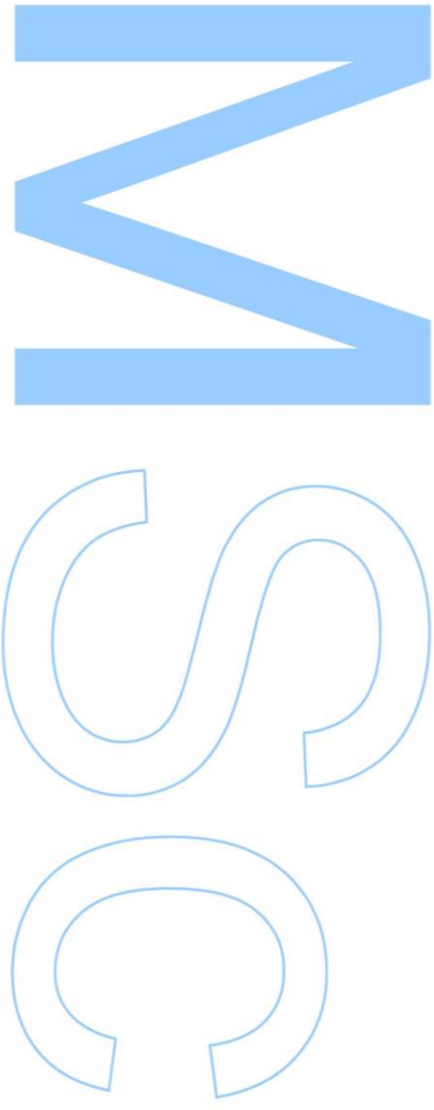
Automatic Acquisition of Seismological and Earth's Magnetic Field Data from IGUP

Rui Pedro Carvalho Coelho
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2019

Orientador
Sérgio Crisóstomo, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto, Investigador Instituto de Telecomunicações

Coorientador
Luís Lopes, Professor Associado, Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____

Abstract

Earth observation from the point of view of various variables is of great importance nowadays. The Faculdade de Ciências da Universidade do Porto (FCUP) has maintained equipment of this nature for several decades at the Instituto Geofísico da Universidade do Porto (IGUP) and has come the opportunity to modernize the acquisition infrastructure and the transfer of the old dataset. Throughout all these years of data gathering, there is a large data set of analog and digital data. Nowadays, data is only recorded in the digital format but it needs human intervention in order to store that data. With the current solution, data is not easily visualized by the community so it is important to have access to that data online to make life easier for the person interested in viewing the data.

This dissertation will describe our work at Instituto Geofísico da Universidade do Porto (IGUP) in order to automate seismic and earth's magnetic field data acquisition, from a seismograph and a magnetometer, respectively. In the course of this work, we developed a software that allows not only the automatic acquisition of data from the institute's instruments but also its storage in the cloud and its visualization, exportation and further analysis through a visualization tool based on a Web service. Historical records in digital format have also been processed and entered into the database providing users with access to a dataset with decades of information.

Resumo

A observação da Terra do ponto de vista de várias variáveis é de grande importância hoje em dia. A Faculdade de Ciências da Universidade do Porto (FCUP) mantém equipamentos dessa natureza há várias décadas no Instituto Geofísico da Universidade do Porto (IGUP), e surgiu a oportunidade de modernizar a infraestrutura de aquisição e a transferência do antigo conjunto de dados. Ao longo de todos estes anos de recolha de dados foi armazenado um grande *dataset* tanto em formato analógico como em formato digital. Atualmente, os dados são apenas recolhidos em formato digital, mas é necessária a intervenção humana para os dados serem guardados. Com a solução atual, os dados não são facilmente visualizados pela comunidade, portanto, é importante ter acesso a esses dados on-line para facilitar a vida da pessoa interessada em visualizá-los.

Esta dissertação irá relatar o nosso trabalho no Instituto Geofísico da Universidade do Porto (IGUP) na automatização da aquisição de dados sísmicos e do campo magnético da terra obtidos a partir de um sismógrafo e de um magnetómetro, respetivamente. No decorrer deste trabalho, desenvolvemos um *software* que permite não apenas a aquisição automática de dados dos instrumentos do instituto, mas também o armazenamento na *cloud* e a sua visualização, exportação e posterior análise através de uma ferramenta de visualização baseada num serviço *web*. Os registos históricos em formato digital também foram processados e inseridos na base de dados, fornecendo aos usuários acesso a um conjunto de dados com décadas de informação.

Acknowledgements

I would like to thank my dissertation advisors, Prof. Sérgio Crisóstomo, and Prof. Luís Lopes for all the patience, guidance, support, and for all the time they spent with me. Also, I would like to thank IGUP for this opportunity, more specifically to Prof. Helena Sant'ovaia, Prof. Rui Moura and to Ana Moreira for their support and availability to anything that was needed.

Most importantly I would like to thank my family, especially my parents and my sister, for their patience, support and motivation throughout not only this dissertation but throughout all my years of study, always providing everything that I needed and giving me continuous encouragement. Mostly, I am what I am today because of you!

To my grandparents that are not with us anymore, hope I have made them proud!

I would like to especially thank my girlfriend, Ana Amaral, for all the support, motivation, strength, trust, and encouragement in the most exhausting and stressful moments. For your availability in every moment and for all the values and advice that you gave me, thank you!

Finally, I would like to thank the Faculdade de Ciências da Universidade do Porto, especially to Departamento de Ciências de Computadores for these last five years. To all my friends and to CC & Redes for all the support, encouragement and for all the good moments, thank you!

To my parents, sister, grandparents, and girlfriend...
Hope you are proud!

