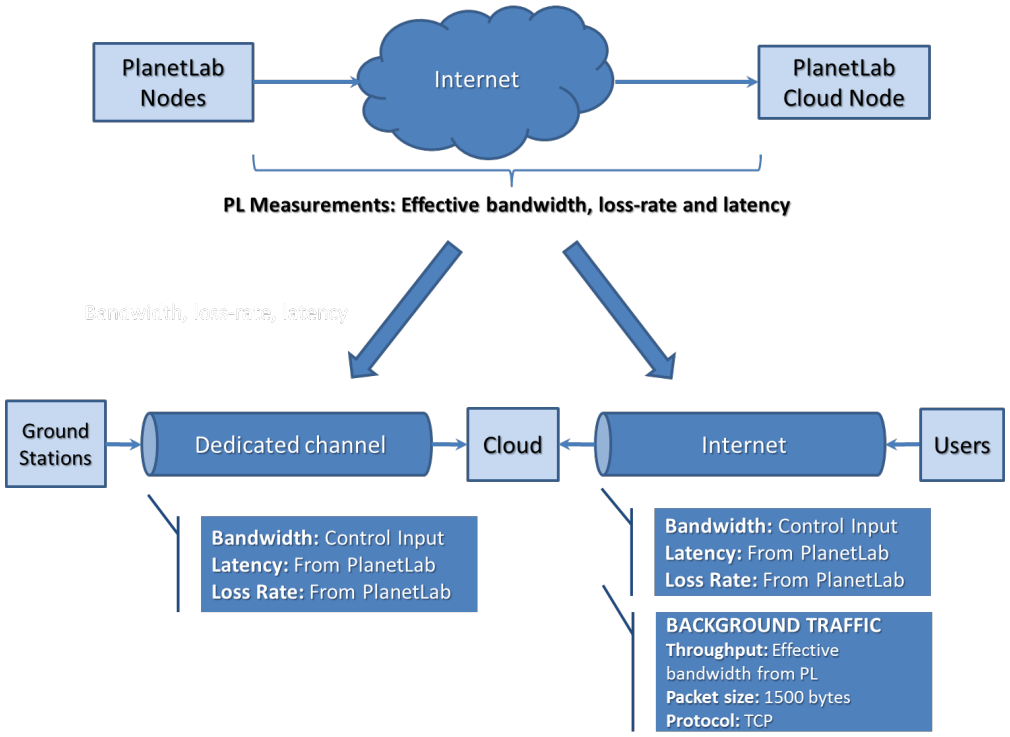


Figure 5.26: PlanetLab and modelled links equivalences

5.5.4.1 System Modeling

The real system was modeled into three main components: i) a network of ground stations acquiring imagery data from a constellation of optical satellites, ii) a cloud infrastructure that ingests the data from the ground stations, processes it, stores it and distributes it through web services and iii) end users around the world accessing to the web services offered. The system can be divided into two layers:

1. *Layer 1* is constituted by 12 ground stations connecting with a cloud infrastructure. The ground stations and their location are depicted in Table 5.18. Their locations and footprints are depicted in Figure 5.3. The footprints represent the area in which the satellites can establish the communication with the ground stations.

2. *Layer 2* is constituted by the end users accessing the web services implemented in cloud. These users are distributed around the world and can be governments, emergency services, media and individuals among others.

From the previous layers two networks can be identified: the network between the ground stations and the cloud and the network between the end users and the cloud. The system and the interconnections between components are depicted in Figure 5.27. The connections between the ground stations and end users with the cloud are represented as arrows with different line types to represent that every connection can have different characteristics and impairments. All the connections are TCP.

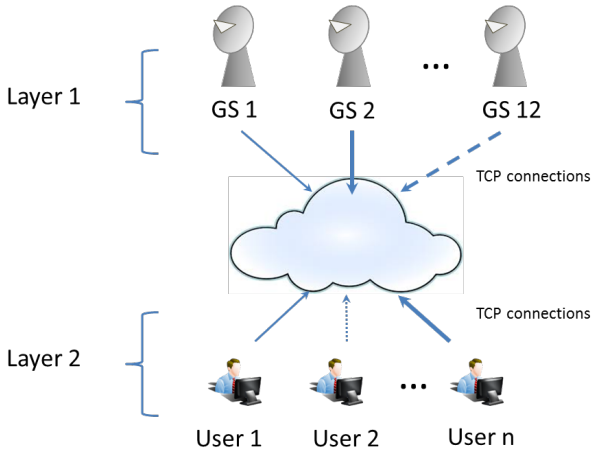


Figure 5.27: System description

The network was defined in function of the following representative impairments: *effective bandwidth, latency and loss rate*. Thus, every link were represented in function of the previous impairments: effective bandwidth, latency, loss rate. This system was implemented in *Virtual Wall* and *BonFIRE* as depicted in Figure 5.28. The experiment in *PlanetLab* was used to update the network parameters connecting *Virtual Wall* and *BonFIRE* .

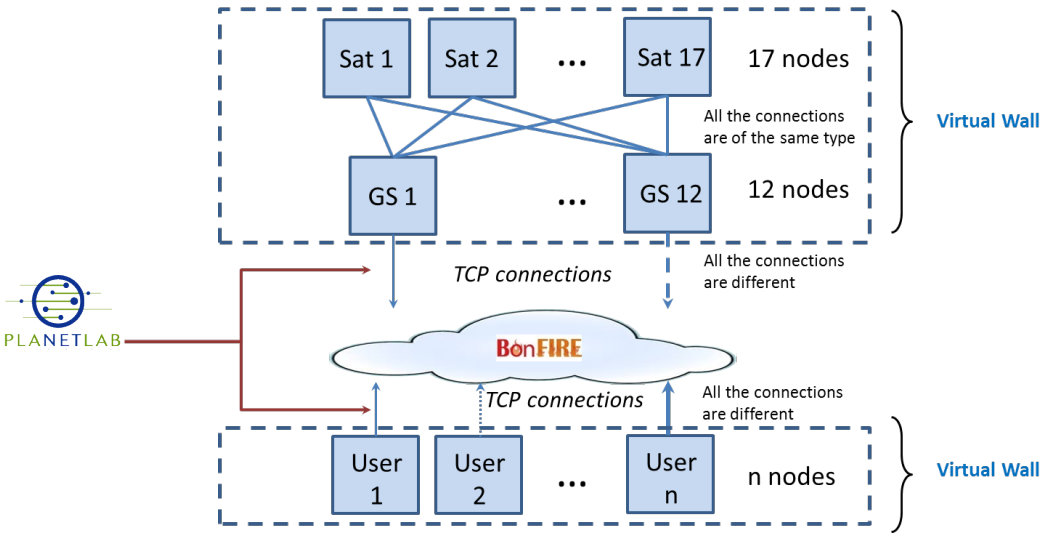


Figure 5.28: Scheme of the system implemented in Geo-Cloud

5.5.4.2 Network Design

The model of the ground stations network, the cloud and the end users accessing the web services provided was simplified to a set of interconnected nodes. The network was divided into two layers connected by a central node representing the cloud servers for similarity with the real system:

- **Layer 1:** it represents the connections between 12 nodes representing the ground stations and a central node representing the cloud servers. In *PlanetLab Europe* and *PlanetLab Central*, the nearest nodes to the real location of the ground stations were selected. For the central node, a node in *INRIA* was chosen, since the *BonFIRE* cloud has servers in the same location. This layer then represents the transfer of geodata acquired by the constellation of satellites from the ground stations in which the data is downloaded to the cloud. The network topology implemented is peer-to-peer, i.e. each node representing the ground stations is directly connected with the central node. In Table B.1 the *PlanetLab* selected nodes for layer 1 and for the cloud central node are shown. The nodes are numbered in ascending order in function of the distance to the central node, i.e. the closest node is the number 0 and the furthest the 37.
- **Layer 2:** it represents the connection between the central node representing the cloud servers and the end users. 31 different nodes were selected in *PlanetLab Europe* and *PlanetLab* Central in 31 different countries around the world. This allows us to have a representative sample of global users accessing the web service s. In this case, the network topology is also peer-to-peer. In Table B.2 the nodes selected for layer 2 are listed. We tried to increase the number of nodes in different countries, but during the execution of the experiment we did not find available PlanetLab nodes in the following countries: Austria, Cyprus, Denmark, Egypt, Ecuador, Iceland, India, Jordan, Mexico, Pakistan, Puerto Rico, Romania, Slovenia, Sri Lanka, Tunisia, Turkey, Venezuela, Uruguay and Taiwan.

Figure 5.29 shows a scheme representing the network created in *PlanetLab* . The connections between the nodes are *TCP*.

5.5.4.3 Experiment Design and Execution

The experiment was designed to measure the impairments of the network. Those impairments are required parameters in *Virtual Wall* to deploy a topology network in such a testbed. Then, the latency, loss-rate and effective bandwidth were measured. The deployment of the experiment was done with *NEPI* , in which *Iperf* and *Ping* were implemented to measure the impairments. The experiment consists of establishing communications between any node in layer 1 or layer 2 with the central node and measuring the previously described impairments. 21600 trials were performed during 6 hours of the experiment execution in steps of one second for each pair of nodes, i.e. a node from layer 1 or 2 and the central node.

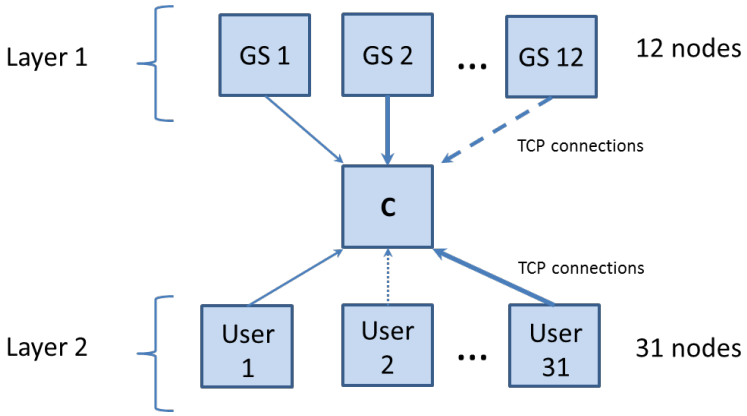


Figure 5.29: PlanetLab Network Scheme

The software developed to measure the impairments is constituted of 6 scripts:

- Script to measure the effective bandwidth in the ground stations nodes: "bandwidthGS.py".
- Script to measure the effective bandwidth in the end users nodes: "bandwidthEndUser.py"
- Script to measure the latency in the ground stations nodes: "latencyGS.py".
- Script to measure the latency in the end users nodes: "latencyEndUser.py"
- Script to measure the loss rate in the ground stations nodes: "lossRateGS.py"
- Script to measure the loss rate in the end users nodes: "lossRateEndUser.py"

The pair of scripts that measure the same impairment are differentiated one from each other in the provisioning of the nodes. Those nodes representing the ground stations are manually selected, while the end users nodes are automatically provisioned by *NEPI* by indicating the country name as parameter. This parameter allows *NEPI* to select an available node in that country.

The previous six scripts were individually executed in a local host and they started their workflow.

5.5.4.3.1 The bandwidthGS.py script

The *bandwidthGS.py* script measures the effective bandwidth in the ground stations nodes. When it is executed it carries out the next tasks:

1. Provisioning of the nodes that were manually selected.
2. Creation of the commands to be uploaded:
    - In the cloud node:

timeout %dm iperf -s -f m -i 1 -p %d
Timeout is a command that executes a program during a specified time %dm in minutes. For this experiment dm was chosen to be 6 hours.

- s indicates that *Iperf* is executed in server mode
- f m indicates the format to report the received data. In this case in Mb.
- i 1 Periodic reports every 1 second
- p %d indicates the port to listen. In this case 20004.

- In the ground station nodes:

iperf -i 1 -f m -c %s -t %d -p %d -y c > node%d.out

- i 1 Periodic reports every 1 second.
- f m indicates the format to report the received data. In this case in Mb.
- c %s indicates the server to establish the communication with.
- t indicates the data transmission time. In this case 3600 seconds.
- p %d indicates the port to listen. In this case 20004.
- y c> node%d.out indicates that the report format is CSV. The output file is node%d.out, where %d indicates the number of the node tested.

By default *Iperf* is executed in TCP mode.

3. Uploads the commands to the nodes
4. Executes the command in cloud
5. Executes the command in the rest of nodes
6. During the execution of the commands the data is collected
7. Finishes the execution of the commands
8. The data collected is retrieved
9. The resources are released.

The flow diagram of the effective bandwidth measurements in the ground station nodes is depicted in Figure 5.30b.

5.5.4.3.2 The bandwidthEndUser.py script

The *bandwidthEndUser.py* script measures the effective bandwidth in the end users nodes. When it is executed it carries out the next tasks:

1. Automatic provisioning of the nodes.
2. Tasks 2 to 9 of the *bandwidthGS.py* script.

The flow diagram of the effective bandwidth measurements in the end users nodes is depicted in Figure 5.30a.

5.5.4.3.3 The lossRateGS.py script

The *lossRateGS.py* script measures the loss rate in the ground stations nodes. When it is executed it carries out the next tasks:

1. Provisioning of the nodes that were manually selected.
2. Creation of the commands to be uploaded:
   - In the cloud node:

     timeout %dm iperf -s -f m -i 1 -p %d -u

     Timeout is a command that executes a program during a specified time %dm in minutes. For this experiment dm was chosen to be 6 hours.
     - s indicates that *Iperf* is executed in server mode
     - f m indicates the format to report the received data. In this case in Mb.
     - i 1 Periodic reports every 1 second
     - p %d indicates the port to listen. In this case 20004.
     - u indicates that the *Iperf* software is executed in *UDP* mode.
   - In the ground station nodes:

     iperf -i 1 -f m -c %s -t %d -p %d -y c > node%d.out

     - i 1 Periodic reports every 1 second.
     - f m indicates the format to report the received data. In this case in *Mb*.
     - c %s indicates the server to establish the communication with.
     - t indicates the data transmission time. In this case 3600 seconds.
     - p %d indicates the port to listen. In this case 20004.
     - y c> node%d.out indicates that the report format is CSV. The output file is node%d.out, where %d indicates the number of the node tested.
     - u indicates that the *Iperf* software is executed in *UDP* mode.

   By default *Iperf* is executed in TCP mode.
3. Uploads the commands to the nodes
4. Executes the command in cloud
5. Executes the command in the rest of nodes
6. During the execution of the commands the data is collected
7. Finishes the execution of the commands

8. The data collected is retrieved
9. The resources are released.

The flow diagram of the loss rate measurements in the ground station nodes is depicted in Figure 5.30b.

5.5.4.3.4 The lossRateEndUser.py script

The *lossRateEndUser.py* script measures the loss rate in the end users nodes. When it is executed it carries out the next tasks:

1. Automatic provisioning of the nodes.
2. Tasks 2 to 9 of the *lossRateGS.py* script.

The flow diagram of the loss-rate measurements in the end users nodes is depicted in Figure 5.30a.

5.5.4.3.5 The latencyGS.py script

The *latencyGS.py* script measures the latency in the ground stations nodes. When it is executed it carries out the next tasks:

1. Provisioning of the nodes that were manually selected.
2. Creation of the commands to be uploaded:

   ping %s -w %d

   - %s indicates the host to do ping
   - w %d indicates the time of the ping execution.
3. Uploads the commands to the nodes
4. Executes the command in cloud
5. Executes the command in the rest of nodes
6. During the execution of the commands the data is collected
7. Finishes the execution of the commands
8. The data collected is retrieved
9. The resources are released.

The flow diagram of the latency measurements in the ground station nodes is depicted in Figure 5.30d.

5.5.4.3.6 The latencyEndUser.py script

The *latencyEndUser.py* script measures the latency in the end users nodes. When it is executed it carries out the next tasks:

1. Automatic provisioning of the nodes.
2. Tasks 2 to 9 of the *latencyGS.py* script.

The flow diagram of the latency measurements in the end users nodes is depicted in Figure 5.30c.

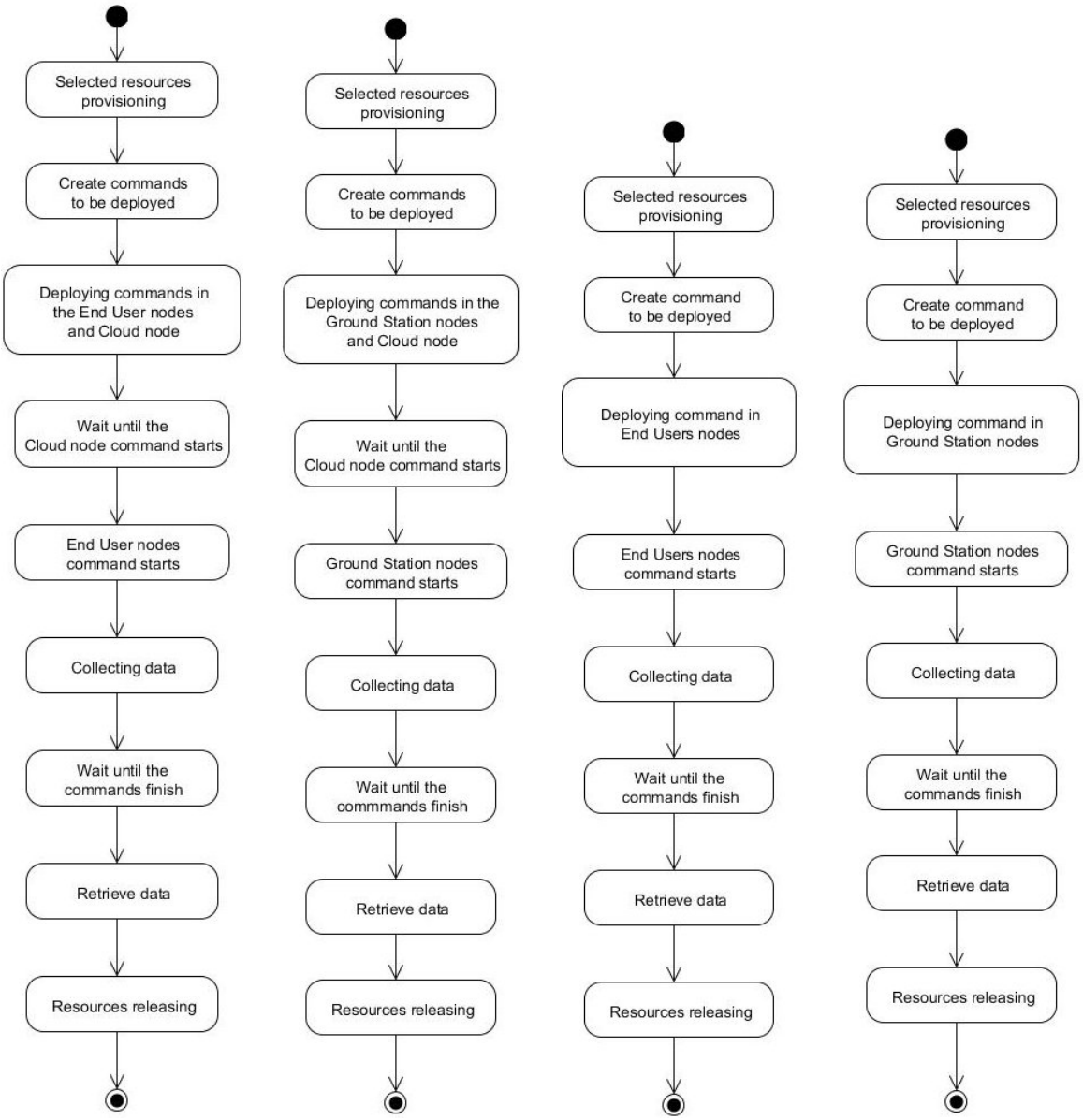
5.6 GEO-Cloud Graphical User Interface

In this section, the GUI developed for managing the execution of the defined scenarios in the GEO-Cloud experiment is exposed. Firstly, the architecture is explained. Secondly, the components of the graphical user interface are described. Finally, the interface is shown to identify the components.

5.6.1 Architecture

The graphical user interface of the GEO-Cloud experiment is composed by the following classes:

- *Experiment Controller*: it manages the *Space System Simulator* turning it on or shutting it down by using SSH connections and obtains the workload of the *Orchestrator* and the *Processing Chain* machine.
- *SSH Connection*: it creates an *SSH* connection with a *Virtual Wall* machine.
- *SSH Order*: it remotely executes an *SSH* order in an *SSH Connection* object.
- *getLoad*: it obtains the workload of the *Orchestrator* and the *Processing Chain* machines.
- *JFedParser*: it parsers the *Rspec* obtained from the *JFed* software once the *Virtual Wall* deployment was made to obtain the hostnames of the *Virtual Wall* nodes.
- *UI Controller*: it manages the graphical widgets and the *Experiment Controller* interactions.
- *Video Widget*: it shows a video demonstration of the satellite constellation acquiring images at the same time that the satellites record the Earth's surface in the simulator implemented in *Virtual Wall*.
- *About Widget*: it shows the *About* window where the acknowledments are shown.
- *Log Widget*: it prints the logs which receives from satellites and ground stations.



(a) Flow diagram of the *bandwidthEndUser.py* and *lossRateEndUser.py* scripts.

(b) Flow diagram of the *bandwidthGS.py* and *lossRateGS.py* scripts.

(c) Flow diagram of the *latencyGS.py* script.

(d) Flow diagram of the *latencyEndUser.py* script.

Figure 5.30: Flow diagrams of the scripts developed for the *PlanetLab* experiment.

- *Tab Widget*: it contains two graphics to plot the workload of the *Orchestrator* and the *Processing Chain* machines.

The Figure 5.31 shows the class diagram in which the relations between the components are depicted.

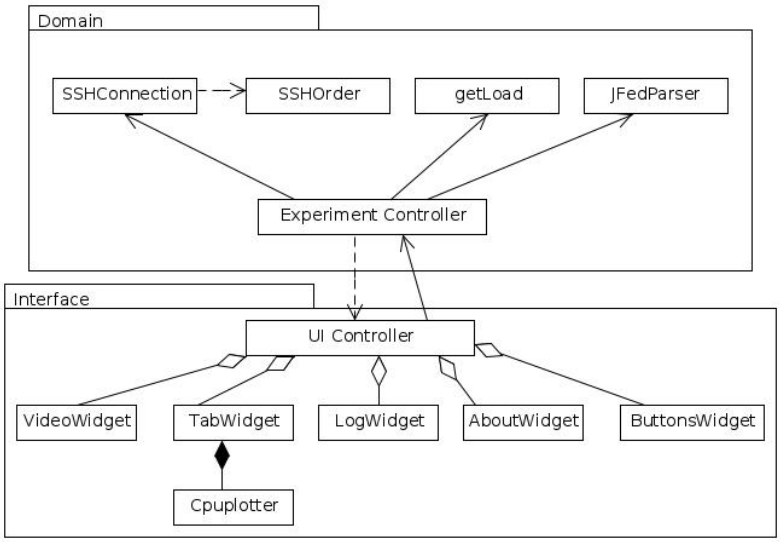


Figure 5.31: Class diagram of the graphical user interface.

5.6.1.1 Experiment Controller

The functions of the *Experiment Controller* are the following:

- To obtain the hostnames of the *Virtual Wall* nodes included in the *Rspec* file located in "source/gui/resources/jfed.out". It creates a *JFedParser* object to do this. These hostnames are used to create *SSH Connections* objects to send orders to the *Virtual Wall* nodes.
- To create the *SSH Connections* of all the nodes deployed in *Virtual Wall* .
- To create the *SSH Commands* to manage the experimentation in *Virtual Wall* .
- To obtain the workload of both the *Orchestrator* and the *Processing Chain* machines. Then, these measures are sent to the *UI Controller* for plotting.
- To communicate the experiment logs from *Virtual Wall* to the *UI Controller*.

5.6.1.1.1 Experiment Controller Workflow

The workflow of the *Experiment Controller* is the following:

1. The file which contains the *Rspec* is parsed.

2. Using the hostnames obtained, the *Experiment Controller* creates an *SSH Connection* per hostname obtained.
3. When the experimenter selects a scenario and pushes the start button, the *Experiment Controller* creates *SSH Commands* indicating to start the selected scenario. Moreover, it starts the reproduction of the video of the selected scenario.
4. When the experimenter selects an scenario and pushes the stop button, the *Experiment Controller* creates *SSH Commands* indicating to *Virtual Wall* nodes that the selected scenario must to be stopped.
5. The log from the *SSH Commands* performed is collected and it is sent to the *UI Controller*.
6. Two *getLoad* objects are created to connect with the *Orchestrator* and the *Processing Chain* machines to obtain in real time their workload by using *SSH Connections*. The workloads are passed to the *UI Controller* for plotting.

5.6.1.2 SSH Connection

This class connects the *Virtuall Wall Bastion* machine to the *Virtual Wall* nodes. This connection is performed by using the public key described in Section A.1.

This class has the *executeCommand* method in order to remotely carry out the commands given by the *Experiment Controller*.

5.6.1.2.1 SSH Connection Workflow

The workflow of the *SSH Connection* is the following:

1. The object is created by passing as arguments: the hostname of the *Virtual Wall* machine and the IP address of the *Virtual Wall Bastion* machine.
2. By using *Paramiko* library, the connection is performed.
3. When a command is executed, the log is returned and sent to the *Experiment Controller* class.

5.6.1.3 SSH Order

This class creates an *SSH order* to be sent through an *SSH Connection* object to a *Virtual Wall* node.

5.6.1.3.1 SSH Connection Workflow

The workflow of the *SSH Connection* is the following:

1. The object is created by passing the order that has to be remotely executed as argument.

5.6.1.4 getLoad

This class obtains the workload of both the *Orchestrator* and the *Processing Chain* machines by using an *SSH Command* through their *SSH Connections* objects.

5.6.1.4.1 getLoad Workflow

The workflow of the *getLoad* is the following:

1. The object is created by passing the IP address of the remote machine as argument.
2. An *SSH Connection* is created.
3. Periodically, *getLoad* sends a *SSH Order* to get the workload through the *SSH Connection*.

5.6.1.5 JFedParser

This class parses the *Rspec* of the *JFed* to obtain the hostnames of the *Virtual Wall* nodes.

5.6.1.5.1 JFedParser Workflow

The workflow of the *JFedParser* is the following:

1. The object is created by passing as argument the path of the *Rspec* file.
2. By using an XML Python parser (Minidom library), the hostnames of all nodes deployed in *Virtual Wall* are obtained.
3. When the operation *getHostnames* is called, the hostnames are returned.

5.6.1.6 UI Controller

The *UI Controller* is based on the *Controller* design pattern and it has the following functionalities:

- To manage the experiment status through the graphical widget interactions and to send them to the *Experiment Controller*.
- To show the experiment status such as the workload, logs and video from the *Experiment Controller*.

5.6.1.6.1 UI Controller Workflow

The workflow of the *UI Controller* is the following:

1. It commands the *Experiment Controller* to execute the experiment when the user selects an scenario and the start button is clicked. When the user starts the execution, the video of the selected scenario is started by calling to the *Video Widget* with the selected scenario.
2. It commands to the *Experiment Controller* to stop the execution of the experiment when the stop button is clicked.
3. It receives the logs from the *Experiment Controller* and sends them to the *Log Widget*.
4. It receives the workloads from the *Experiment Controller* and sends them to the *Tab Widget* to be plotted.
5. When the *About Button* is clicked, the *About Widget* is created and shown in the screen.

5.6.1.7 Video Widget

This class reproduces the video of the selected scenario.

5.6.1.7.1 Video Widget Workflow

The workflow of the *Video Widget* class is the following:

1. When a scenario is started, the *Video Widget* is set up to reproduce the video of the selected scenario.
2. When the video finishes, it stops the reproduction.

5.6.2 About Widget

This class shows a dialog with the acknowledments.

5.6.2.0.2 About Widget Workflow

The workflow of the *About Widget* class is the following:

1. When this object is created, it loads the resource images and the ackowledments and shows them.

5.6.3 Log Widget

This class shows the logs received from the *UI Controller*.

5.6.3.0.3 Log Widget Workflow

The workflow of the *Log Widget* class is the following:

1. When the *UI Controller* calls the *appendLog* operation of the *Log Widget*, the logs sent as parameter are shown in the GUI.

5.6.4 Tab Widget

This class has two *CPUPlotter* objects. The first one plots the *Orchestrator* workload and the second one, the *Processing Chain* workload. Each machine of them are located in a tab in which the experimenter can see the workload of the machines. The *CPUPlotter* class was developed by using the *QWT* libraries of the *QT* framework. It creates a linear plot which changes with the input of new data. It was distributed with *Psymon* application (developed by Dimitris Diamantis) under the GPLv3, reused in this project.

5.6.4.0.4 Tab Widget Workflow

The workflow of the *Log Widget* class is the following:

1. The *UI Controller* sends the workload from the *Experiment Controller* to the *Tab Widget* object.
2. Depending of the source of the data (*Orchestrator* or *Processing Chain* machine) the data is plotted in one or the other.

5.6.5 Implementation of the Graphical User Interface

The implementation of this module was done in Python 2.7. The python's libraries needed to implement the software listed in Table 5.19.

5.6.6 Execution of the Graphical User Interface

To execute the GUI the following dependencies are required:

- The libraries listed in Table 5.19.
- The last *Rspec* specification of the *Virtual Wall* deployment in *JFed*. This file is located in "source/gui/resources/jfed.out".
- *E*thernet interface for the network connection in the *BonFIRE WAN*.
- *E*thernet interface for connecting with the *Virtual Wall* nodes through the *Virtual Wall Bastion*.

The execution of the GUI is done as follows (the experimenter must be located in "source/gui" folder):

```
> python main.py
```

5.6.7 Components of the Graphical User Interface

The Graphical User Interface is composed by four well diferenciated elements:

- *Video zone*: it is the zone where the video of each scenario is reproduced.
- *Log zone*: it is the zone where the logs are shown during the experiment.
- *Workload zone*: it is the zone where the workloads of the *Orchestrator* and the *Processing Chain* machines are shown.
- *User interaction zone*: it is the zone where the experimenter manages the experiment.

The Figure 5.32 shows the different components of the graphical user interface.

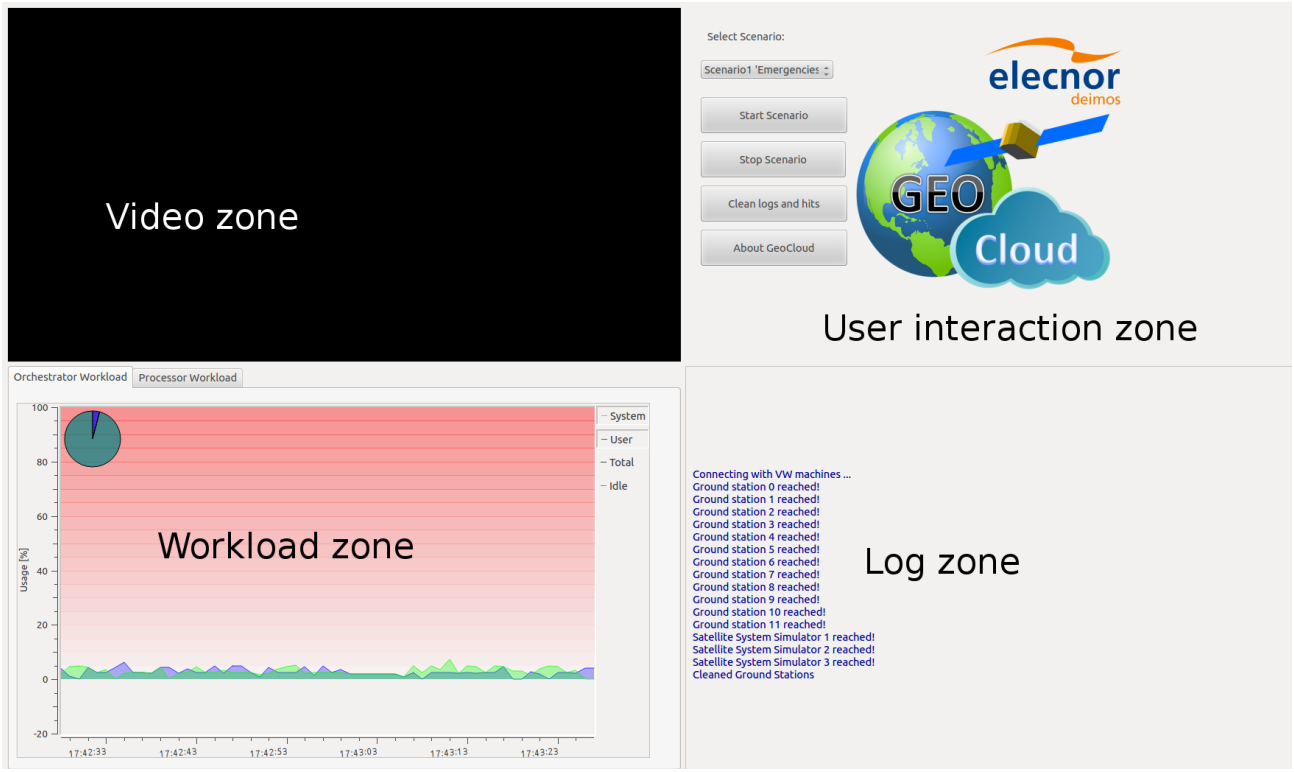


Figure 5.32: Components of the GUI

Ground Station	Country of GS location
Irkutsk	Russia
Puertollano	Spain
Svalbard	Norway
Troll	Antarctic
Chetumal	Mexico
Córdoba	Argentina
Dubai	United Arab Emirates
Kourou	French Guiana
Krugersdorp	South Africa
Malaysia	Malaysia
Prince Albert	Canada

Table 5.18: Ground Station Location

Python Library	Function
Threading	System library for creating, syncronizing and managing threads
OS	Library which provides operative system interactions
Time	Libraty to manage the time
Paramiko	Library to create and manage SSH connections
Phonon	Library to manage multimedia files and widgets
Pdb	Used for debugging the software
PyQT	Library used to build the GUI
Psutil	Library used to obtain the workload of the CPU
XML DOM	Library to parse XML files

Table 5.19: GUI Python Libraries

Chapter 6

Evolution and Costs

THE different phases performed in the development of this project are described, specifying the milestones of each one and their temporal cost and complexity.

6.1 Project Evolution

Before the start of the development phase (February 2nd), a training stage was executed with the different *Fed4FIRE* testbeds. The training also included the familarization with the space and the specific technologies of *Elecnor Deimos*. This stage had a duration of 1 month. Then the requirement analysis was started. Technological and functional terms were studied. Once the requirements were specified, the iterative development started. Next these phases are described in detail.