Contents

List of Tables

List of Figures

Listings

Acronyms

ACK Acknowledgement

ADC Analog to Digical Coverter

CSV Comma-Separated Values

DB Database

DBMS Database Management System

DCC Departamento de Ciência de Computadores

DLL Dynamic Link Library

FCUP Faculdade de Ciências da Universidade do Porto

HTML Hypertext Markup Language

IBM International Business Machines

ID Identity

IGUP Instituto Geofísico da Universidade do Porto

IoT Internet of Things

IP Internet Protocol

JSON JavaScript Object Notation

LP Long Period

MQTT Message Queuing Telemetry Transport

NoSQL Not only SQL

PC Personal Computer

QoS Quality of Service

SMTP Simple Mail Transfer Protocol

SP Short Period

SQL Structured Query Language

TSDB Time Series Database

UP Universidade do Porto

WSN Wireless Sensor Network

WWSSN World-Wide Standardized Seismograph Network

Chapter 1

Introduction

Humans have always been very curious beings who likes to know, understand, and study everything around them. Since the Earth is our home, we have the need to explore it and take advantage of all the knowledge we can in order to make our life better. For example, we rely on earth to obtain resources; we used the earth's magnetic field to navigate, etc.
In all these years, earthquakes always happened, and people studied this phenomenon, emerging the term seismology that is the science that studies earthquakes and the propagation of elastic waves through Earth.

In a world where the information is disseminated quickly and efficiently, events like earthquakes, when they happen, are diffused all around the world by journalists that need to have access to all the information available about the occurrence in the simpler and faster way. On the other hand, data scientists need to have access to the information in the same way.
Nowadays, we live in an era where everything tends to be digital and automated. We live in an era where data is so precious that sometimes it is more valuable than money, then the loss of information can be a catastrophe. Back in the days, geophysics information was stored in paper and books, a method that is highly fallible due to the dangers inherent in this type of data support like fire, floods, humidity. Because of this high risk of data loss, we began to store data in a digital support. It is still fallible if we do not take the necessary measures, but, in general, it is safer than analog data.

This project will occur in parallel with two other projects. One of them is the exploratory analysis of meteorological data from IGUP, and the other is the automatic data acquisition of the meteorological station of IGUP. Because of this, this project will have some common parts, so some decisions will be taken with this in mind.
Also, this project will focus its implementation in Instituto Geofísico da Universidade do Porto (IGUP) (Figure 1.1). IGUP was founded in 1885, is located in Serra do Pilar in Vila Nova de Gaia and has a privileged view over the city of Porto and Douro river. Being one of the first weather stations in Portugal, it is a historic and important organization in scientific and technological culture in the city of Porto. In 1993 IGUP worked as a seismic station and was included in a global network called Worldwide Standardized Seismograph Network (WWSSN), being part of a

network of 125 stations installed by WWSSN over the world to cover all seismic events around the globe. This organization was created mostly because of the cold war, to monitor nuclear explosions to gain some advantage in war [Moura et al., 2014].

Figure 1.1: Instituto Geofísico da Universidade do Porto (IGUP).

1.1 Problem Statement

Even though the years passed and IGUP upgraded the equipment, it still has problems. The data is stored in a computer, and the only way to obtain that data is to go there physically and transfer it. This also has an inherent problem, the possibility of data loss, because data is stored in a single physical PC and, in case of damage, data can be lost. On the other hand, there is a necessity of making this data available to the world, so we would need to make an automatic acquisition of the data, so there is no need for someone to go there physically to transfer it all. After that, it would be stored and then made available online in order to interested people analyze it and even download the data in some type of format.

1.2 Proposal

To solve the problems defined in Section 1.1, we will need to develop and work with some tools in order to achieve our goal, and for that, we propose to:

- Create and implement a database for geophysical data: to store all the data obtained by the sensors we will need to create a model to save that data in order to have access to it in

the web;

- Implementation of a server that will make interface between the sensors (seismograph and magnetometer) and the database: to collect and record all the data we need to communicate with the sensors to obtain the data and then organize it to store in the database. For that purpose, we will implement a server that will take care of this process and all transactions;
- Implementation of a web interface to visualize the data available in the database: to visualize the data we will create a web interface that will make queries to the database asking for the information that the user insert as input;
- Insertion of the old geophysical data in the database;
- Control and monitoring of data to notify specialists about peculiar values: with this feature, we want to be capable of notifying specialists about certain events like earthquakes and errors in the readings, for example;
- Data storage not only in a physical database but also in the cloud: to keep the data safe, we will save the database in the cloud to prevent from data loss and disasters.

1.3 Outline

After this introduction, which has a brief context, motivation, and objectives, we follow onto Chapter 2 where an introduction to seismology and earth's magnetic field is made, as well as the description of the equipment available in IGUP, technologies inherent to this project and the study of databases. Thereafter, we have Chapter 3 where we have the architecture of the solution, providing an overview of the system architecture and going into detail about each component. Later, in Chapter 4, we go into detail about the implementation of the system. Next, we describe how we processed the historic data. Finally, we have the final Chapter where we conclude having an overall thought about this project as well as some thoughts about future developments of this project.

Chapter 2

State-of-the-art

In this Chapter, for a better understanding of this article, we will introduce some basic concepts about the theory inherent to this project. Also, we describe the equipment available at IGUP and some technologies implicit to our work.

2.1 Concepts

In this Section, we introduce some base concepts about seismology and earth's magnetic field that are important for the rest of the article.

2.1.1 Seismology

An earthquake is the sudden shaking of the surface of the earth as a result of a release of energy in the earth's lithosphere creating seismic waves that cause the ground to shake.
Generally, the word earthquake is used to describe seismic events that have natural or human causes. Earthquakes are caused mostly by the collision of tectonic plates, but also by landslides, volcanic activity, and nuclear tests.
To measure an earthquake, there exist two scales: intensity and magnitude. The seismic intensity scales are based on observations like damages in structures, in the natural landscape or the level of alarm that people and animals got. The seismic magnitude scales measure the seismic waves recorded by the seismograph. Magnitude scales vary on the type and component of seismic waves measured and in the calculations used. Typically, the media reports earthquakes strength by the Richter scale (magnitude scale).

Seismic waves are vibratory movements of rock particles caused by a sudden release of energy in the form of waves that travel through the earth layers. Those waves are recorded by a seismometer and then are studied by a seismologist. There are distinct types of seismic waves with different behavior and velocity. This phenomenon helps seismologists find the hypocentre (or focus) of the seismic event which points to where it was its origin. After that, we also infer

what was the epicentre, which is the point in the earth's surface directly above the hypocentre (Figure 2.1).

There are two types of waves: Body and Surface waves [Shearer, 2009].

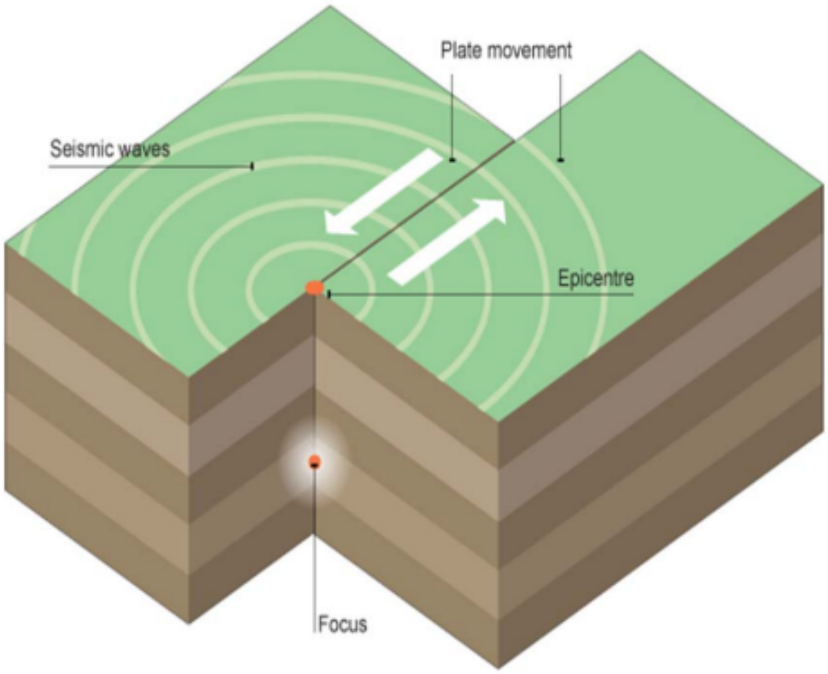


Figure 2.1: Illustration of an earthquake [Boppidi, 2014].

Body waves travel through the interior of the earth. These are divided also into two types: Primary waves (P-waves) and Secondary waves (S-waves). Primary waves are compressional waves and have a longitudinal movement in the particles. They are called primary because they are the fastest waves and are the first reaching the seismographic stations. Secondary waves are shear waves that have a transverse movement in the particles and, as the name refers, they arrive after the P-waves. We can see the behavior of both of these waves in Figure 2.2.

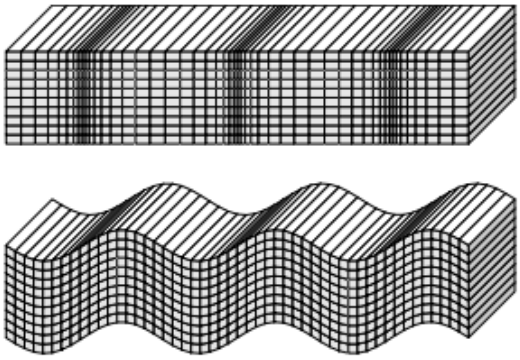


Figure 2.2: P-wave (top) and S-wave (bottom). S-wave propagation is pure shear with no volume change, whereas P waves involve both a volume change and shearing (change in shape) in the material [Shearer, 2009].

Surface waves travel along the earth's surface. Like the Body waves, these are divided into two types: Rayleigh waves and Love waves. Rayleigh waves are acoustic waves that travel along in solids. Love waves are Horizontally shear waves that are faster than the Rayleigh waves. We can see the behavior of both of these waves in Figure 2.3.

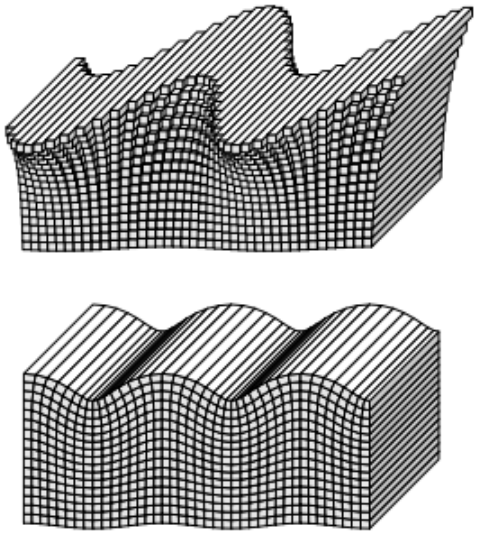


Figure 2.3: Love wave (top) and Rayleigh wave (bottom). Love waves are purely transverse motion, whereas Rayleigh waves contain both vertical and radial motion [Shearer, 2009].

2.1.2 Earth's Magnetic field

A magnetic field is a physical phenomenon created by magnetized materials or an electrical current. More specifically, the earth's magnetic field is created by electrical currents generated in the core of the earth. Because of its composition being mostly iron, the metallic liquid core has convection currents that create electrical currents, thus creating a magnetic field. This theory

is called geodynamo hypothesis because of its similarity with a dynamo [Kono and Roberts, 2002].

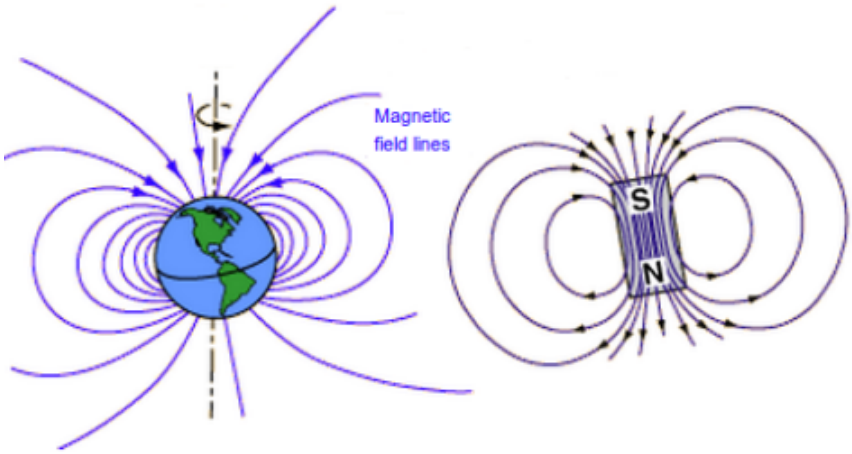


Figure 2.4: Earth magnetic field (left) and a dipole magnetic field (right) [Gunnarsdóttir et al., 2012].