6.1.1 Training stage

For a month, the *BonFIRE*, *PlanetLab* and *Virtual Wall* testbeds were studied. The manner of create experiments and automatize them also was investigate. In this stage there were lots of troubles related with the used technologies of these testbeds and the testbed maintainers solved them. The tools provided by the *Fed4FIRE* were studied and some experiment examples were achieved. Once the basics of the experimentation were learnt, the preliminary requirements analysis of the *GEO-Cloud* experiment started.

6.1.2 Preliminary requirements analysis

The initial requirements analyses which the entire system should satisfy were obtained after a week from the end of the training stage. Firstly, some basics requirements were fixed:

- It must be based in open source and open standards.
- The language used for the implementation must be multiplatform, dynamic and multipurpose.
- The tools used for the development should be the open source or the *Fed4FIRE* tools.
- The platforms to check the software should be the *Fed4FIRE* platforms.
- The implemented system must be as realistic as possible.

Then a document containing the requirements of the whole system and each of component was accomplished. In this document the components of the *Space System Simulator*, the cloud architecture and the experiment in *PlanetLab* were stated. The requirements of each component of the *Space System Simulator* such as the *Satellite System Simulator* and the *Ground Station System Simulator* were obtained. To obtain the IP addresses was implemented in the system. The cloud architecture components were individually analysed, together with the communications between them and their interactions. The network requirements, the topologies and the features of the machines required were studied.

The *PlanetLab* experiment consisted of analysing which impairments were involved in the network communications. They were studied and investigated to know the influence over the transmission speed, loss rate data and latency in the Internet network.

Then the design of the scenarios and their executions were defined in order to check all the functionalities and to evaluate the performance expected in the experiment. Moreover the GUI was not taken into account but at the moment the experiments started, it was decided to design and to implement the graphical interface.

6.1.3 General design

The GEO-Cloud project was designed in a modular and extensible frame which governed the development in a transparent, open and scalable manner. It is composed by many self-contained components, platforms and interfaces. Thus if it were necessary to introduce or to modify any functionality, the effort to do this will be minimal.

Furthermore, it was required to automate the execution of the experiments. For that purpose, the GUI was developed in order to any user without acknowledges about the implementation, could to use and understand it.

In addition, some software engineering patterns were applied such as *Proxy*, *Facade*, *Controller* and *Singleton*. These patterns facilitate the understanding and maintenance of the source code.

6.1.4 Iterations of the development

In this section the iterations done to develop GEO-Cloud are enumerated and explained describing the achievements, the complexity of some parts and the most relevant decisions considered.

6.1.4.1 Iteration 1

Firstly the folder structure in which the source and the documentation are located, were created. The scale of the project is quite considerable so it was strictly necessary to establish a main directory for each testbed. Furthermore, a several accounts were created; the first one, in the *BonFIRE* official page, the second one in the *iMinds Emulab* official page and the

last one in *PlanetLab* testbed. In these profiles, the public RSA key generated was uploaded. For the design of the entire cloud architecture, a top-down model was selected. Thus, the Ground Segment on premises of *Elecnor Deimos* was studied and the main components for obtaining geolocated images of the Earth surface were identified. These cloud architecture components were also designed by studying their relationships and the data flows between them. The design of the interfaces and relationships between all the cloud components were analysed. The basic architecture of the Orchestrator component were designed, drafted and some Universal Modelling Language (UML) diagrams were pictured.

Regarding the *Space System Simulator* , the data files from Satellite Tool Kit (STK) software were obtained. Then, a data analysis was carried out in order to design and build the data base and its associated script to fill it.

The main objective of this iteration can be summarized as the preparation of the development environment, the design of the cloud system and the obtaintion of data from the satellite constellation in order to build a customized and realistic simulator.

6.1.4.2 Iteration 2

The implementation of the cloud architecture components was started. First, the Orchestrator component was developed to run in a local machine. The class *Processing Chain Controller* was designed using a *Singleton* pattern [GGV+13]. Then, the Archive and Catalogue module was also locally implemented. *Geo-Server* and *Geo-Network* were studied in order to obtain which of these was more convenient to be implemented in the system. *Apache Tomcat* and *Geo-Server* were installed and the CSW plugin of *Geo-Server* was also installed and tested. The product processors of *Elecnor Deimos* were studied and installed in the development computer. The Processing Chain component was implemented and tested. It was executed via command line.

To implement the *Space System Simulator*, some Python libraries for scheduling and planning actions were studied. Then, the development of the *Satellite System Simulator* was carried out by obtaining the satellite data from the database and by scheduling functions depending on the behaviour of the satellite in each scenario.

The development of the *Ground Station System Simulator* was done by conceiving the *Ground Stations Simulators* as simple servers processing the received packets. The *Satellite System Simulator* sent a raw data in a constant bit rate. However, the required bit-rate was $160\ Mbps$. Thus the network over the Internet could not guarantee its transmission rate. The solution was to implement send tiny packets with different meanings encoded in order to transmit them in the required time. Whith this approach, the *Ground Stations Simulators* takes into account the number of packets received. The number of the received images by the ground stations depended on the number and type of these packets (see Section 5.2.5.1.1.1). Several test cases were done in order to validate this approach.

At the end of the iteration 2, the *Space System Simulator* was finished, tested and the cloud architecture was locally implemented. Its components were also individually tested.

6.1.4.3 Iteration 3

The main objective of this iteration consisted of designing and implementing the *PlanetLab* experiment. For that, the Federation tools provided by *Fed4FIRE* project were investigated. As a result, *NEPI* was selected to do the profiling in *PlanetLab*. In the development of this experiment, the following stages were performed:

- Searching and selecting nodes around the world for simulating the end-users and ground stations.
- Testing the connectivity of all the selected nodes and creating some scripts for testing them. This process was complex because most of the nodes appeared in the *PlanetLab* webpage as available and working but executing the script, some of them were unreachable. Thus, this trouble delayed the experiment because it was necessary to check node by node just to select the available ones.
- Analysing of the tools to measure the impairments (loss-rate, effective bandwidth and delay).
- Finally, the implementation and execution of the developed scripts were accomplished.

Once the results were obtained, some scripts in *Python-matplotlib* were developed. These scripts read the collected data and graphically represents it.

6.1.4.4 Iteration 4

The *Space System Simulator* was not implemented yet in the *Virtual Wall* testbed. To deploy the nodes, *JFed* was used. There were many problems with that because the nodes connectivity was correct for IPv6 but not for IPv4. Once the nodes deployment were successfully deployed and reachable via SSH, the *Space System Simulator* was installed in them.

The architecture in *BonFIRE* was also implemented and integrated. The machines which harboured the cloud components were prepared. The FTP server was configured in the Orchestrator, the *GeoServer* software and *Apache* were installed in the *Archive and Catalogue* and the product processors were laid into the *Processing Chain* machine.

Once the cloud architecture was prepared, the software was uploaded. Then, the scenario 1 was selected for testing the system and it was executed. Some adjustments were done and some bugs corrected. In the beginning, the executions were manually made. However there were some task and interactions with machines that required to develop a user interface to automatize all the processes.

Finally, the GUI was performed and the execution of the experiments were perfectly automated. Furthermore, the *European Commission* invited us to present this project and carry

out a demonstration of the system working in real time.

At the end of this iteration, the *Space System Simulator* was fully implemented and deployed. The cloud architecture was built and all its components interconnected.

6.1.4.5 Iteration 5

In this iteration the main issues were the experiment executions and the collection of some results.

The scenario 1 and 2 were executed to obtain preliminary results. These results are shown in Section 7.2. Next the implementation of a shared storage in order to avoid the transfer of data. This shared storage is based in NFS. This protocol enables sharing files between some compute resources and it maintains the consistency of these files. The experiment was also executed with this latest enhancement. Preliminary results without shared storage were obtained. This comparative is shown in Section 7.2.

6.1.4.6 Iteration 6

The last step of this project was to implement all the components of the cloud system using a distributed middleware: ZeroC ICE. First, the slice file which contains the definition of the distributed interfaces were carried out. Then, the implementation of each individual component were performed by implementing the operations which were defined in the slice file. During the development of this architecture, many manuals tests were performed. Once the implementation was finished and the test cases successfuly executed, IceGrid was used to automatically deploy the system in all the distributed nodes in a local machine. The nodes implementation and deployment using IceGrid was tested. As future work, to install and execute the cloud arquitecture based on ICE middleware may be a line of research. The implementation of this architecture in the *BonFIRE* cloud was not carryied out because it was not in the scope of this project.

6.2 Resources and costs

This section lists and describes the resources used, both temporal and economic. Furthermore, the statistics of the repository used to control the source and documentation versions are presented.

6.2.1 Economic cost

The training time of this project comprises from 3rd of February to 10th of March (24 days). The approximated daily dedicated time is 8 hours and the considered price for that period of time was 15 €/hour. The approximated dedicated time for the implementation of the system was 8 hours daily during 105 days i.e. 840 hours in that period. It was considered an average price of 15 €/hour a programmer (reference prices acquired from

`https://freelance.infojobs.net/freelancers`). The Table 6.1 shows the breakdown of resources used during the development.

Resource	Amount	Cost
Training salary	1	2880 €
Programmer salary	1	14400 €
Intel Core i5 3450 3.1 GHz 8GB RAM	1	420 €
European Commision Review	1	730 €
FOSS4G Conference	1	820 €
FIDC Conference	1	870 €
RAQRS Conference	1	475 €
TOTAL		20595 €

Table 6.1: Economical breakdown for the GEO-Cloud project.

6.2.2 Repository statistics

The source version control tool used in the development of this project was *Git* supported by the *Bitbucket* repository service. In Figure 6.1 the evolution of the source code is shown. Furthermore, the number of commits performed during the development of the project and the documentation are represented in Figure 6.2.

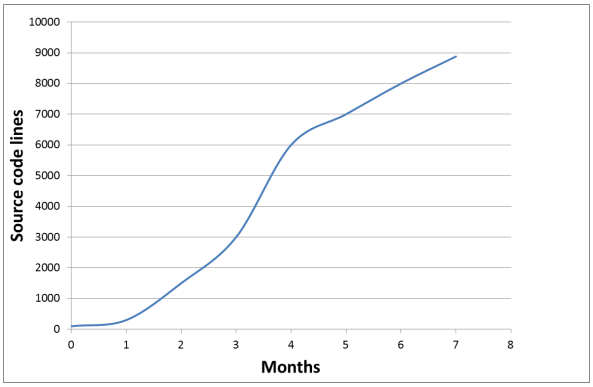


Figure 6.1: Source code evolution.

The *Cloc* tool was used for counting the source lines of code of all the programming languages in which GEO-Cloud was developed. This tool counts all lines of the code differentiating between code, blank lines and comment lines properly. The Table 6.2 shows the execution results of *Cloc* in the source folder.

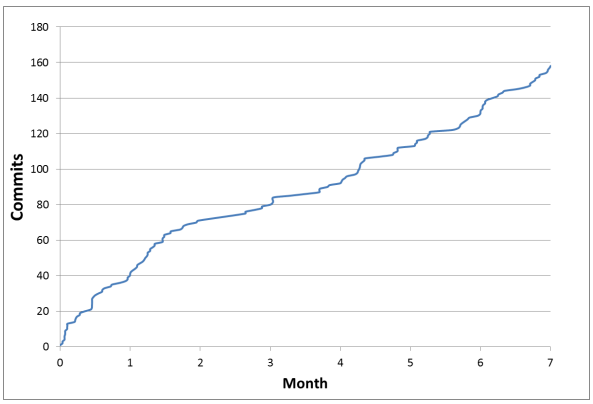


Figure 6.2: Evolution of the number of commits.

Languaje	Files	Blank spaces	Comments	Source lines of code
Python	91	2008	1675	7047
XML	9	1	12	765
Bourne Shell	17	75	139	230
Make	2	20	19	40
Total	119	2104	1845	8082

Table 6.2: Number of source lines of code of project.

Chapter 7

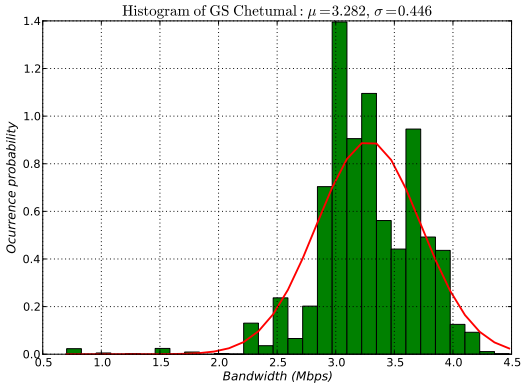
Results

THE obtained results of the implementation, execution and validation of the GEO-Cloud experiment are explained. The *PlanetLab* experiment for measuring the network impairments (bandwidth, loss-rate and RTT) are shown and plotted. These plots represent how the impairments directly depend on the distance between both source and destination elements. Then the experiment carried to demonstrate the correct implementation of the Geo-Cloud is shown. Then, the evaluation of the architecture using ICE are described. Finally, the evaluation of the system is explained.

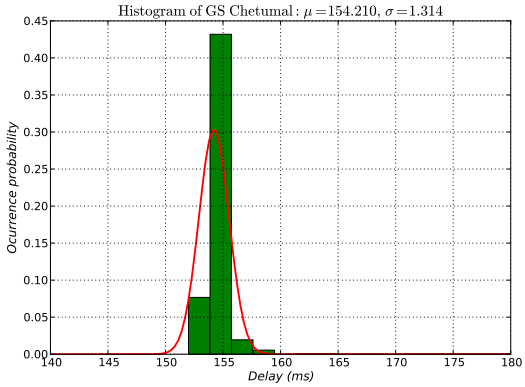
7.1 PlanetLab Experiment Results

During the execution of the *PlanetLab* experiment (see Section 5.5) 21600 communications were established between every node representing ground stations and users and the central node representing the cloud. The bandwidth, latency and loss rate were measured. In Figure 7.1a the 21600 samples acquired in the communication between the node 22 and the central node during 6 hours of continuous execution are represented in a normalized histogram. The data accurately fits to a gaussian distribution with mean $3.28\ Mbps$ and standard deviation $0.446\ Mbps$. In Figure 7.1b a normalized histogram of the measured latency is represented. It was fitted with a gaussian distribution with mean $154.210\ ms$ and standard deviation $1.314\ ms$. The loss rate between this node and the central node was obtained to be 0.0096% [GPB+14].

The previous procedure was followed for the rest of the nodes. In Figure 7.1 the mean and the standard deviation for each node are depicted respectively. It can be observed that in general, the bandwidth decreases when the distance between nodes increases. However the node 5 in Madrid presents a higher dispersion in the bandwidth value with respect to the rest of nodes. It has a bandwidth of $15.2\ Mbps$. The measurements were fitted with different functions by using the least squares optimization method. Exponential, polynomial, logarithm and hyperbolic functions were used to fit the samples. For each fitted function the R^2 coefficient of determination, which varies between 0 and 1, and indicates how well the statistical distribution is fitted. The highest the value of the R^2, the better the fitting. The following results were obtained: for exponential function: $Bandwidth = 4.655e^{-10^{-4}x}$,$R^2 = 0.5454$;



(a) Bandwidth of the node representing Chetumal ground station



(b) Latency of the node representing Chetumal ground station

for polynomial function $Bandwidth = -3 \cdot 10^{-4}x + 4.9444$, $R^2 = 0.3628$ and for logarithm function $Bandwidth = -1.519Ln(x) + 15.325$, $R^2 = 0.4539$, where x is the distance between any node and the central node in km. The bandwidth was obtained in $Mbps$. However, the hyperbolic function was the one that best fitted the distribution:

$$Bandwidth = 184.91x^{-0.547};\ R^2 = 0.582 \tag{7.1}$$

Figure 7.2 shows the mean and standard deviation of the latency measured in all the nodes connecting the central node in France. In this case the fitting used was a linear function. It accurately fitted the data distribution. The equation that approximated the data is the following:

$$Latency = 0.0228x + 17.88;\ R^2 = 0.8927 \tag{7.2}$$

The latency is obtained in ms when the distance x is introduced in km.

Figure 7.3 shows the loss rate between any node and the central node during the whole execution of the experiment. In most of the communications the loss rate was under 0.2%. Two cases are remarkable: on the one hand, between the node 20 in Russia and the central node 0% of loss rate was measured, which means that no packets were lost; on the other hand, between Greece (node 16) and the central node, a loss rate of 15.68% was measured, maybe because of interruptions in the network, overload of the server in Greece or routed network fails. The mean of the loss rates between all the communications was 0.053% with a standard deviation of 0.097% without considering the node in Greece in the calculations.

Table C.1 summarizes the obtained measures. Column 5 and 6 show the mean and the standard deviation of the bandwidth measured in all the nodes connecting the central node in France. Column 7 and 8 also depict the mean and standard deviation of the latency in the nodes. Finally, column 9 represents the measured loss-rate between any node and the central node during the experiment.

7.2 GEO-Cloud Experiment Results

The scope of this project is to develop and implement in a Federated Future Internet infrastructure a simulation environment that allows experimenters testing different realistic Earth Observation scenarios, in which a constellation of satellites record images of the world surface in a daily basis, which are downloaded to a distributed network of ground stations, processed and distributed through a data center computed in cloud.