Earth's magnetic field influences our daily life in a variety of ways. Although we know that magnetic fields do exist, its study is indirect because it is not a phenomenon that we feel, like an earthquake. It is more like the gravity field, we do not have any way of sensing it, we sense its influence by pulling us down. The magnetic field it's the same, we do not have any way of feeling it, but we do know that he protects the earth from harmful radiation and its useful for navigation (e.g., compass).

There are two basic magnetic quantities:

- **Magnetic induction, B:** it is a measure of the strength of a magnetic field at a given point. Its common unit is Tesla (T);
- **Magnetic field strength, H:** it measures the ability of an electric current to produce a magnetic field at a given point. Its common unit is Ampere/meter (A/m).

Magnetic field sensors are called magnetometers and typically, measure magnetic induction [Reeve, 2010].

2.2 Technology

In this Section, we present a description about seismographs, magnetometers, and seismic data formats as well as some information about the equipment and software that is currently used in IGUP. Also, we describe some technologies inherent to this project.

2.2.1 IGUP Seismograph

In order to capture seismic waves, we use a seismograph. We can think of a seismograph like a pendulum that bounces according to earth-shaking caused by the waves. In the past, the signal was recorded in an analog format. Seismograms were produced using ink or were produced using beams of light in photographic paper. Nowadays, that method became obsolete, because of the threats that analog data has, so actually, the data is recorded in a digital format.

In IGUP the seismograph (Figure 2.5) is composed by six seismometers that equals to two seismographs each with three components. One of them is the long period seismograph (LP) of the Ewing-Press type and the Sprengnether brand, sensitive to lower frequencies, hence recording distant seismic events. The other is a short period seismograph (SP) of the Benioff type and the Geotech brand, sensitive to high frequencies, hence recording closer seismic events. Both of these seismographs are equipped with two horizontal sensors (East-West and North-South) and one vertical sensor (Z). The acquisition unit had a precise quartz watch, a radio receiver (T.S.F.) for the reception of time signals, a control of calibration, and a set of batteries in case of energy failure [Moura et al., 2014]. As was previously mentioned, the data was recorded in photographic paper using very sensitive galvanometers. This recorded analog data was read like books, top to bottom and from left to right with a scale of one hour for LP data and fifteen minutes for SP data.

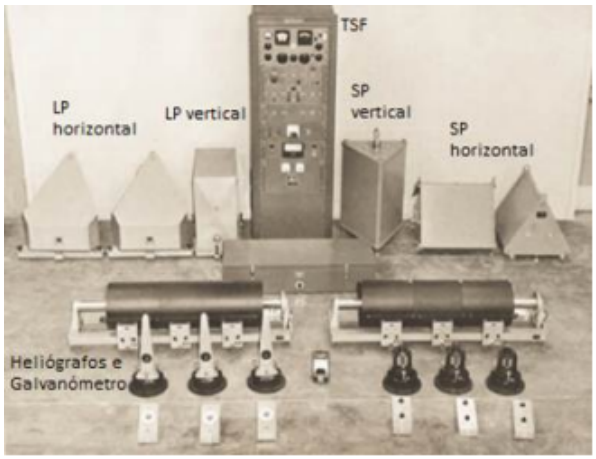


Figure 2.5: IGUP seismograph [Moura et al., 2014].

Nowadays, like in almost every place, IGUP adopted the digital data solution. It still uses the original seismic sensors to record data, but instead of using photographic paper to record data, it was implemented a signal conversion unity to convert from analog to digital using amplifiers of low noise and 16-bit analog to digital converters (ADCs). The digital data is transmitted to a computer using the serial port and a USB adapter being then visualized and stored using a software called WinSDR [Moura et al., 2014].

2.2.2 WinSDR

WinSDR [Cochrane, 2019c] (Figure 2.6) is a Windows and Linux based data logger for the PSN-ADC-SERIAL/PSN-ADC-USB 16-Bit Analog-to-Digital Converter boards. This software is used to configure the seismograph and to capture seismic data being then visualized using WinQuake [Cochrane, 2019b] that is a software developed by the creators of WinSDR to visualize seismic data [Moura et al., 2014].

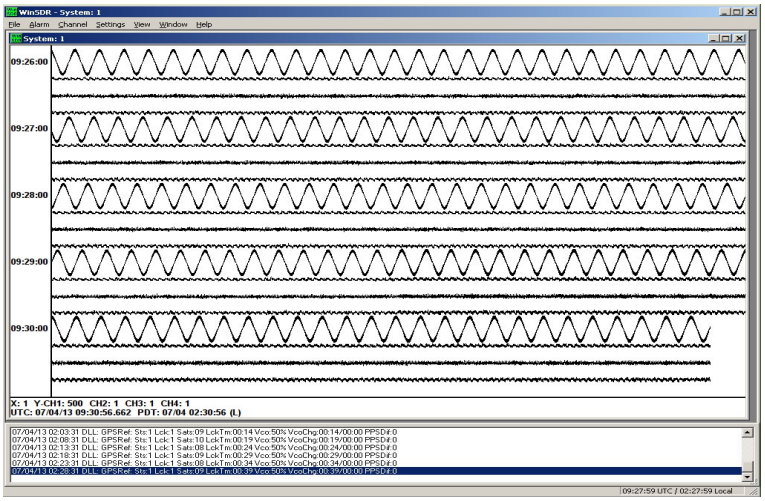


Figure 2.6: WinSDR interface. Adapted from [Cochrane, 2019c].

2.2.3 Seismic Data Formats

In the early days of seismographs, data was represented, and stored in paper which has a high risk of loss. Since seismology depends on cooperation and exchange of information, data should be shared between international data centres, seismic stations, etc. This was difficult because of the uniqueness of paper seismograms (susceptible to damage or loss) and the high cost of making copies of them [Dost et al., 2012]. Later, the problem of the fragility of paper seismograms was solved by storing data in a digital format. However, this evolution had another problem since each seismograph manufacturer had his own file format. This is bad because of the reliance on seismological data for the seismic study. If different seismic stations had their own format, it would interfere with the process of sharing and analysis of the data, because it would be mandatory to know the structure of the file and to parse it accordingly. We can see here a necessity of having compatible data. With this in mind, some attempts were made to create a standard format for seismic data [Raykova and Nikolova, 2000]. There are a large number of parameters in seismology, like coordinates (longitude, latitude, and depth), phase name, hypocentre, arrival time, etc. and some formats only include some of them [Dost et al., 2012]. In this case, we will only work with digital waveform data. Waveform data is the raw data captured by the sensors that later can be represented as a graphic, displaying traces with the x-axis being

time and the y-axis being the sensor reading (Figure 2.7).

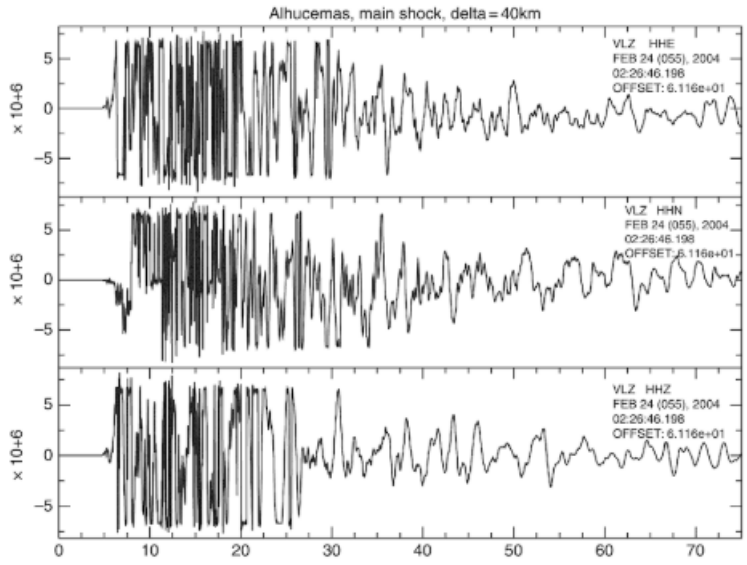


Figure 2.7: Waveform data represented in a graphic called seismogram. Each graphic represents one of the three components: North-South (HHN), East-West (HHE) and a vertical (HHZ) component [Udias and Buforn, 2017].

Nowadays, many different types of data formats are used. According to [Dost et al., 2012] and [Raykova and Nikolova, 2000] we can group most of the formats into one of the following formats:

- Local formats i.e. format that is used in a particular seismic station or network (e.g., ESSTF, PDR-2, BDSN, GDSN; mini-SEED);
- Formats used in standard analysis software (e.g., SEISAN, SAC, AH, BDSN; mini-SEED);
- Formats designed for data exchange and archiving (SEED, GSE);
- Formats used in database systems (CSS, SUDS);
- Formats for real-time data transmission (IDC/IMS, Earthworm; mini-SEED).

2.2.4 IGUP Magnetometer

Magnetic field sensors have been used for a long time. One of its first uses was on navigation over the oceans. Nowadays, they are still used for navigation but have a lot more uses. This industry has vastly expanded, and magnetic field sensors detect the presence, strength, or direction of magnetic fields.

According to [Caruso et al., 1998] we can divide magnetometers into low, medium or high field sensing.

- **Low field sensing:** Magnetometers that sense magnetic fields less than 1 microgauss (1×10^{-10} Tesla) and are used typically for medical applications and military surveillance. Some examples of low field sensing magnetometers are SQUID (Superconducting Quantum Interference Device) and Search-Coil.
- **Medium field sensing or earth's magnetic field sensors:** Magnetometers that sense the magnetic field with a range of 1 microgauss (1×10^{-10} Tesla) to 10 gauss (10×10^{-4} Tesla). These are a good option to measure the earth's magnetic field. Some examples of medium field sensing magnetometers are Fluxgate and Magnetoinductive.
- **High field sensing or bias magnet field sensors:** Magnetometers that sense the magnetic field above 10 gauss (10×10^{-4} Tesla). Most of the industrial magnetometers belong to this type of sensor as they detect fields way larger than the earth's magnetic field and are not affected by it. Some examples of high field sensing magnetometers are Reed switches.

IGUP magnetometer is a medium field sensor, more specifically a Fluxgate magnetometer.

Fluxgate magnetometers measure the intensity and direction of a magnetic field. It is the most common magnetometer for navigation systems. They were developed around 1928 and were important in military use to detect submarines. They have also been used for geological prospect and airborne magnetic field mapping [Caruso et al., 1998].

2.2.5 Simple Aurora Monitor (SAM)

Simple Aurora Monitor (SAM) [Langenbach and Hansky, 2019] is a magnetometer system that is composed by a fluxgate magnetometer (Figure 2.8) and a software (Figure 2.9) to monitor and configure the device being this the magnetometer system currently used by IGUP.

Figure 2.8: SAM Magnetometer that is used at IGUP. This is only one of three sensors i.e. it represents only one component of measurement.

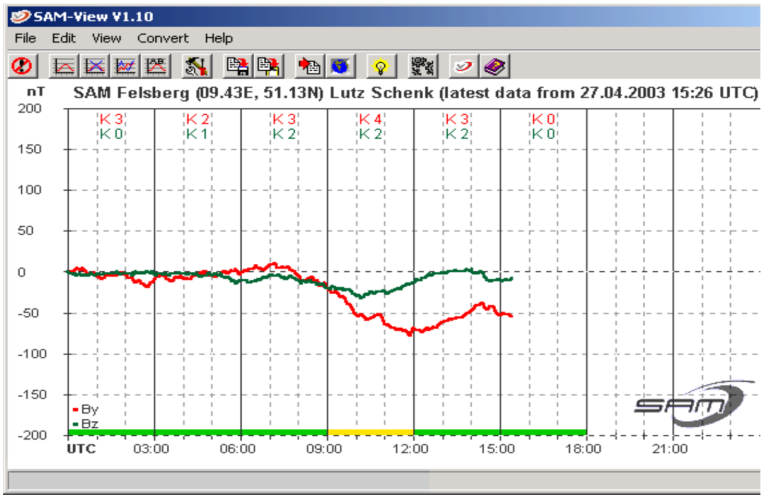


Figure 2.9: SAM View is used to visualize data recorded by the magnetometer [Langenbach and Hansky, 2019].