The experiment carried out to demonstrate the correct implementation of the GEO-Cloud system consisted in the execution of the "Scenario 1: Emergencies-Lorca Earthquake" (see Section D). This scenario focuses in the Earthquake that took place in the city of Lorca in May 11th 2011. When the natural disaster occurred it was required to have as soon as possible recent images of the affected area in order to assess the damages and plan the emergency services involvement. Satellite imagery has been proven to be an indispensable tool to face natural disasters. Satellite imagery is then required before, during and after the disaster occurred.

Firstly, the whole system was started and configured, and the graphical user interface executed. Notice that the Ground Stations of the *Space System Simulator* was configured with the network impairments measured in the PlanetLab experiment (see Section 7.1). The Table 7.1 shows these values.

Once the experiment was started by clicking the start button in the GUI the system started working in an integrated fashion:

- The satellite connects with the ground station.
- The satellite downloads the acquired data to the ground station.
- The cloud ingests, process, archive and catalogue the images resulting from processing the acquired data by the satellites.

In the simulation of the scenario, the *Virtual Wall* nodes were reached and they received the command to start the scenario. These nodes are the 12 ground stations and the constellation formed by 17 satellites. The ground stations were initialized by the *GUI* by sending the identity of each ground station, and the scenario. The same occurs for the satellite constellation. The GUI sends to each satellite a number to identify it and the scenario. The

Ground Station	Bandwidth [Mbps]	Latency [ms]	Loss-rate
Irkutsk	2.215692	242.297943	0.024
Puertollano	15.594752	27.197632	0.005
Svalbard	7.319547	59.234261	0.006
Troll	1.422580	340.900785	0.204
Chetumal	3.360441	154.209601	0.010
Córdoba	0.215605	302.688424	0.240
Dubai	0.215605	302.688424	0.240
Kourou	2.292386	70.081679	0.001
Krugersdorp	1.568929	317.206839	0.076
Malaysia	2.293113	207.366898	0.010
Prince Albert	2.229976	201.515008	0.063
Sidney	1.331592	375.310920	0.004

Table 7.1: Impairments implemented in *Virtual Wall*.

Satellite System Simulator and the *Ground Station System Simulator* started their executions. The satellites acted regarding the data obtained from the STK software. There are different behaviours depending on the zone in which the satellite enters (see Section 5.2). In the Scenario 1, the satellites which enters into any Ground Station visibility zone are: 1, 7, 4, 5 and 11. The durations of each access are shown in the Table 7.2.

Satellite Number	Ground Station	Duration of the passes
5	Troll	23400
6	Troll	23400
8	Krugersdorp	23400
11	Puertollano	23400
12	Puertollano	16813
12	Svalbard	16813
13	Svalbard	23400

Table 7.2: Satellite accesses in Scenario 1

The only satellite that acquires Area of Interest (AOI) in this scenario is the satellite 11. The others satellites enters in their corresponding footprints and they download the acquire non AOI raw data files into the Ground Stations. When the satellite 11 enters into the Puertollano footprint, a connection with the Puertollano Ground Station is stablished. The satellite 11 starts imaging at $1395\ Mbps$ the Lorca surface when flies Spain and at the same time, it starts to download at $160\ Mbps$ the data to the Puertollano ground station. During the duration of this scenario (23.4 seconds) an image of the Lorca region is acquired and downloaded in the Puertollano ground station. The other satellites download at $160\ Mbps$ non AOI data until the scenario ends.

When the satellite 11 leaves of the Puertollano footprint, the ground station counts the

received packets. These packets are "AOI" packets, so all the data belong to AOI area. An AOI raw data file is created in the Puertollano ground station. The other satellites download non "AOI" packets during the scenario duration at $160\ Mbps$. For example, the Svalbard ground stations receives the packets from the satellite 12. This ground station creates a non AOI image in its memory.

The features of raw data are summarized as follows:

- The raw data occupied $302\ MB$.
- The raw data represented a scene acquired by a satellite. A scene is a piece of the Earth surface. In this case represented the Lorca AOI zone.
- The raw data scene is splitted on 8 sectors. A scene is a set of metadata and values representing a piece of the total image acquired.
- The raw data is conformed by the data of the different spectral bands together the metadata. The metadata are the geolocation data and the decompression data among others.

Meanwhile, the *Orchestrator* is periodically polling the ground stations FTP connections for detecting when an image is created. When it detected that the Puertollano ground station created an *AOI* image, it automatically downloaded it. For the others ground stations the same. Between a *Virtual Wall* node (ground station) and a *BonFIRE* node (the *Orchestrator*), around $235\ Mb/s$ is the transfer data rate. However the simulated bandwidth were customized with the results showed in Table 7.1.

Then, when the AOI image of Puertollano ground station was detected, the *Orchestrator* downloaded it through FTP. The *Orchestrator* queued the raw data for processing. If the *Processing Chain* module were free, the *Orchestrator* sent that data for processing. If the *Processing Chain* were busy, the *Orchestrator* queued the raw data until the *Processing Chain* were free. As the Lorca image is the first acquired raw data of the scenario, it was sent to process. The *BonFIRE* machines are interconnected through *Gigabit Ethernet* interfaces, so the transfer data rate between the *Orchestrator* and the *Processing Chain* was high. Note that in the implementation using a shared storage, this data sending were avoid because the *Orchestrator* only sends the path of the raw data to the *Processing Chain*, so this sending time were nil.

Once the raw data was sent to the *Processing Chain*, the transformation from raw data to a geolocated and orthorectified image started. In this implementation the on demand creation of the *Processing Chains* could not be implemented due to the *Fed4FIRE* tools for that were not finished. Thus, when the raw data was in the *Processing Chain* component, it automatically started the processing. The processing accomplishes the L0, L0R, L1A, L1BG, L1BR, L1C stages. The functions of the different stages are explained in Section 5.4.1.2. As summary, the functions of the *Product Processors* were the following:

- *Level 0* obtained unprocessed images, in digital count numbers.
- *Level L1A* obtained calibrated products, in units of radiance.
- *Level L1B* obtained calibrated and geometrically corrected products (ortho-rectified), blindly geolocated. There are two L1B processors: L1BG and L1BR. Both *Product Processors* does the same but the first is configured manually and the second one, automatically.
- *Level L1C* obtained calibrated and geometrically corrected products (ortho-rectified), precisely geolocated using ground control points.

The required times obtained for each *Product Processor* are shown in the Table 7.3. In this case, these *Product Processors* were implemented with a datablock storage as background file system. These times were obtained by sampling and these are the mean values. The first column of the table "User Time" shows the time of the *Product Processors* used to the image process. The second column "System Time" represents the time that the system required to perform the system calls. The "Elapsed Time" is the total time required to process the image at each different level. The times of every I/O operations are included in it. The "RAM" column represents the required *RAM* memory to perform the process. The last column "CPU" is the spent Central Processing Unit (CPU) for processing this stage.

Processor	**User Time (s)**	**System Time (s)**	**Elapsed Time (HH:MM:SS)**	**CPU**	**RAM**
L0	84.13	4.51	00:1:42	86%	3.7%
L0R	100.34	17.34	00:1:50	106%	23.26%
L1A	116.73	17.24	00:1:48	122%	12.2%
L1BR	1065.62	568.48	00:17:42	153%	26.4%
L1BG	1112.4	164.92	00:17:10	123%	39%
L1C	229.33	5826.6	1:40:34	123%	39%

Table 7.3: Processing times of each product processor in the datablock storage

As it can be observed in the table, the *RAM* required for each *Product Processor* is different. This is because the actions performed by the *Product Processors* are different. For example the L0 stage slightly process the raw data and the other hand, the L1BR processor accomplishes the geolocation of the image, so *CPU* is more intense. The same is applied for the use of the *RAM* memory, which depends on the *Product Processor* actions. Furthermore, the uses of the CPU sometimes overload 100%. This is because the processing required more than one cores. Thus workload was over 100% means that the *Product Processor* occuped more than a node to carryied out the task.

The times required each *Product Processor* were different. The L1B required more time than the others because it accomplished the image geometric corrections. The actions performed by each *Product Processor* are explained in Section 5.4.1.2.

Consecuently, these results are contrasted with the times obtained when executing the processors in the shared storage. These results are depicted in Table 7.4 and there are significant differences in terms of the time.

Processor	User Time (s)	System Time (s)	Elapsed Time (HH:MM:SS)	CPU	RAM
L0	88.43	3.67	00:4:15	36%	3.7%
L0R	80.70	27.94	00:36:02	5%	23.26%
L1A	104.75	38.91	00:38:32	6%	12.2%
L1BR	788.96	14856.87	4:32:42	95%	26.4%
L1BG	1030.16	1763.40	1:45:27	44%	39%
L1C	345.21	152378.52	9:18:34	44%	39%

Table 7.4: Processing times of each product processor in the shared storage

The time each *Product Processor* required by using a shared storage as file system was more. The difference consisted of the NFS in which the shared storage was implemented and located in other *Fed4FIRE* testbed. This testbed was the *IBBT* platform which is located in Ghent (Belgium), so all the *I/O* operations such as reads and writes were performed using the Internet network. The delays on the communications, the congestion of the network and the packets distribution algorithms of the network provoked more delays than when performing locally or when using the shared storage.

To corroborate that hypotesis, a performance test in the shared storage was carried out. This test consisted of executing the following command:

> dd if=/dev/zero of=/mnt/shared/test-performance bs=1M count=2560 conv=fdatasync

It measured the write rate to the shared storage. The result obtained was the following:

- 2684354560 bytes (2.7 GB) copied, 1174.6 s, 2.3 MB/s

2.7 GB of data was copied in $1174.6\ s$, obtaining a data rate of $2.3\ MB/s$ between the *BonFIRE* machine and the store located in *IBBT*.

Then, the command was repeated for executing in the local storage as follows:

> dd if=/dev/zero of=/mnt/local/test-performance bs=1M count=2560 conv=fdatasync

It measured the write rate to the shared storage. The result obtained was the following:

- 2684354560 bytes (2.7 GB) copied, 50.975 s, 52.7 MB/s

2.7 GB of data was copied in $50.97\ s$, obtaining a data rate of $2.3\ MB/s$ between the *BonFIRE* machine and the store located in *BonFIRE* or in the machine.

Thus it is concluded that the bottleneck obtained in the products processors implemented in the shared storage was due to the implementation of the NFS storage which the *Fed4FIRE* testbed implemented.

As a result, the image of *Lorca* was processed in the following times:

- Using a datablock storage: $2\ h : 17' : 43''$
- Using a shared storage: $17\ h : 56' : 43''$

Once the AOI Lorca image was obtained, the *Processing Chain* sended it to the *Archive and Catalogue* module. The transfer data rate between *BonFIRE* machines was $1\ Gbps$, so the transferring time was short. By using the shared storage for implementing the background filesystem, this transferring time was avoid. It was stimated that this transferring time was about 30'. The *Archive and Catalogue* stored and catalogued the image. A workspace and a data store were created for storing the image. Once the image was in *GeoServer*, layers, tiles and image pyramids were created. The cataloguing time of any image was near to cero in the local implementation of the storage, and it was nil in the implementation of the shared storage because the image was already in the filesystem. The next step consisted of to publish the catalogue by using a CSW service and the web interface. By using the web-browser, the Lorca image was visualized in the map-view of *GeoServer*. The image was catalogued in a workspace and it was visualized and downloaded. The url to access the web interface is formed as follows:

- `http://IP_address_GeoServer\/geoserver`

where the *IP_address_GeoServer* was the IP address of the *Archive and Catalogue* machine.

Furthermore, the implemented architecture in ICE was checked. A client to test that architecture was accomplished. The *Orchestrator* component ingested the images acquired by the satellites in the system, the *Processing Chains* transformed the raw data to orthorectified and geolocated images, and finally the *Archive and Catalogue* performed the archiving and cataloguing successfully.

Finally, a video at real time system execution named "video-demo" is included in the "video" folder in the attached CD-ROM.

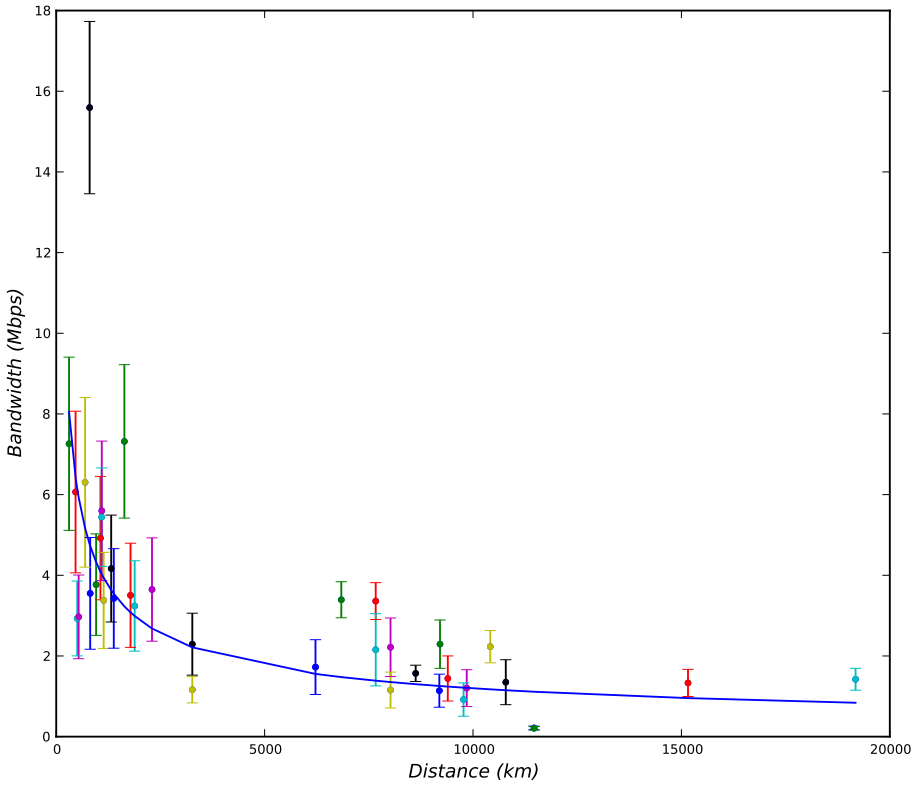
7.3 Publications

The publications that has led this project are listed as follows:

- Gerardo González, Rubén Pérez, Jonathan Becedas, Manuel Latorre and Félix Pedrera, "Measurement and Modelling of PlanetLab Network Impairments for Fed4FIRE's GEO-Cloud Experiment". *IEEE 26th International Teletraffic Congress - Workshop on Future Internet and Distributed Clouds (ITC 2014)*, Karlskrona, Sweden, September 2014.
- Rubén Pérez, Gerardo González, Jonathan Becedas, Manuel Latorre and Félix Pedrera, "TesTest Cloud Computing for Massive Space Data Processing, Storage and Dis-

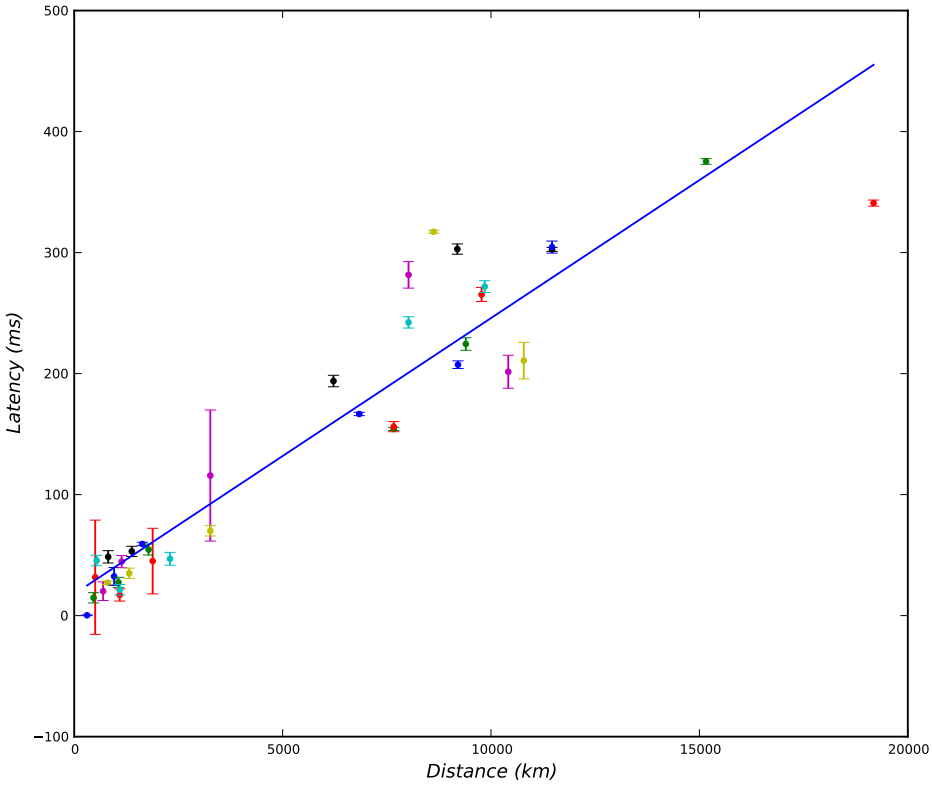
tribution with Open-Source Geo-Software". *FOSS4G-Europe Congress - Workshop on Free and Open Source Software for Geospatial, INSPIRE and Big Data (FOSS4G-Europe 2014)*, Bremen, Germany, July 2014.

- Jonathan Becedas, Gerardo González, Rubén Pérez, Manuel Latorre and Félix Pedrera, "Cloud Architecture for Processing and Distribution of Satellites Imagery". *FOSS4G-Europe Congress - Workshop on Free and Open Source Software for Geospatial, INSPIRE and Big Data (FOSS4G-Europe 2014)*, Bremen, Germany, July 2014.
- Manuel Latorre, Félix Pedrera Jonathan Becedas, Gerardo González and Rubén Pérez, "Validation of an experimental on-demand cloud infrastructure for Earth Observation Web Services". *FOSS4G-Europe Congress - Workshop on Free and Open Source Software for Geospatial, INSPIRE and Big Data (FOSS4G-Europe 2014)*, Bremen, Germany, July 2014.
- Jonathan Becedas, Gerardo González, Rubén Pérez, Manuel Latorre and Félix Pedrera, "Validation of an Experimental Cloud Infrastructure for Earth Observation Services". *RAQRS'IV Congress - Workshop on Recent Advances in Quantitative Remote Sensing*, Valencia, Spain, September 2014.



Node 0: $\mu = 7.261104$, $\sigma = 2.147249$	Node 19: $\mu = 2.292386$, $\sigma = 0.767429$
Node 1: $\mu = 6.064255$, $\sigma = 2.003282$	Node 20: $\mu = 1.724467$, $\sigma = 0.679163$
Node 2: $\mu = 2.930276$, $\sigma = 0.927951$	Node 21: $\mu = 3.393083$, $\sigma = 0.446765$
Node 3: $\mu = 2.969305$, $\sigma = 1.037486$	Node 22: $\mu = 3.360441$, $\sigma = 0.456685$
Node 4: $\mu = 6.304907$, $\sigma = 2.102949$	Node 23: $\mu = 2.153858$, $\sigma = 0.895294$
Node 5: $\mu = 15.594752$, $\sigma = 2.135783$	Node 24: $\mu = 2.215692$, $\sigma = 0.725628$
Node 6: $\mu = 3.551883$, $\sigma = 1.385088$	Node 25: $\mu = 1.155372$, $\sigma = 0.444665$
Node 7: $\mu = 3.769356$, $\sigma = 1.258299$	Node 26: $\mu = 1.568929$, $\sigma = 0.202462$
Node 8: $\mu = 4.922084$, $\sigma = 1.529851$	Node 27: $\mu = 1.139138$, $\sigma = 0.409484$
Node 9: $\mu = 5.441677$, $\sigma = 1.220430$	Node 28: $\mu = 2.293113$, $\sigma = 0.599198$
Node 10: $\mu = 5.598359$, $\sigma = 1.728834$	Node 29: $\mu = 1.441929$, $\sigma = 0.559358$
Node 11: $\mu = 3.378253$, $\sigma = 1.191509$	Node 30: $\mu = 0.919341$, $\sigma = 0.412962$
Node 12: $\mu = 4.167417$, $\sigma = 1.324835$	Node 31: $\mu = 1.202864$, $\sigma = 0.457197$
Node 13: $\mu = 3.426796$, $\sigma = 1.234180$	Node 32: $\mu = 2.229976$, $\sigma = 0.400011$
Node 14: $\mu = 7.319547$, $\sigma = 1.902225$	Node 33: $\mu = 1.349700$, $\sigma = 0.557100$
Node 15: $\mu = 3.503871$, $\sigma = 1.291074$	Node 34: $\mu = 0.215605$, $\sigma = 0.044660$
Node 16: $\mu = 3.240286$, $\sigma = 1.120166$	Node 35: $\mu = 0.207834$, $\sigma = 0.045172$
Node 17: $\mu = 3.647157$, $\sigma = 1.280860$	Node 36: $\mu = 1.331592$, $\sigma = 0.336595$
Node 18: $\mu = 1.162643$, $\sigma = 0.326491$	Node 37: $\mu = 1.422580$, $\sigma = 0.270814$

Figure 7.1: Bandwidth of all nodes



Node 0: $\mu = 0.273073$, $\sigma = 0.301535$
Node 1: $\mu = 14.745501$, $\sigma = 4.261593$
Node 2: $\mu = 31.728539$, $\sigma = 47.203988$
Node 3: $\mu = 45.529232$, $\sigma = 4.288186$
Node 4: $\mu = 20.186817$, $\sigma = 7.702484$
Node 5: $\mu = 27.197632$, $\sigma = 1.418612$
Node 6: $\mu = 48.594328$, $\sigma = 5.097654$
Node 7: $\mu = 32.414597$, $\sigma = 7.345108$
Node 8: $\mu = 27.521367$, $\sigma = 4.136220$
Node 9: $\mu = 17.093975$, $\sigma = 5.070244$
Node 10: $\mu = 21.461830$, $\sigma = 4.278779$
Node 11: $\mu = 44.628323$, $\sigma = 5.033462$
Node 12: $\mu = 35.037211$, $\sigma = 4.226534$
Node 13: $\mu = 53.103548$, $\sigma = 4.204185$
Node 14: $\mu = 59.234261$, $\sigma = 1.382998$
Node 15: $\mu = 54.617127$, $\sigma = 4.565056$
Node 16: $\mu = 45.069598$, $\sigma = 27.110465$
Node 17: $\mu = 46.972868$, $\sigma = 5.292955$
Node 18: $\mu = 115.761992$, $\sigma = 54.158721$
Node 19: $\mu = 70.081679$, $\sigma = 4.243071$
Node 20: $\mu = 193.865830$, $\sigma = 4.743945$
Node 21: $\mu = 166.633028$, $\sigma = 1.364346$
Node 22: $\mu = 154.209601$, $\sigma = 1.314127$
Node 23: $\mu = 156.166992$, $\sigma = 4.195054$
Node 24: $\mu = 242.297943$, $\sigma = 4.664348$
Node 25: $\mu = 281.606416$, $\sigma = 10.965620$
Node 26: $\mu = 317.206839$, $\sigma = 1.226175$
Node 27: $\mu = 302.908380$, $\sigma = 4.257230$
Node 28: $\mu = 207.366898$, $\sigma = 3.150162$
Node 29: $\mu = 224.464973$, $\sigma = 5.175334$
Node 30: $\mu = 265.410228$, $\sigma = 5.815682$
Node 31: $\mu = 271.900753$, $\sigma = 4.865709$
Node 32: $\mu = 201.515008$, $\sigma = 13.641897$
Node 33: $\mu = 210.717257$, $\sigma = 15.094545$
Node 34: $\mu = 302.688424$, $\sigma = 1.628971$
Node 35: $\mu = 304.554190$, $\sigma = 4.977954$
Node 36: $\mu = 375.310920$, $\sigma = 2.353757$
Node 37: $\mu = 340.900785$, $\sigma = 2.509783$

Figure 7.2: Latency of all nodes

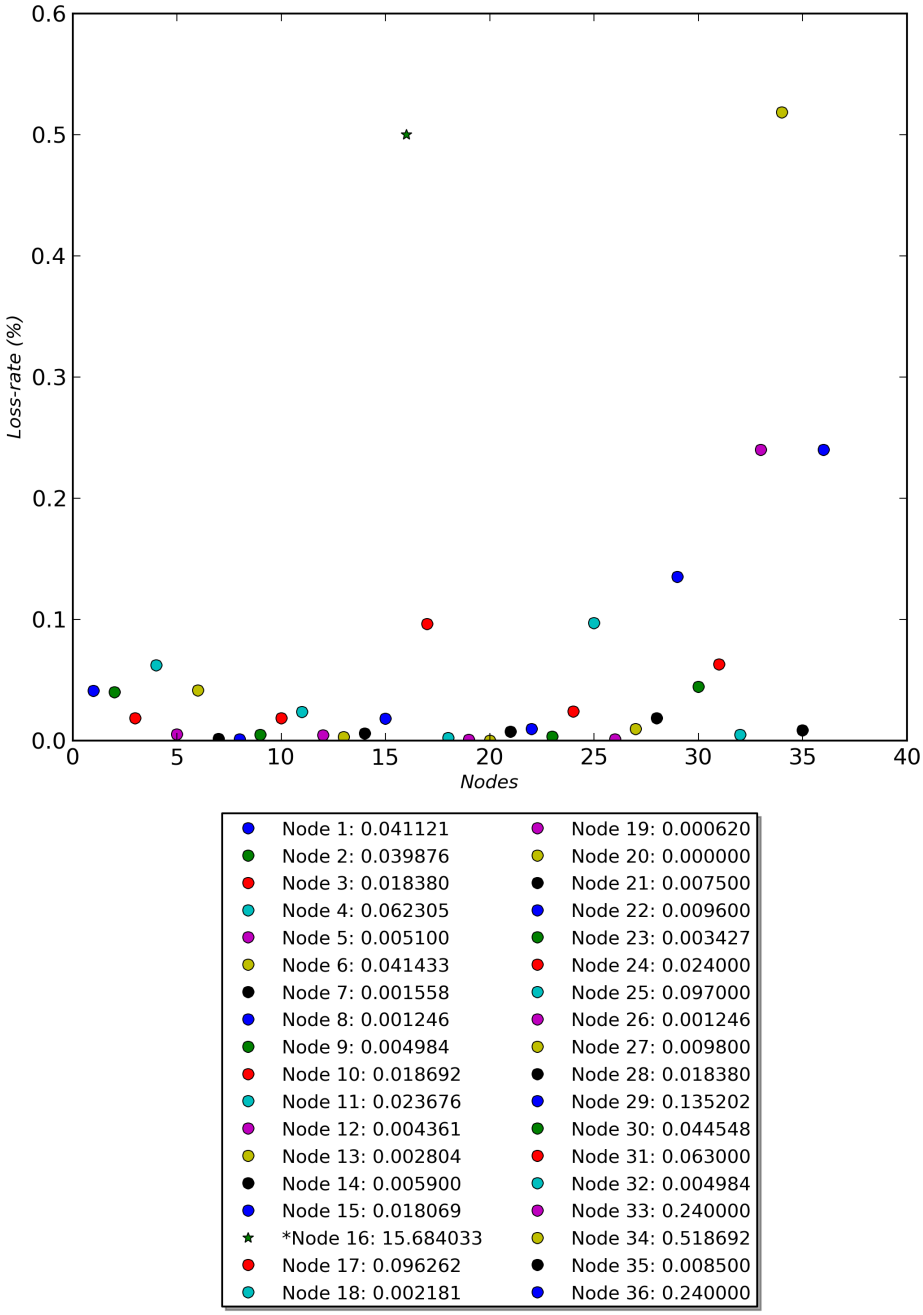


Figure 7.3: Bandwidth of all nodes.

Chapter 8

Conclusions

THE objectives and subobjectives proposed in the Chapter 3 were tackled for their implementation during the development time (see Chapter 6). In this Chapter the main aspects obtained of the accomplished milestones, the relevant solutions and decisions adopted, the problems encountered and finally some interesting lines of research are detailed.

8.0.1 Accomplished objectives

The objectives and subobjectives described in Chapter 3 were performed. GEO-Cloud is a modular and multiplatform cloud architecture for Earth Observation which allows the simply and fastly deployment and execution of experiments.

The main objective can be summarized as the modelling and the implementation of a close to real world Earth Observation system in the *Fed4FIRE* platform. This implementation includes the implementation of the *Space System Simulator* , the *PlanetLab* experiment and the implementation of the cloud system. They together allows to simulate a real constellation obtaining images of the Earth surface and processing these images in a cloud architecture for collecting results.

In the *PlanetLab* experiment, topology networks were created to simulate the communications between the ground stations and the cloud infrastructure and between this and the end users accessing the web services. The experiment consisted of measuring the real network impairments (bandwidth, latency and loss rate) to model them. NEPI, Iperf and Ping were used to measure the impairments. The experience with those tools was positive, mainly with NEPI, which facilitated the implementation and automation of different processes very easily. 21600 trials during 6 hours continuous execution were carried out between any node and the central node. A normal distribution was adjusted to the bandwidth and the latency measured between each pair of nodes. The loss rate was obtained in percentage.

After obtaining the impairments for all the nodes, the bandwidth and the latency were fitted in function of the distance between nodes. The bandwidth best fitting was a hyperbolic function and that the latency linearly increased with the distance between nodes. In addition, the loss rate was obtained to be 0.053% in mean, with a standard deviation of 0.097%. The previous results allowed to extrapolate the results to do load tests to the system implemented

in GEO-Cloud.

In the *Virtual Wall* testbed, the *Space System Simulator* was created to simulate the satellite constellation and the ground station network distributed around the world. The *Space System Simulator* was splitted on two parts: the *Satellite System Simulator* and the *Ground Station System Simulator*. With the first one, the simulation of the designed satellite constellation was accomplised. The data of each satellite and each satellite access to each ground station were obtained. Then, these data was processed, sorted and stored in a database.
The types of acquisition were analysed and identified in the scenarios. Thus, different kinds of functions were performed in order to schedule them depending on the area in which the satellite is in each moment. Moreover, the raw data, the bandwidth that the network between the satellites and the ground stations were analysed. The bandwith requirements was $160\ Mbps$ and this network did not provide this data rate. So, several kinds of packets meaning the satellite acquisition were accomplished.
The satellite constellation performed the simulation of the real constellation. The second one is composed by the network of ground stations distributed around the world. The results obtained in the *PlanetLab* experiment were used to customize the ground stations links between the node and the *BonFIRE* nodes. This allowed to simulate the real network behaviour of the real ground stations (see Section 5.1.1), connected to the *BonFIRE* cloud node. Then, the process of the packets were done.
The creation of images by counting the receives packets of data were elaborated. At the end, the creation of the raw data files to be ingested by the *Orchestrator* in the cloud was performed.

In the *BonFIRE* testbed, the cloud architecture was developed. This infrastructure for EO provided the automatic ingestion of raw data and processing the raw data to obtain images, and later storing and cataloguing transparently, dynamically and automatically. The *Orchestrator*, *Processing Chain* and the *Archive and Catalogue* were implemented. At first, the interfaces between the system were studied. Then, these components were designed and accomplished. The *Orchestrator* automatically ingested the raw data of the *FTP* ground stations connections. Then, these data was sent to the *Processing Chain* module. It processed the raw data obtaining the geolocated and orthorectified images.
Finally, these images were catalogued in the *Archive and Catalogue* module. Furthermore, there were implemented two filesystems: the first one was the datablock storage which was locally to each component, and the second one was the shared storage. This last one was shared between all the cloud components facilitating the implementation and deleting the data submissions between the cloud components. The performance in these filesystems was calculated.

The Graphical User Interface was implemented. It managed the nodes implemented in *Virtual Wall* testbed. It provided the experimentation by executing a scenario automatically. It

connected with the *Virtual Wall* nodes and started the simulation of the scenario in them.

The next step was to create the experiment integrating the developments in all testbeds. Then, the cloud architecture implemented in *BonFIRE* was validated. One of the defined scenarios was executed and the results were obtained. The cloud architecture, the *Space System Simulator* and the GUI were integrated and worked properly.

8.1 Fed4FIRE infrastructure conclusion

8.2 Future Work

The GEO-Cloud experiment is a cloud implementation for testing and validating the use of the cloud platforms and the future internet technologies in EO systems.

Next, several lines for research are proposed:

- **To improve the cloud implementation**: The cloud implementation presented in this project can be more flexible and dynamic. The *Processing Chain* was statically implemented due to the troubles with the *Fed4FIRE* testbeds, so as future work the implementation of this component as a dynamic and on demand component by using an elastic service or similar, is proposed. Furthermore, the *Orchestrator* does not manage all the stages of the processing independently. This approach consists of to create several *Processing Chains* components in order to the *Orchestrator* can select which *Processing Chain* is less overloaded and to implement a load balancing algorithm. In addition, the approach of the implementation of the *Processing Chain* component as a Grid is proposed.
- **To check the cloud implementation in other cloud platforms**: The cloud architecture implemented was only tested in the *BonFIRE* cloud with the services which that platform provides. It would be interesting to test the architecture in others cloud systems such as *Amazon*, *Azure*, *IBM Cloud* and *Citrix* among others.
- **To implement the ICE architecture in cloud**: The implementation of the ICE architecture provides an integrated cloud system for EO images processing. This architecture was tested in a local machine, so it is proposed to implement it and to check it in a cloud platform. The *Processing Chain* components were joinned in a Replica Group for load balancing. It would be interesting to implement them forming a Grid and to take advantage of the free resources of the system.
- **To integrate the GUI**: To execute an experiment, it is necesary to accomplish the deployment of the nodes of both testbeds *BonFIRE* and *Virtual Wall* before to execute the GUI. Thus, it may be interesting to deploy an experiment without knowing where it is deployed and how the set up of the nodes was performed.
- **To improve the *Space System Simulator*** : The *Space System Simulator* was ad-hoc implemented for GEO-Cloud. The implementation of a customizable *Space System*

Simulator may be potentially important to deploy experiments with other kind of satellites and features.

- **Building a Web Site**: A major milestone for the expansion of this system is building a website where you can download the source code of this platform. Moreover a section may be added to lay tutorials for new users to have more facilities in their learning use. Another interesting section would be a bug tracker where experimenters can make suggestions and report bugs.

8.3 Personal conclusion

The development of a system like GEO-Cloud, and any Master Thesis, generally implies a close to real work experience that any aspirant to be a Computer Engineer will play for the rest of his life. It must be studied and domining a wide range of areas, platforms, libraries and work methods, always thinking of the end user. This is a new approach compared to the role which the student has done during his formative years which brings maturity to deal with their immediate future.