2.2.6 PSNADBoard.DLL

The PSNADBoard.DLL is a Dynamic Link Library used to interface the user's application and the PSN-ADC-SERIAL/ PSN-ADC-USB 16-Bit 8 Channel Analog-to-Digital Converter board. Since it is needed the software WinSDR in order to communicate with the board that makes interface with the seismograph, there was a need of allowing people to develop their own software and, because of that, this DLL was created to allow users to write their own software and be capable to configure and receive data from the ADC board. [Cochrane, 2019a]

2.2.7 ObsPy

Since Python has a large collection of scientific open-source modules, ObsPy [ObsPy, 2017] is an extension that aggregates several modules in order to make a powerful tool for seismologists.

ObsPy is an open-source Python library that provides tools for processing seismological data. It is intended to make the development of seismological software easier, since it can read and write several seismological file formats, convert them into one another, has toolkits to work with waveform data (time series data) and it also offers signal processing routines [Krischer et al., 2015].

2.2.8 Message Queuing Telemetry Transport (MQTT)

MQTT, also known as, Message Queuing Telemetry Transport is an application layer protocol that is based on a publish/subscribe architecture. It is a lightweight open messaging protocol designed by IBM, that works on top of TCP/IP protocol and originally it was designed to be used in unreliable networks with low resources and constrained devices [Lampkin et al., 2012].

This protocol is composed of a Broker and two kinds of clients: a publisher and a subscriber. Also, MQTT enables us to have a one-to-one, one-to-many and many-to-many communication (Figure 2.10).

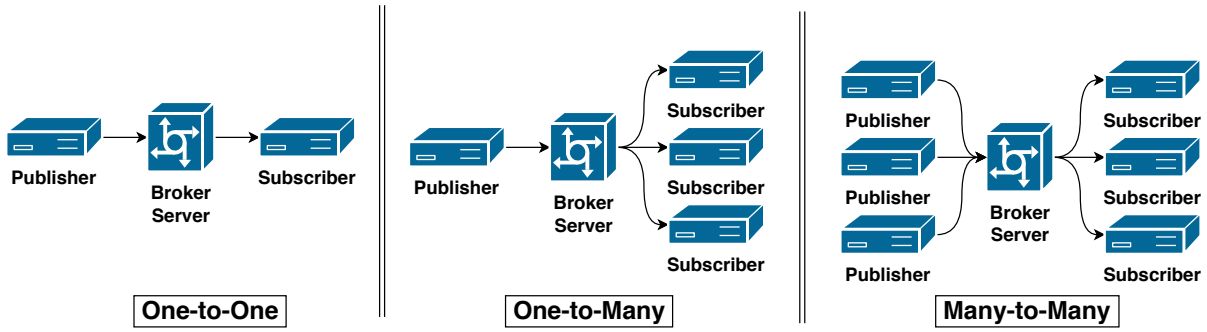


Figure 2.10: MQTT architeture. Adapted from [Torres et al., 2020].

A publish/subscribe system is a messaging pattern that basically sends data to anyone that shows interest in some type of available information. The act of showing and registering interest is called subscription, so the interested client is the subscriber. In the other half, there is the component that provides data, and this component does that by publishing the data. This component is called the publisher. Finally, the component that is responsible for the delivery of the data from the publisher to the interested subscriber is called broker. There are three principal types of publish/subscribe systems: content-based, type-based, and topic-based [Hunkeler et al., 2008].

- Content-based system: The subscriber describes what he wants to receive in each message. For example, he can choose the wind readings greater than a certain defined threshold, working almost like a query to a database;
- Type-based system: The subscriber says what type of data he wants to receive;
- Topic-based system: This system uses topics in which messages are published and in which clients subscribe to receive the data inherent to that topic. The list of topics is usually known in order to make a subscription.

In the case of MQTT, it uses a topic-based system in his publish/subscribe paradigm. Topics can be defined as a path, as in a file system, to maintain a hierarchy if necessary, and each level is separated by a slash ("/") [Manandhar, 2017]. For example, in a network of meteorological stations, we can have topics defined as follows in Listing 2.1.

```
Meteorology/Meteorological_Station_ID/Sensor/Sensor_ID/Temperature
Meteorology/Meteorological_Station_ID/Sensor/Sensor_ID/Wind_Speed
```

Listing 2.1: Example of a general definition of topics in the case of a meteorological stations network.

With the general configuration of the topics defined in Listing 2.1, we can specify some examples of possible topics by replacing Meteorological_Station_ID and Sensor_ID by actual IDs. So we would have something like in Listing 2.2.

```
Meteorology/1765/Sensor/17/Temperature
Meteorology/1765/Sensor/17/Wind_Speed
Meteorology/2574/Sensor/27/Temperature
Meteorology/2574/Sensor/27/Wind_Speed
Meteorology/7777/Sensor/21/Temperature
Meteorology/7777/Sensor/21/Wind_Speed
```

Listing 2.2: Specific example of topics in the case of a meteorological stations network.

Any publisher or subscriber could use these topics shown in Listing 2.2 to publish about and to subscribe to, respectively. In the case of the subscriber, it would be troublesome to hard-code every subscription. In this example, they are only six topics, but in a real meteorological station network it could be a significantly higher number, so MQTT has a solution for this that is called wildcards [Manandhar, 2017]. These wildcards can only be used for subscriptions. For example, if we want to subscribe to all the sensors of a certain meteorological station you use the character "#" as shown in Listing 2.3. The '#' is a multi-level wildcard and will match any topic that begins with the previous topic levels. This wildcard can only be used in the end of the topic.

```
Meteorology/1765/Sensor/#
```

Listing 2.3: Use case of the "#" wildcard in a topic.

Another possible thing to do with wildcards is, for example, to get all of the temperature sensors in a given meteorological station we can use the character "+" as shown in Listing 2.4. This wildcard will replace one topic level, and this will lead to a match in every possible topic at that level.

```
Meteorology/1765/Sensor/+/Temperature
```

Listing 2.4: Use case of the "+" wildcard in a topic.