When a student performs his Master Thesis, he shows that the Computer Engineers are not secondhand engineers that repair computers, or format operating systems. A Computer Engineer must have quite knowledge of many areas and very diverse techniques, such as linear algebra, networking, distributed systems, language processors, computer architecture, software engineering or databases.

In my opinion, we must defend our identity with dignity, always asking for a deal and a respect commensurate with our training up to other engineering. That makes us rated engineers in addition to carrying a full work around us of other kind of engineers.

Computer Engineering is a profession with much potential, growth and versatility on market, so it is an honour and a pleasure to have made this choice five years ago.

APPENDIX

Appendix A

User Manual

This appendix explains the project operation by running the "Scenario 1: Emergencies-Lorca Earthquake" defined in appendix D. To execute it the experimenters must create an account in the *Fed4FIRE* platform and after that, they can already run the experiment. Then, the *Rspec* of the nodes in *Virtual Wall* is obtained from *JFed*. The GUI provides an automatic manner to execute the experiment.

A.1 First steps

To experiment with the *Fed4FIRE* testbeds, the following steps have to be done (for more information, see `http://doc.fed4fire.eu/`):

- To create an account in *Fed4FIRE* in order to get some certificates to access testbeds.
- To create an account in *BonFIRE* for getting access in this testbed. At this moment, in order to use *BonFIRE*, the experimenters also need to register for an additional account.
- To create an SSH key and upload it in both the *Emulab* and the *BonFIRE* account.
- To download the X.509 certificate which authorizes and authenticates the in all the *Fed4FIRE* tesbeds.

A.1.1 Creating a Fed4FIRE account

The *Fed4FIRE* webpage for creating an account is available in `http://www.wall2.ilabt.iminds.be`. Once the page was loaded, the user has to select the option "Request an Account" which is located on the middle of the page.

Then, the experimenter must fill the registration form and to click in the "Submit" button. The account is created and it shall be validated by the *Fed4FIRE* reviewers.

A.1.2 Creating a BonFIRE account

To a *BonFIRE* account, the experimenter has to visit the platform webpage `http://portal.bonfire-project.eu/`. Then, the experimenter fills the registration form and sends it for approving.

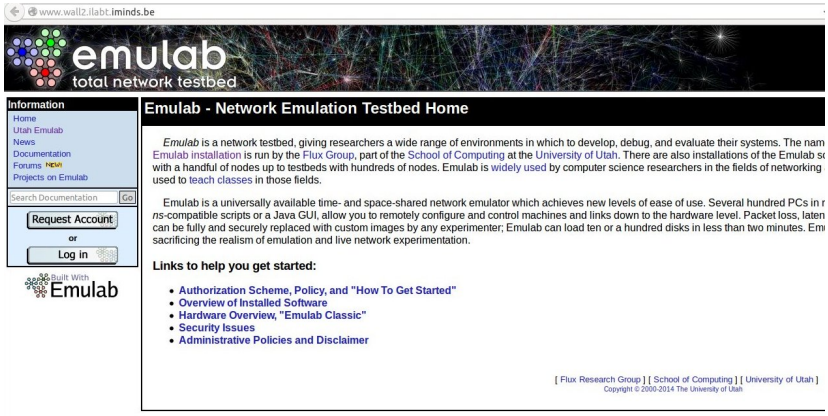


Figure A.1: Fed4FIRE webpage for creating an account

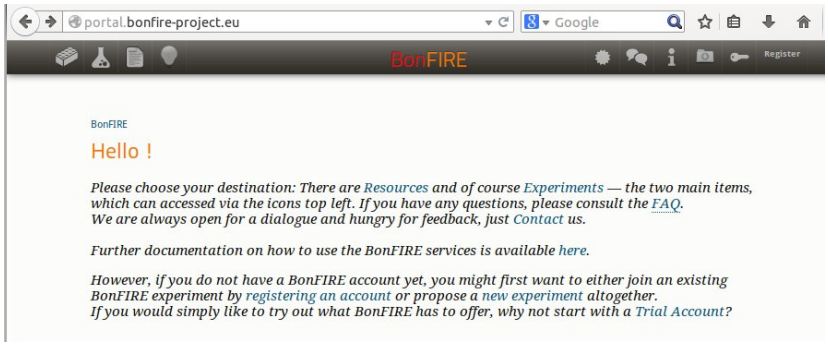


Figure A.2: BonFIRE platform webpage

A.1.3 Creating an SSH key

Once the experimenter is registered in the *Fed4FIRE* platforms, it is necessary to create a pair of RSA keys and to upload the generated public key to *BonFIRE* and *Fed4FIRE* web plattforms. The creation of the RSA keys, depending on the operative system, can be done as follows:

- *Windows platforms*: *Putty* can be used. *PuTTY* is a widely used SSH client for Windows and it includes the tool *PuTTYgen* to create an SSH key. *PuTTY* can be downloaded from `http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html`.

- *Unix platforms*: For UNIX platforms, creating an SSH key can be done through a command line tool in a terminal. The command that performs that is the following: *"ssh-keygen -t rsa"*.

A.2 Execution of a Scenario

The prerequisites for a experimenter can perform the execution of a scenario are the following:

- The *Space System Simulator* is running in the *Virtual Wall* testbed
- To execute the software of cloud components.
- To execute the GUI to manage the execution.

A.2.1 Execution of the Satellite System Simulator in Virtual Wall

To start the execution of the *Space System Simulator* in *Virtual Wall* , it is neccesary to do the implementation as the Section 5.3 explains.

Firstly it is necesary to obtain the *Rspec* of the deployed nodes in *JFed*. Openning *JFed*, the specification is located in "source/vw/Description.rspec" (see Figure A.3). Then, the Rspec is shown in the window as the Figure A.4 shows.

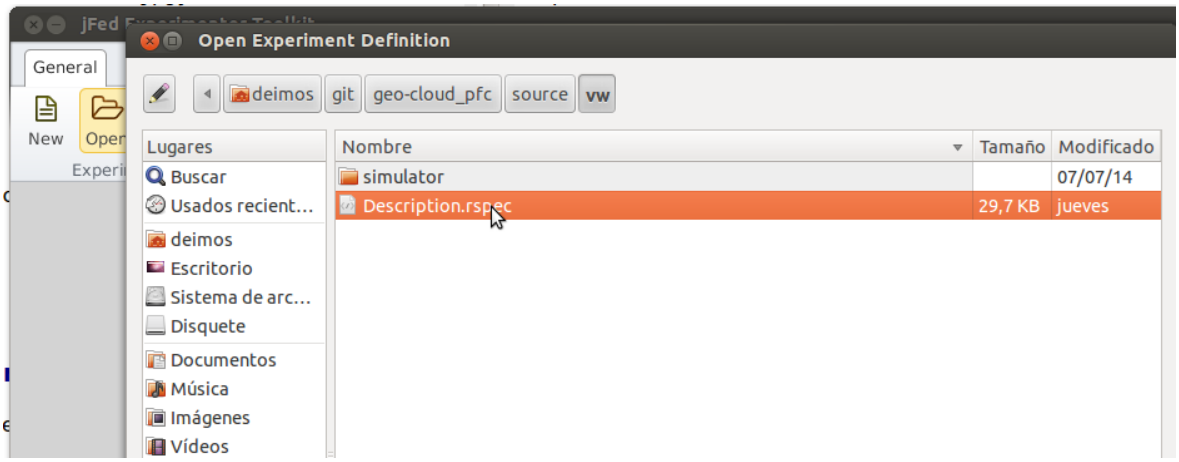


Figure A.3: Loading specification in JFed

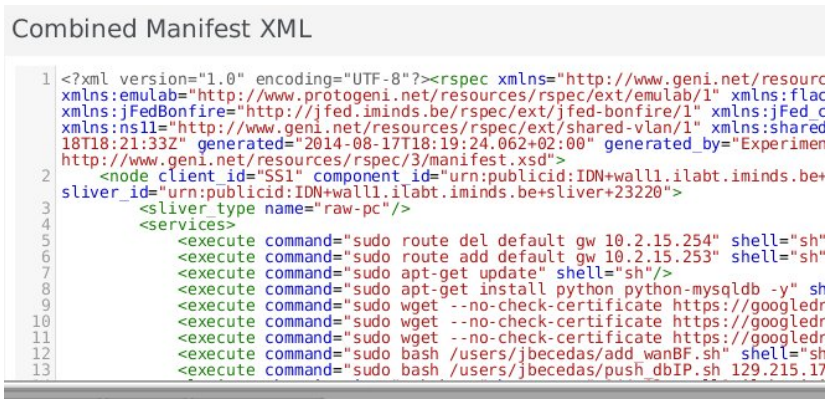


Figure A.4: Rspec in JFed

The execution of the *Space System Simulator* is executed by clicking of the *Start* button of the *JFed* interface. Thus, the experiment is executed (see Figure A.5).

At the end, all the nodes of the *Space System Simulator* are deployed and ready to execute the software by using the *GUI* (see Figure A.6).

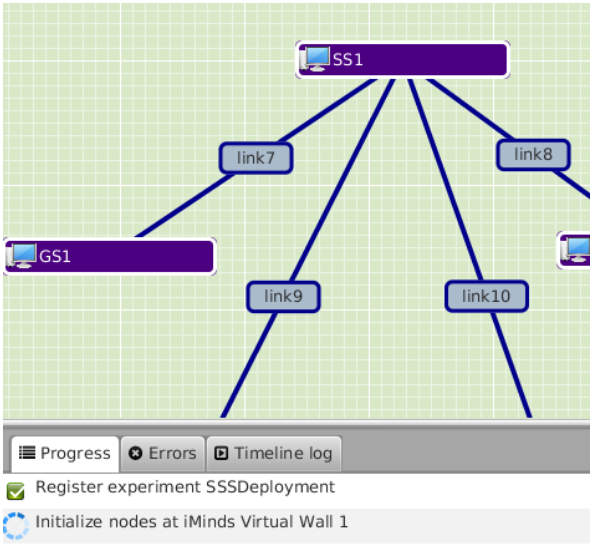


Figure A.5: Running experiment in JFed

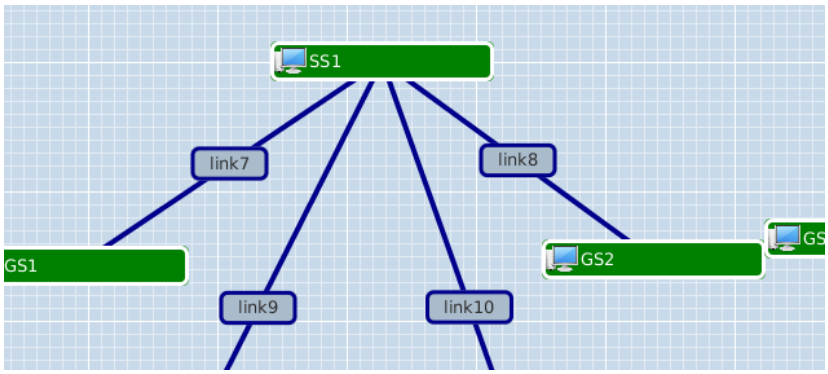


Figure A.6: Nodes deployed in JFed

A.2.2 Execution of the Cloud Architecture in BonFIRE

The execution of the cloud components are done as the Section 5.4.2 shows. As summary, the orchestrator file config has to be configurated with the *Processing Chain* machine address, the database address and the *Archive and Catalogue* machine address. Then, the *GeoServer* software in the *Archive and Catalogue* machine is required in order to the Orchestrator can be executed. Then, the *Orchestrator is executed* (see Figure A.7) and the other components of cloud are executed on demand as soon as the *Orchestrator* requires them for processing images and archiving and cataloguing them.

A.2.3 Execution of the GUI

Once the cloud architecture and the *Space System Simulator* are running respectively in *BonFIRE* and *Virtual Wall* and their setups were done, the system is ready for executing the scenarios described in Annex D. The scenario selected in order to explain this section was the “Scenario 1: Emergencies-Lorca Earthquake”.

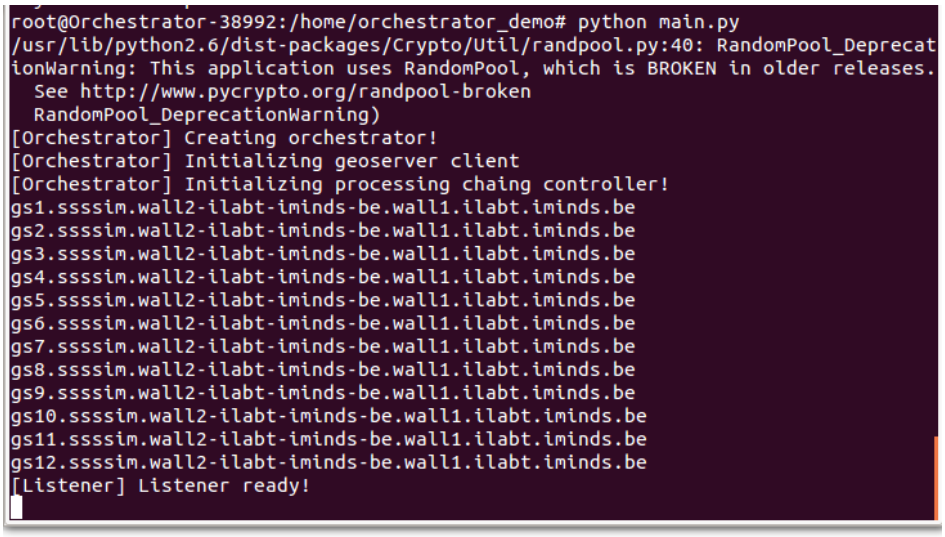


Figure A.7: Orchestrator execution

The Figure A.4 shows the *Rspec* specification which the user has to fully select and copied into the "gui/resources/jfed.out" file, replacing its content.

The next step is to execute the graphical user interface for experimenting. The GeoCloud GUI can be executed as "python main.py" in the main directory of the GUI "source/gui". Before that it is necessary to copy the *Rspec* specification of the *JFed* when the experiment is running. This is because the *Experiment Controller* component of the GUI

The main window appears and the experimenter can select the required scenario.

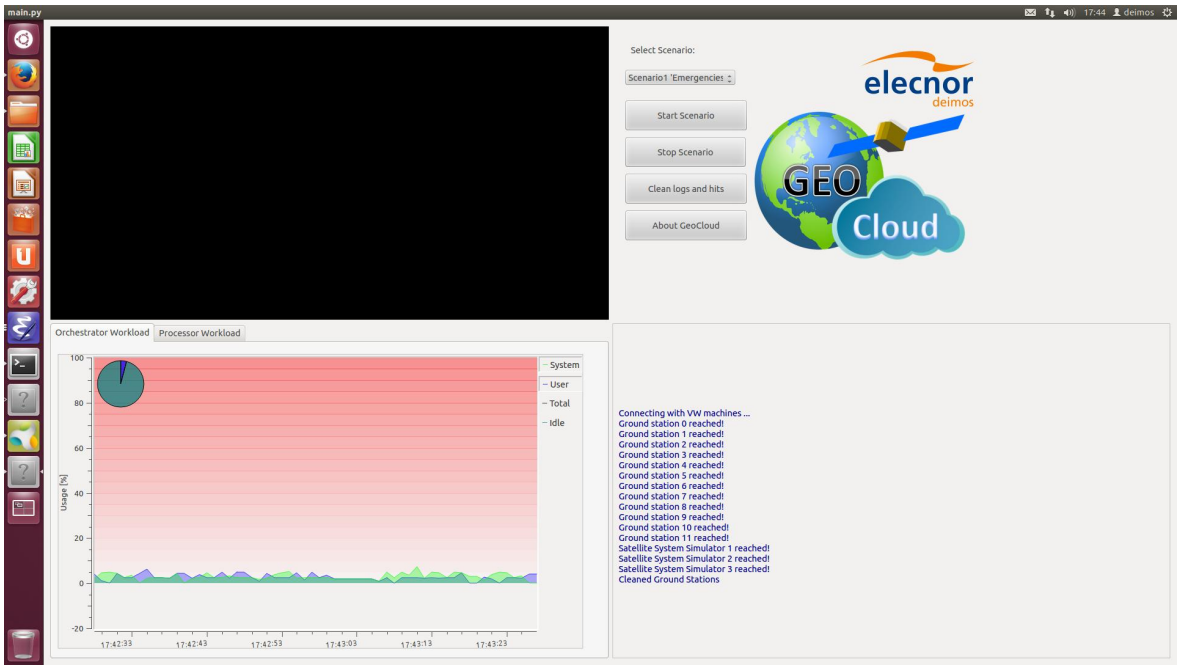


Figure A.8: GUI main window

When the experimenter clicks the start button, the scenario begins to run and the following actions are carried out: in the Log framework, several messages indicating the state of the execution appear; in the Workload framework, it is displayed the workload of both orchestrator and processing chain machines; and in the Video framework, the simulation of the satellite constelation is displayed.

During the scenario execution, the acquired images from the satellites are processed in the processing chain machine and sent to the catalogue. In the end, these images are available in the web service provided by the *Archive and Catalogue* module of the cloud.

Figure A.9: GeoServer catalogue

A.2.4 Collection of the results

While a scenario is executing, the *Orchestrator* is receiving all the interactions that in the cloud system are carrying through. Basically, the experimenter has to connect to the *Orchestrator* machine and to get the "orchestrator.log" file for obtaining the experiment results. This file contains the interactions produced in the system time stamped.