In summary, the MQTT protocol works like this: the publisher publishes data on a certain topic and sends it to the broker. This broker forwards the messages received from the publishers to the subscriber clients that subscribed to that topic. So the broker just acts as an intermediary between the publisher and the subscriber, and both of the clients do not know each other, they only know the broker. In this protocol, the clients are designed to have a simple implementation being all the work done in the broker where all the complexity of the process resides. This makes the process of adding new publishers and subscribers simple, being one of the advantages of this protocol, the scalability [Bellavista and Zanni, 2016]. In Figure 2.11, shows the MQTT message transmission process.

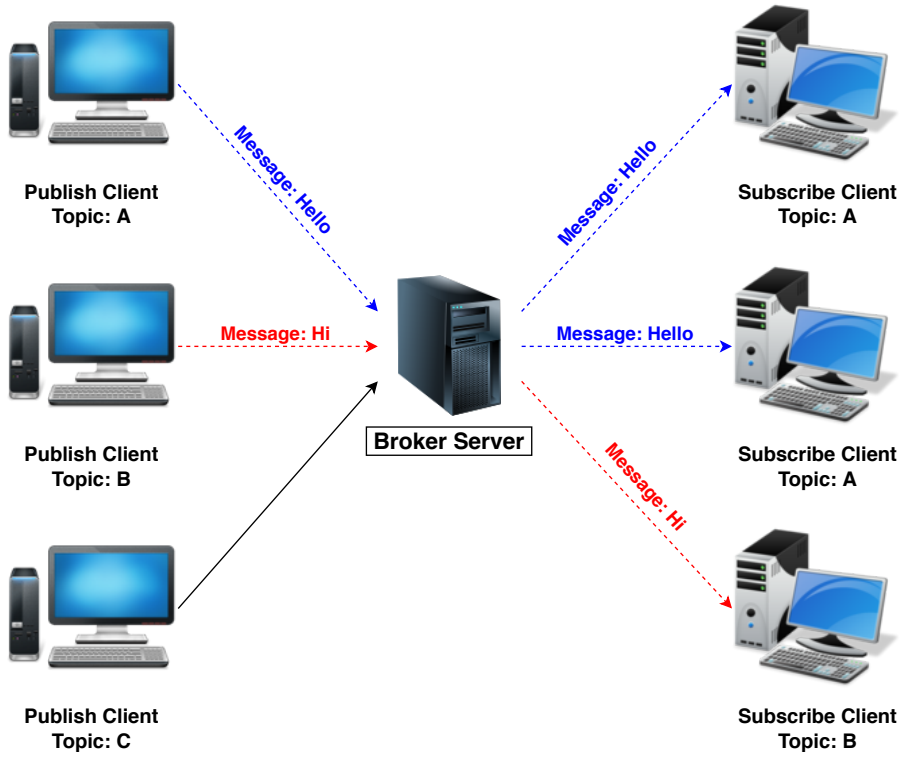


Figure 2.11: MQTT message transmission process. Adapted from [Lee et al., 2013].

As referred, this protocol was designed for unreliable networks, so it has a mechanism of messaging reliability called Quality of Service (QoS). This mechanism has 3 levels: 0, 1, and 2 [Lee et al., 2013].

- Level 0 is a best-effort delivery system, in other words, it sends the message once, and there is no acknowledgment of the arrival of the message. This means that it is possible that the message is not delivered;
- Level 1 ensures that the message is delivered at least once since there is an acknowledgment of the arrival by the receiver. However, if there is a problem with the arrival of the acknowledgment the message is retransmitted and causes a duplicate;
- Level 2 is the safest, but it is the slowest. This level ensures the arrival of the message exactly once by using a four-way handshake to acknowledge the arrival.

Figure 2.12, illustrates this QoS process.

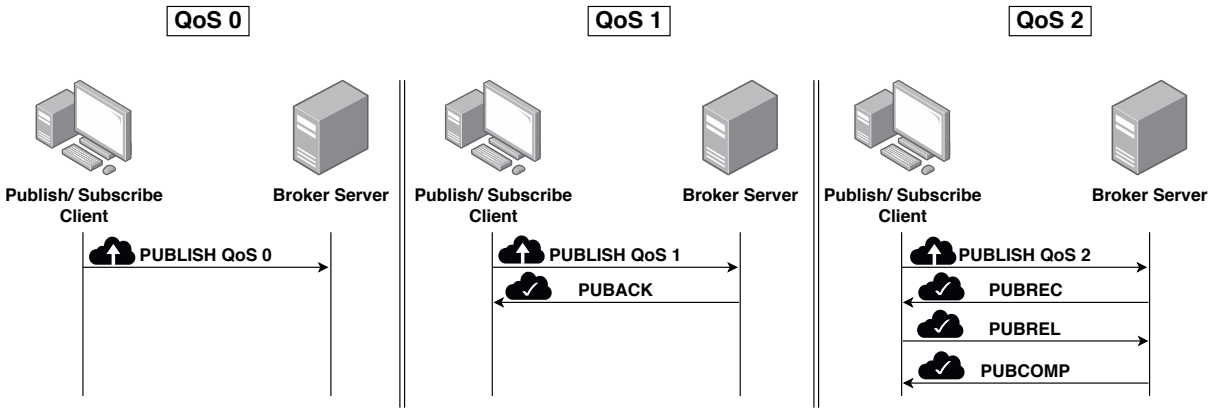


Figure 2.12: QoS message flow [Lee et al., 2013].

Besides QoS, there are several features like retained messages, clean session, and last wills.

Retained messages its a feature that stores the messages sent for a certain amount of time. Even if the message was sent to all the subscribers, the message is stored, and if there is a new subscription the retained message is sent to that new subscriber [Lampkin et al., 2012].

Clean session is a flag that is set when the connection between the client and the broker is established. This flag enables the client to have a persistent connection. This means that when the client for any reason disconnects, the broker maintains or not his subscriptions and messages sent with QoS greater or equal than one depending on the value of the flag. If the flag is set to true, after the disconnection occurs, the broker forgets all the subscriptions that the client had. If the flag is set to false the connection will be persistent and the subscriptions are saved, and subsequent messages with higher QoS than zero are stored and delivered when the connection is reestablished [Lampkin et al., 2012].

Last will is also defined when the connection with the broker is established and is a message that is published on a certain defined topic or topics in the occurrence of unexpected disconnection. This is useful to notify that probably something went wrong with the client [Lampkin et al., 2012].