Appendix B

PlanetLab Nodes

Ground Station	Node Location	Selected Node	Site	Node number
Irkutsk[1]	China	planetlab1.buaa.edu.cn	Beihang University	24
Puertollano	Spain	Planetlab2.dit.upm.es	Universidad Politécnica Madrid	5
Svalbard	Norway	planetlab1.cs.uit.no	University of Tromso	14
Troll[2]	New Zealand	planetlab1.cs.otago.ac.nz	University of Otago	37
Chetumal[3]	USA	Planetlab1.eecs.ucf.edu	University of Central Florida	22
Córdoba	Argentina	planet-lab2.itba.edu.ar	Instituto Tecnológico Buenos Aires	34
Dubai[4]	Israel	planet1.cs.huji.ac.il	The Hebrew University of Jerusalem	19
Kourou[5]	Brazil	planetlab1.pop-pa.rnp.br	RNP	26
Krugersdorp[6]	Reunion Island (France)	lim-planetlab-1.univ-reunion.fr	Universite de La Reunion	28
Malaysia	Malaysia	planetlab1.comp.nus.edu.sg	National University of Singapore	32
Prince Albert	Canada	planetlab-2.usask.ca	University of Saskatchewan	21
Sidney	Australia	pl1.eng.monash.edu.au	National ICT Australia	36
Cloud[7]	France	ple6.ipv6.lip6.fr	University Pierre et Marie Curie	N/A

Table B.1: Ground Segment Nodes

Country	Selected Node	Number	Country	Selected Node	Number
Argentina	planet-lab2.uba.ar	35	Japan	planet1.pnl.nitech.ac.jp	31
Australia	pl1.eng.monash.edu.au	36	Korea, Republic of	netapp7.cs.kookmin.ac.kr	27
Belgium	rochefort.infonet.fundp.ac.be	2	The Netherlands	planetlab1.cs.vu.nl	4
Brazil	planetlab1.pop-pa.rnp.br	26	New Zealand	planetlab1.cs.otago.ac.nz	37
Canada	planetlab-2.usask.ca	21	Norway	planetlab1.cs.uit.no	14
China	planetlab1.cqupt.edu.cn	25	Poland	ple2.dmcs.p.lodz.pl	13
Czech Republic	planetlab1.cesnet.cz	8	Portugal	planet1.servers.ua.pt	11
Finland	planetlab-1.research.netlab.hut.fi	17	Russian Federation	plab1.cs.msu.ru	20
France	inriarennes2.irisa.fr	0	Singapore	planetlab1.comp.nus.edu.sg	33
Germany	planetlab02.tkn.tu-berlin.de	6	Spain	dplanet2.uoc.edu	3
Greece	planetlab1.ionio.gr	15	Sweden	planetlab2.s3.kth.se	16
Hong Kong	planetlab1.ie.cuhk.edu.hk	30	Switzerland	planetlab2.unineuchatel.ch	1
Hungary	planet2.elte.hu	12	Thailand	ple2.ait.ac.th	29
Ireland	planetlab-node-01.ucd.ie	10	United Kingdom	planetlab-2.imperial.ac.uk	9
Israel	planetlab2.tau.ac.il	18	United States	planetlab-04.cs.princeton.edu	23
Italy	planet-lab-node1.netgroup.uniroma2.it	7			

Table B.2: User Nodes

Appendix C

Summary Table of PlanetLab Results

The following table represents every selected nodes and their loss-rate, bandwidth and latency measured values. The meaning of columns are the following: CN means Central Node; BW means Bandwidth; μ means mean; σ means standard deviation and $\#$ means Number.

Country	Selected Node	#	Layer	BW μ [Mbps]	BW σ [Mbps]	Latency μ [ms]	Latency σ [ms]	Loss Rate %
France	ple6.ipv6.lip6.fr	CN	1.2	N/A	N/A	N/A	N/A	N/A
France	inriarennes2.irisa.fr	0	2	7.261104	2.147249	0.273073	0.301535	0.041
Switzerland	planetlab2.unineuchatel.ch	1	2	6.064255	2.003282	14.745501	4.261593	0.040
Belgium	rochefort.infonet.fundp.ac.be	2	2	2.930276	0.927951	31.728539	47.203988	0.018
Spain	dplanet2.uoc.edu	3	2	2.969305	1.037486	45.529232	4.288186	0.148
Netherlands	planetlab1.cs.vu.nl	4	2	6.304907	2.102949	20.186817	7.702484	0.062
Spain	Planetlab2.dit.upm.es	5	1	15.594752	2.135783	27.197632	1.418612	0.005
Germany	planetlab02.tkn.tu-berlin.de	6	2	3.551883	1.385088	48.594328	5.097654	0.041
Italy	planet-lab-node1.netgroup.uniroma2.it	7	2	3.769356	1.258299	32.414597	7.345108	0.002
Czech Republic	planetlab1.cesnet.cz	8	2	4.922084	1.529851	27.521367	4.136220	0.001
United Kingdom	planetlab-2.imperial.ac.uk	9	2	5.441677	1.220430	17.093975	5.070244	0.005
Ireland	planetlab-node-01.ucd.ie	10	2	5.598359	1.728834	21.461830	4.278779	0.019
Portugal	planet1.servers.ua.pt	11	2	3.378253	1.191509	44.628323	5.033462	0.024
Hungary	planet2.elte.hu	12	2	4.167417	1.324835	35.037211	4.226534	0.004
Poland	ple2.dmcs.p.lodz.pl	13	2	3.426796	1.234180	53.103548	4.204185	0.003
Norway	planetlab1.cs.uit.no	14	1.2	7.319547	1.902225	59.234261	1.382998	0.006
Greece	planetlab1.ionio.gr	15	2	3.503871	1.291074	54.617127	4.565056	15.68
Sweden	planetlab2.s3.kth.se	16	2	3.240286	1.120166	45.069598	27.110465	0.096
Finland	planetlab-1.research.netlab.hut.fi	17	2	1.120166	1.280860	46.972868	5.292955	0.002
Israel	planetlab2.tau.ac.il	18	2	1.162643	0.326491	115.761992	54.158721	0.042
Israel	planet1.cs.huji.ac.il	19	1	2.292386	0.767429	70.081679	4.243071	0.001
Russian Federation	plab1.cs.msu.ru	20	2	1.724467	0.679163	193.865830	4.743945	0.000

Country	Selected Node	#	Layer	BW μ [Mbps]	BW σ [Mbps]	Latency μ [ms]	Latency σ [ms]	Loss Rate %
Canada	planetlab-2.usask.ca	21	1.2	3.393083	0.446765	166.633028	1.364346	0.008
USA	Planetlab1.eecs.ucf.edu	22	1	3.360441	0.456685	154.209601	1.314127	0.010
United States	planetlab-04.cs.princeton.edu	23	2	2.153858	0.895294	156.166992	4.195054	0.003
China	planetlab1.buaa.edu.cn	24	1	2.215692	0.725628	242.297943	4.664348	0.024
China	planetlab1.cqupt.edu.cn	25	2	1.155372	0.444665	281.606416	10.965620	0.048
Brazil	planetlab1.pop-pa.rnp.br	26	1.2	1.568929	0.202462	317.206839	1.226175	0.076
Korea, Republic of	netapp7.cs.kookmin.ac.kr	27	2	1.139138	0.409484	302.908380	4.257230	0.001
Reunion Island, France	lim-planetlab-1.univ-reunion.fr	28	1	2.293113	0.599198	207.366898	3.150162	0.010
Thailand	ple2.ait.ac.th	29	2	1.441929	0.559358	224.464973	5.175334	0.018
Hong Kong	planetlab1.ie.cuhk.edu.hk	30	2	0.919341	0.412962	265.410228	5.815682	0.135
Japan	planet1.pnl.nitech.ac.jp	31	2	1.202864	0.457197	271.900753	4.865709	0.045
Malaysia	planetlab1.comp.nus.edu.sg	32	1	2.229976	0.400011	201.515008	13.641897	0.063
Singapore	planetlab1.comp.nus.edu.sg	33	2	1.349700	0.557100	210.717257	15.094545	0.005
Argentina	planet-lab2.itba.edu.ar	34	1	0.215605	0.044660	302.688424	1.628971	0.240
Argentina	planet-lab2.uba.ar	35	2	0.207834	0.045172	304.554190	4.977954	0.519
Australia	pl1.eng.monash.edu.au	36	1.2	1.331592	0.336595	375.310920	2.353757	0.004
New Zealand	planetlab1.cs.otago.ac.nz	37	1.2	1.422580	0.270814	340.900785	2.509783	0.204

Table C.1: Obtained values of PlanetLab Experiment

Appendix D

Definition of Scenarios

D.1 Scenario 1: Emergencies - Lorca Earthquake (Spain)

D.1.1 Scenario description

In May 11th of 2011, an Earthquake took place in Lorca, in the Region of Murcia (Spain) at 18:47 local time. It was a moderate magnitude 5.1 in moment magnitude scale (M), preceded by a 4.5M foreshock at 17:05 local time. The seismic affected to other regions as Almería, Albacete, Granada, Jaén, Málaga, Alicante, Ciudad Real and Madrid. Several aftershocks occurred up to 3.9M in the following days. See (Instituto Geográfico Nacional) for more information.

Murcia is the most active seismic zone in Spain. Shortly after the second earthquake struck, the Spanish government, at the request of regional government of Murcia, activated the Military Emergencies Unit, a branch of the Spanish Armed Forces responsible for providing disaster relief.

In these kind of natural disasters, emergencies units and humanitarian organizations demand accurate imagery before and after the disaster that has been proven to be very helpful to provide accurate Global Positioning System (GPS) navigation maps to reach affected areas, especially on developing and underdeveloped countries, as demonstrated in the 2010 Haiti earthquake or 2013 Haiyan Typhoon in Philippines (Hot).

D.1.2 Response

The Government needs the following service:

- The image of Lorca and surroundings before the first seismic activity is detected.
- The image of Lorca and surroundings when the first seismic activity is detected.
- The image of Lorca and affected regions in the day of the event is requested with urgency.
- After this day:
  - All the images daily acquired by the constellation are demanded during the first week, with urgency too, in order to manage emergency services and to assess the damages in the infrastructures.

  - Until the first month after the earthquake, weekly images are requested without urgency.
  - Then, monthly images also without urgency would be demanded until the first year after the event.

The humanitarian organizations and volunteers may need:

- Images of Lorca before the earthquake, available through WMS and tiles.
- Images of Lorca within the week after the earthquake, available through WMS and tiles.

D.1.3 Data distribution

The following data distribution mechanisms will be tested for this use case.

- FTP: Scenes are distributed directly to the government via FTP to be incorporated into the archives and used to their own criteria.
- WMS: To facilitate the analysis of the images to provide an adequate response to the event, a private Web Map Service will be provided. From the scenes of the affected region, composites images (mosaic) will be created and populated into the WMS data store. The timestamp of the images will be incorporated to provide support to temporal requests; providing a TIME parameter with a time value in the WMS request does this. Having a WMS enables to have a rational distribution of images among the implicated organizations (firefighters, civil defense, army, etc.). The images can be requested only for needed times and regions by specifying time ranges and geographical area of interest. The WMS can be consumed from the GIS tools used by the emergency response organizations designated by the government.
- Tiles: A public tiles service for the area of interest will be deployed. Humanitarian organizations and volunteers can create accurate maps of the affected area with these data. This service will have potentially high demand (thousands of volunteers accessing the service) in a very short period of time (during the crisis response). Tiles from imagery before and after the disaster will be published.

D.1.4 Area of Interest

The Area of Interest is the Region of Murcia in Spain (see Figure D.1).

D.1.5 Users

The main user of the service is the Spanish Government, and other organizations related to emergencies management. Other users could be volunteers and humanitarian organizations.

Figure D.1: Region of Murcia (Spain) (image from `www.20minutos.es`)

D.1.6 Service Type

The service type is a basic service that can eventually include a hosting service as well for the distribution of data to different emergency services and Estate Forces. Due to the urgency of the images, no added value would be required.

D.1.7 Processing Level

The processing level will be low due to the urgent. In this scenario we do not contemplate added value services for damage assessment.

D.1.8 Storage Level

The storage level will be low.

D.1.9 Communications Level

The communication level is urgent.

D.1.10 Demand Variability

The demand is highly variable in this service.

D.2 Scenario 2: Infrastructure monitoring. Affection in railway infrastructures by sand movement in desert areas (Spain)

D.2.1 Scenario description

New high performance railway lines at ambient conditions characterized by the presence of wind, sand dunes, sand in the air and high temperatures, require innovative developments technology to minimize the impact of these phenomena on different infrastructure elements during the operation.

Figure D.2: High Speed line Medina-La Meca in Saudi Arabia. Image from `https://www.thalesgroup.com/sites/default/files/asset/document/Gonzalo%20Ferre.pdf`

Remote sensing technologies can be applied to assess the risk of affection in the infrastructure and to minimize and prevent the impact: Sand movement and speed, risk maps, sand storms monitoring, floods monitoring, etc.

Satellite imagery can be combined with Unmanned Aerial Vehicles (UAV), with higher resolution and flexible operation modes, to provide data to assist in the deployment, operation and maintenance of these lines.

D.2.2 Response

The solicited service is based on the following premises:

- Coverage during operation of the infrastructure.
- Use of archive images to provide information about dunes movements and speed.
- Monthly revisit time.
- Urgent images only in case of storms and floods to inspect damages in the infrastructures.
- The images require an analysis to identify risks zones, dune and sand speed fields, etc.

D.2.3 Data distribution

- WMS: A Web Map Service will be deployed to distribute the scenes of the area of interest to the managers of the infrastructure. Imagery and derived products (risk maps, wind speed fields, etc.).
- Value added services (WFS, HTTP/REST, Tiles, etc.). Interactive value-added services may be provided to the managers in order to have monitoring tools.

D.2.4 Area of Interest

Several areas of interest have been detected: The Medina-La Meca (Saudi Arabia) railway line currently in construction and the south of Spain (Andalucía) where testing technologies with sensors and UAV are in development.

D.2.5 Users

- Private organizations, responsible of the construction and operation of the lines.
- Public institutions responsible of the management of the railway infrastructures.

D.2.6 Service Type

The service type is advanced. It includes mosaics of the area of interest and combination with other sources of information.

D.2.7 Processing Level

The processing level is high

D.2.8 Storage Level

The storage level is high.

D.2.9 Communications Level

The communication level is not urgent.

D.2.10 Demand Variability

The demand is variable.

D.3 Scenario 3: Land Management-South West of England

D.3.1 Scenario description

There is a need to characterize landscape and crops in the South West of England, United Kingdom. Besides, the users demand a multitemporal product which allows analysing the land cover dynamics and detecting changes. For this task, multitemporal classification and regression techniques are used to provide products exploiting the high revisit time, high spatial resolution, high swath and large number of bands in the system (Sensyf).

D.3.2 Response

The primary focus of the service is Landscape Management linked to the use of the land for agricultural purposes versus conservation.

The service is the following:

- Satellite images with a weekly frequency shall be offered, including the requested processing through a push type service.
- Multitemporal Land Cover Classification and Change Detection Service are needed. Land cover and land cover change products will be generated from mosaics with the same temporal frequency, weekly, and service type, push.

D.3.3 Data distribution

- WFS to provide direct access to the derived products: Land cover, etc.
- Value-added services (REST/HTTP, WMS, Tiles) for the creation of web applications and interactive solutions.

D.3.4 Area of Interest

The Area of Interest is the South West of England, United Kingdom, see Figure D.3.

Figure D.3: South West of England (image from `http://www.holiday-home-dealers.co.uk/`)

D.3.5 Users

The user of this service is the European Landscape Convention.

D.3.6 Service Type

The service type is high added value (Push). Processing is required to offer the requested services: crops characterization, classification, irrigation planning, and change detection.

D.3.7 Processing Level

The processing level is medium.

D.3.8 Storage Level

The storage level is medium.

D.3.9 Communications Level

The communication level is not urgent since the changes in the crops are not immediate and the frequency of the images acquisition is higher than such changes.

D.3.10 Demand Variability

The demand is variable. It depends on the season.

D.4 Scenario 4: Precision Agriculture-Argentina

D.4.1 Scenario description

Small farm operators, cooperatives, large agro-holdings, agricultural land investment trusts, logistics and supply chain operators need homogeneous and reliable means to manage their crops. Precision Agriculture provides support services for irrigation, based on the use of Earth Observation data, hydrological models and meteorological data. Using high resolution imagery, inadequate irrigation or practices can be identified quickly and other agricultural treatments can be more accurately assessed and optimized.

In Latin America the leading country is Argentina, where it was introduced in the middle 1990s with the support of the National Agricultural Technology Institute (Sensyf), (Astrium).

D.4.2 Response

The designed constellation will provide mosaics of Argentina in a daily basis, which shall be processed in order to offer several layers of precise information at the resolution of the satellite system to the users. Some processing shall offer info about irrigation planning, improved management of fertilizer usage, meteorological data affecting crops and fruit maturity.

D.4.3 Data distribution

- WFS to provide direct access to the derived products.
- Value-added services (REST/HTTP, WMS, Tiles) for the creation of web applications

and interactive solutions.

D.4.4 Area of Interest

The Area of Interest is Argentina, see Figure D.4.

Figure D.4: Map of Argentina (from `http://www.dk.co.uk/static/cs/uk/11/worldfactfile/countries/ar.html`)

D.4.5 Users

The user is the National Agricultural Technology Institute.

D.4.6 Service Type

The service type is high added value (push/hosting) (subscription services for agriculture treatments, irrigation planning and general monitoring... Hosting could be employed for users that do not have where storage the data).

D.4.7 Processing Level

The processing level is high (several layers of post processing shall be offered to the users employing mosaics of the country, i.e. the National Agricultural Technology Institute).

D.4.8 Storage Level

The storage level is high.

D.4.9 Communications Level

The communication level is not urgent.

D.4.10 Demand Variability

The demand variability is constant (depending on the season).

D.5 Scenario 5: Basemaps-Worldwide

D.5.1 Scenario description

Satellite operators, space agencies and mapping companies offer basemaps as part of their services and data product. Built from the satellite scenes, they provide seamless and cloud-free coverage of certain region surface (countries, continents and worldwide) with multiple zoom layers and high detail. They are distributed as data archives (FTP, direct download) or through OGC Web Services (WMS) and Tiles. These are some examples:

- NASA's Blue Marble: offers a year's worth of monthly composites at a spatial resolution of 500 meters. These monthly images reveal seasonal changes to the land surface: the green-up and dying-back of vegetation in temperate regions such as North America and Europe, dry and wet seasons in the tropics and advancing and retreating Northern Hemisphere snow cover. They are available for download as georreferenced raster files.

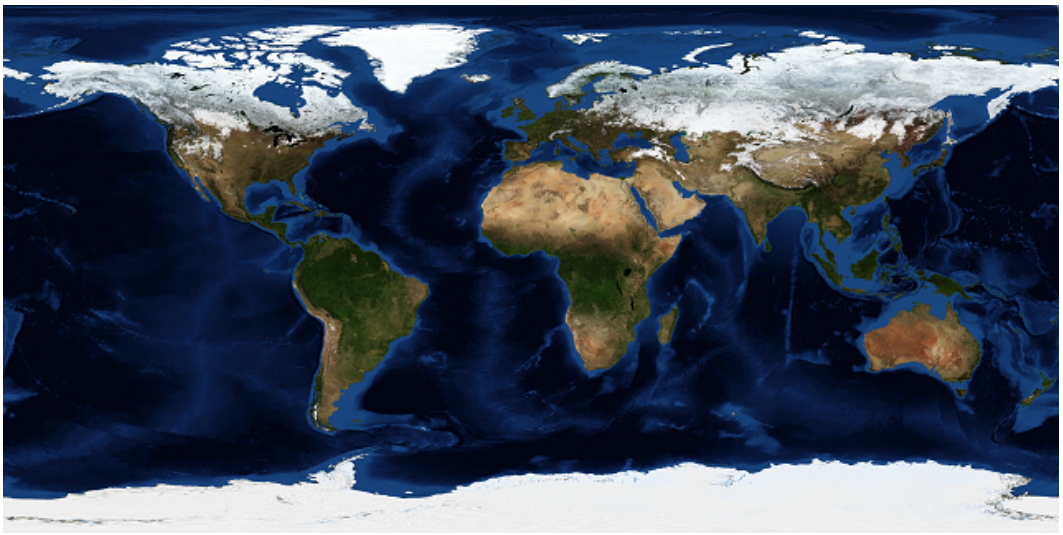


Figure D.5: NASA's Blue Marble from December with topography and bathymetry (from `http://visibleearth.nasa.gov/view.php?id=73909`

- SPOTMaps: From Astrium, provides $2.5\ m$, natural color, seamless ortho-mosaics derived from SPOT 5 data, providing nationwide and regional basemaps for a large

part of the globe. They are currently available over more than 110 countries, and more than 94 million km^2. The coverage is growing and updated on a monthly basis.

D.5.2 Response

With the global daily coverage of the GEO-Cloud constellation of satellites, a monthly true-color basemap can be built with very high detail and coverage percentage (cloud-free). In tropical lowlands, cloud cover during the rainy season can be so extensive that obtaining a cloud-free view of every pixel of the area for a given month may not be possible.

Processing will consist of creating monthly mosaics from the daily coverage of the constellation, removing clouds and applying color adjustments to create true-color and visually attractive aerial basemaps.

Resolution of the basemaps will depend on the storage and processing capacity, it will be studied further during the implementation of the scenario.

D.5.3 Data distribution

Data will be delivered though cached WMS and tiles.

D.5.4 Area of Interest

The area of interest is the whole world. Country or continent coverages can be represented. We will focus on Europe.

D.5.5 Users

- Governmental clients.
- Infrastructure managers.
- Third-party service providers.

D.5.6 Service Type

The service type is high added value (push/hosting), using on-line delivery mechanisms.

D.5.7 Processing Level

The processing level is high (creation of a monthly mosaic from a catalogue of daily images, cloud removal or color adjustments)

D.5.8 Storage Level

The storage level is very high. It can be modulated with the maximum zoom level/resolution offered and the area of interest.

D.5.9 Communications Level

The communication level is not urgent.

D.5.10 Demand Variability

The demand variability is variable. See Table 14 for a summary of the service characteristics.

Appendix E

Source code

Due to the extension of the source code (5.000 lines), it is was decided to be included in CR-ROM attached to this document.

The implementation of the GEO-Cloud experiment consists of the following steps:

- To create a *Space System Simulator*.
- To implement a *Virtual Wall* experiment and to use *JFed* to deploy it.
- To create the Orchestrator, A&C , Database and Processing Chain in *BonFIRE*.
- To create setup scripts to initialize the nodes in *BonFIRE* and *Virtual Wall* .

The root directory of all files is "GeoCloud". Inside it there are two folders: "doc" which contains the documentation and "source" which contain all the source code developed in this project.

E.1 Space System Simulator files

The *Space System Simulator* consists of a *Satellite System Simulator* and a *Ground Station System Simulator*. The *Satellite System Simulator* is constituted by 17 *Satellite Simulators* executing at the same time. Also the *Ground Station System Simulator* is constituted by 12 *Ground Station Simulators* executing at the same time. The source of the software is located in *source/vw/simulator* as follows:

- *satellite.py*: source code of the *Satellite Simulator*.
- *groundstation.py*: source code of the *Ground Station Simulator*.
- *runGS.sh*: Bash script that deploys and starts the *Ground Station Simulators*.
- *runSat.sh*: Bash script that deploys and starts the *Satellite Simulators*.
- *clean.sh*: Bash script that cleans all the log files and the temporally files.

E.2 Virtual Wall deployment files

The files used to deploy the *Space System Simulator* in *Virtual Wall* are located in *source/vw*. They are the following:

- *Description.rspec*: Specification in *Rspec* format in order to allow *JFed* to create the experiment. This file contains the nodes reservation and some commands to be executed after the node initialization.
- *install_ftp.sh*: Installation of the ftp server in the node.
- *pus_ip.sh*: Script that puts the IP database in a file in the node.

E.3 Orchestrator, Archive and Catalogue, Database and Processing Chain files

The *BonFIRE* is composed by the Orchestrator, A&C, database and Processing Chain modules. There are two implementations for the Cloud architecture: the first one implemented using SSH and SCP and the second one based in the ZeroC ICE distributed middleware. Each one of them it is located in a different location each other.

The implementation using SSH and SCP is located in *source/bonfire* as follows:

- *Orchestrator*: All the source code is located in *source/bonfire/orchestrator* as follows:
  - *listener.py*: listener class used by the *Orchestrator* component.
  - *main.py*: contains the main method used by the *Orchestrator* component.
  - *orchestrator.py*: contains the class Orchestrator used by the *Orchestrator* component.
  - *processingChain.py*: contains the *Processing Chain* component used by *Orchestrator*.
  - *Orchestrator.conf.xml*: contains the initial configuration for the *Orchestrator* in XML format.
  - *Load.py (Not used)*: Python class that obtains the CPU workload remotely.
- *A&C*: All the source code is located in *source/bonfire/geoserver* as follows:
  - *install.sh*: bash script that install the A&C. It must be executed in a BonFIRE node.
  - *catalog_pp.py*: this script is called by the *Processing Chain* in order to store and catalogue the processed image.
  - *Install_ftp.sh*: bash script that installs an ftp server on the node.
- *Product Processors*: The source code is located in *source/bonfire/processingChain* as follows:
  - *PPscript.sh*: bash script that is called by the Orchestrator in order to process an image.

The second implementation using *ZeroC ICE* is located in *source/ice/* as follows:

- *app*: this folder contains the distributed application for its deployment.

- *cfg*: it contains the configuration files for deploying the nodes by using the *IceGrid* service.
- *certs*: it contains the certificates for authenticating connections.
- *bin*: it contains the clean, start and stop scripts.
- *test*: this folder contains the developed test in order to check the funcionality of the components.
- *src*: this directory contains the source files for each component of the cloud.

The Database files are located in *source/database/modelDatabase* and *source/database/scenaries*.

- The directory *source/database/modelDatabase* contains the following files:
  - *dbmodel.mwb*: database model created by using *Mysql-Workbench*.
  - *dbmodel.mwb.bak*: backup of dbmodel.mwb.
  - *modelDataBase*: database model in SQL language.
  - *schema.png*: database schema picture.
  - *setup.sh*: bash script to install and create the database in a *BonFIRE* node.
  - *Autorun.sh (Not used)*: bash script to start the mysql demon with a customized configuration.
  - *my.cnf (Not used)*: MySQL service customized configuration.
- The directory *source/database/scenaries* contains the following files:
  - *All_Scenarios.csv*: it contains the AOI of all scenarios.
  - *Scenario_1_Emergencies_Lorca_Earthquake.csv*: it contains the visibility zones and the satellites that enters in them for the Scenario 1.
  - *Scenario_2_Infrastructure_monitoring.csv*: it contains the visibility zones and the satellites that enters in them for the Scenario 2.
  - *Scenario_3_South_West_England.csv*: contains the visibility zones and the satellites that enters in them for the Scenario 3.
  - *Scenario_4_Precision_Agriculture_Argentina.csv*: it contains the visibility zones and the satellites that enters in them for the sSenario 4.
  - *Scenario_5_Basemap_Worldwide.csv*: it contains the visibility zones and the satellites that enters in them for the Scenario 5.
  - *setDatabase.py*: script that initializes the database created by setup.sh.

E.4 PlanetLab experiment files

The source code of the *PlanetLab* experiment are located in *source/ple/* directory. This directory contains the following files:

- *iperfClients.py*: It contains the PlanetLab experiment script that obtains the client nodes bandwidth.
- *iperfClientsUDP.py*: It contains the PlanetLab experiment script that obtains the client nodes loss-rate.
- *pingClients.py*: It contains the PlanetLab experiment script that obtains the client nodes round trip time.
- *pPLE.py*: It contains the PlanetLab experiment script that obtains the ground station nodes bandwidth.
- *pPLEUDP.py*: It contains the PlanetLab experiment script that obtains the ground station nodes loss-rate.
- *pingPLE.py*: It contains the PlanetLab experiment script that obtains the ground station nodes round trip time.
- *bandwidth_plotting_nodes.py*: Script that plots the bandwidth nodes ordered by distance.
- *bandwidth_plotting.py*: Script that plots the ground stations nodes bandwidth.
- *bandwith_plotting_node.py*: Script that plots the nodes bandwidth.
- *bandwidth_plotting_per_ground.py*: Script that plots the ground stations nodes bandwidth into a histogram.
- *Loss-rate_plotting_nodes.py*: Script that plots the loss-rate per node.
- *Loss-rate_plotting_nodes_customers.py*: Script that plots the loss-rate per client node.
- **.out*: it contains results of executions
- *Results* directories*: it contains some execution results.

Bibliography

[Aas98] J. Aase. Netcps. Online, February 1998.

[ACG+10] A.I. Avetisyan, R. Campbell, I. Gupta, M. T. Heath, S. Y. Ko, G. R. Granger, M. A. Kozuch, D. O'Hallaron, M. Kunze, T. T. Kwan, K. Lai, M. Lyons, D. S. Milojicic, K. Yan Lee, Y. Chai Soh, N. Kwang, J. Luke, and H. Namgoong. Open cirrus: a global cloud computing testbed. *COMPUTER*, page 9, 2010.

[Amb10] M. Ambrust. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[Ath14] D. Athanasias. Gui programming in python. Online, March 2014.

[Bal13] H. Ball. *Satellite AIS for Dummies*. John Wiley & Sons, Inc, 2013.

[BCL+14] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[Bec14] J. Becedas. The geo-cloud experiment: Global earth observation system computed in cloud. In *The Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computin, IMIS*, pages 2–4, Birmingham, United Kingdom, July 2014.

[Bon14] BonFIRE. Bonfire project, 2014.

[Bra13] R. A. Braeunig. Ororbit mechanics. Online, 2013.

[Com] Ruby Community. Ruby. Online.

[Com12] Hewlett-Packard Company. Care and feeding of netperf 2.6.x. Online, 2012.

[Com14] The GeoNetwork Community. Geonetwork opensource. online, January 2014.

[Coo14] M. Cooper. Advanced bash-scripting guide. online, March 2014.

[Der07] L. Deren. Remote sensing can help monitoring and predication natural disasters. *Science & Technology Review*, 25(6):3, 2007.

[ESA09] ESA. History of earth observation. Online, November 2009.

[Eur10] Euroconsult. *Satellite-based Earth Observation, Market Prospects to 2019*. Euroconsult, 2010.

[Eur14] PlanetLab Europe. Planetlab europe project, 2014.

[Fed14a] Fed4FIRE. Fed4fire project, 2014.

[Fed14b] Fed4FIRE. Fed4fire: Tools, 2014.

[Fou08] The Apache Software Foundation. Hadoop documentation. Online, October 2008.

[Fou14a] Python Software Foundation. Unittest - unit testing framework. Online, August 2014.

[Fou14b] The Apache Software Foundation. Apache tomcat 7.documentation index. Online, July 2014.

[Geo14] Geoserver. Geoserver, March 2014.

[GGV+13] J. García, F.O. García, Pelechano V., Vallecillo A., Vara J.M, and Vicente-Chicote C. *Desarrollo de software dirigido por modelos: Conceptos, métodos y herramientas*. RA-MA Editorial, 2013.

[GPB+14] G. González, R. Pérez, J. Becedas, M. Latorre, and F. Pedrera. Measurement and modelling of planetlab network impairments for fed4fire's geo-cloud experiment. In *26th International Teletraffic Congress - Workshop on Future Internet and Distributed Clouds (FIDC 2014)*, Karlskrona, Sweden, September 2014.

[Gro10] S. Deckelmann. PostgreSQL Global Development Group. Which databdata solve my problem? a survey of open source databases, 2010.

[GW09] X. Ge and H. Wang. Cloud-based service for big spatial data technology in emergency management. In *Proceedings of the ISPRS (ISPRS'09)*, pages 126–129, 2009.

[HAHB+12] A. C. Hume, Y. Al-Hazmi, B. Belter, K. Campowsky, L. M. Carril, G. Carrozzo, and G. Van Seghbroeck. *BonFIRE: A Multi-cloud Test Facility for Internet of Services Experimentation. In Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 81–96. Springer Berlin Heidelberg, 2012.

[INR14] INRIA. Nepi, 2014.

[Ipe14] Iperf. Iperf, 2014.

[Lau98] S. Laurent. Why xml? Online, 1998.

[MG11] P. Mell and T. Grance. The nist definition of cloud computing. *Recommendations of the National Institute of Standards and Technology*, page 3, 2011.

[Mic13a] Microsoft. Ntttcp version 5.28 now available, July 2013.

[Mic13b] Sun Microsystems. Uperf. a network performance tool. Online, December 2013.

[NAS] NASA. Remote sensing: Remote sensing methods. Online.

[NSI] NSIDC. Remote sensing: Active microwave. Online.

[OPS14] T. Oetiker, H. Partl, and E. Schlegl. The not so short introduction to latex2ε. 2014.

[Org] JSON Organization. Json: The fat-free alternative to xml. Online.

[Org14] Lua Organization. Lua 5.2 reference manual. Online, January 2014.

[PCVB13] C. Pelsser, L. Cittadini, S. Vissicchio, and R. Bush. From paris to tokyo: On the suitability of ping to measure latency. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 427–432, 2013.

[PGB+14] R. Pérez, G. González, J. Becedas, M. Latorre, and F. Pedrera. Testest cloud computing for massive space data processing, storage and distribution with open-source geo-software. In *FOSS4G-Europe Congress - Workshop on Free and Open Source Software for Geospatial, INSPIRE and Big Data (FOSS4G-Europe 2014)*, Bremen, Germany, July 2014.

[Pyt14] Python Software Foundation. Python v2.7.6 documentation, March 2014.

[San09] R. Sandau. Satellite earth observation and surveillance payloads. In NATO, editor, *Systems Concepts and Integration Panel*, page 16p, April 2009.

[Ste14] Dr. D. Stern. Kepler's three laws of planetary motion. Online, April 2014.

[Tan03] A. S. Tanenbaum. *Redes de Computadoras*. Editorial Alhambra S.A. (SP), 2003.

[Tea13] Paramiko Team. Paramiko - ssh2 protocol for python. Online, April 2013.

[TvS08] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall International, 2nd rev. ed. edition, 2008.

[Wal14] Virtual Wall. Virtual wall facility, 2014.

[Zer13] ZeroC. Ice manual. Online, 2013.

This document was edited and typed with LaTeX
by using the **arco-pfc** template whose is available in the following address:
`https://bitbucket.org/arco_group/arco-pfc`