To use this protocol, there are several tools to make the process of using MQTT easier. Since the broker is the most complex component of this paradigm, there are several brokers available to use. This makes the work easier in terms of adopting this protocol since we do not have to worry about the details of the development of a broker. Software like Mosquitto [Eclipse, 2019a] was developed to be an easy to use broker.

2.2.9 Mosquitto

Mosquitto is a lightweight open-source message broker that implements the MQTT paradigm [Silva et al., 2016]. It is suitable for all kinds of devices from servers with good specifications to low power single board computers. Besides having the broker, it has also MQTT clients to publish and subscribe through the command line, being these clients called mosquitto_pub and

mosquitto_sub [Light et al., 2017].

2.3 Data Visualization Platforms

In an era where data analysis is extremely important, there was the necessity of creating data visualization platforms to discover patterns and hidden behaviors. Data analysis is not about the data but it is about its representation because, we can see some curious and relevant aspects of the available information such as patterns [Donalek et al., 2014].

A data visualization platform simply represents data in a graphical way to enable us to analyze it. It uses elements like charts and graphs in order to detect trends in that vast and continuous data. Platforms like Grafana [Grafana, 2019] and Kibana [Kibana, 2019] are good examples of data visualization platforms.

Grafana

Grafana[1] is an open-source feature-rich platform for data visualization. This platform has several widgets available to represent data like graphs, heatmaps, tables, etc. Also, it supports a large number of databases to use as data source like InfluxDB, OpenTSDB, Prometheus, etc. It is remarkable at dealing with time-series data since it has features for navigation on the data like zoom, quick range selection, and within a few steps we are visualizing the desired data [Betke and Kunkel, 2017]. It also provides event annotation in case some event occurs it is highlighted in the graphical representation. Another feature is the alert system. This system notifies us about something that occurs, just by defining some conditions.

Kibana

Kibana[2] is an open-source web-based visualization platform developed by Elastic. Kibana is part of the ELK stack, in other words, Elastiksearch, Logstash, and Kibana. This application works on top of Elastiksearch increasing the functionalities of this stack [Gupta, 2015]. It is used to analyze large portions of data to detect patterns and outliers. For this purpose, it has a widget-based interface that makes the creation of dashboards and graphs an easy task. The interface is intuitive and can fit the needs of every user [Andreassen et al., 2015].

[1] https://grafana.com/
[2] https://www.elastic.co/kibana

2.4 Databases

To meet the requirement of storing data we need a database, and that database needs to satisfy certain requirements that will be further defined.
In these days, databases are essential. Almost every company and website has one. Whenever you request some information, it is most likely that the information will be collected from a database. A DB is nothing more than a collection of information that exists over a long period of time [Ullman, 1997].

Nowadays, there exist two types of databases: relational databases and NoSQL databases.

Relational databases also called SQL databases, are the most common databases and are used in most of the environments. SQL databases are based on the relational model that organizes data into tables that contain columns and rows. A row is a related collection of data, and a column is an attribute of each row. Each table represents an entity type that can be related to other entity types [Elmasri and Navathe, 2010].

NoSQL databases are a class of databases that do not follow the principles of a relational database, usually called SQL database. NoSQL stands for "not only SQL" and emerged to solve the problems of a relational DB. As it was already mentioned NoSQL, it is not a database itself, but a term to filter DBs that are not relational. So inside the NoSQL world, we have some types of databases like Key-value, XML, Document, Column, and Graph databases [Nayak et al., 2013].

This project aims to make automatic data acquisition. Because of this, we need to know what kind of data are we working with. As Big data and Internet of Things (IoT) paradigms are increasingly gaining popularity, since we are surrounded by Wireless Sensor Network (WSN) technologies that are devices that are monitoring the physical world, we have a certain type of data that is time-stamped data, being the most common type of data present in sensor readings [Aljawarneh et al., 2016]. Since sensor-based monitoring of any phenomenon creates time-series data [Guralnik and Srivastava, 1999] and since seismographs and magnetometers have sensors and their measures are obtained over time, they produce time-series data.

2.4.1 Time Series Database (TSDB)

Time series data is a sequence of values representing something at a given time (e.g., humidity). For example, time-stamped logs or sensor measurements can be considered time series data. These measurements have to be ordered according to time-stamp, otherwise, we could lose information. An example of time series data is the seismic measurement that is done every day at a rate of 100 samples per second [Naqvi et al., 2017].

With the emerging of this data type, largely due to IoT, databases specialized in time series were created.

A Time Series Database (TSDB) is a type of database designed and optimized to handle

time-series data. It can be implemented in a relational way or in a NoSQL way. Typically, NoSQL is used because of scalability. It is expected that NoSQL offers great scalability for large amounts of data [Sanaboyina, 2016]. These databases are especially good with this type of data because we can infer some behaviors that will happen. Since time series data represent data over time and are ordered by it, a major part of the workload will be writes, and those writes will be most likely appends to the end of the existing data. Also, due to time series characteristics, writes to the past or future or updates to existing data will be rare and will only be done in some special cases. In a read operation, again due to time series nature, commonly the reads will be in a range of continuous data ensuring a fast query response. At last, delete operations will be also rare and when done will be most likely over a range of continuous data. As it was mentioned a large part of the workload will be writes, so we can expect a large amount of data, so there is a necessity of scalability. Since TSDBs work around the time, several things can be done to compress data. Also, these DBs have time-series common operations and functions in order to make analysis and to increase usability [Naqvi et al., 2017]. Some examples of time series databases are InfluxDB [InfluxData, 2019], OpenTSDB [StumbleUpon, 2019], Prometheus [SoundCloud, 2019], etc.

Table 2.1, represents a comparison between some time-series databases and in order to do that we defined some criteria. To make this Table and to define the criteria we used [IT, 2019] and [Bader et al., 2017].

Open-Source - This is an important requirement since we do not have funds, and we do not want to be attached to a commercial solution as we want to have control over the implementation.

Community Size - With a large community size we have bigger support because there are several users that help each other, and probably if we encounter a problem it was already solved by another user.

Project Continuity - This criterion tells us if the database has continuity because we do not want to have something without support or updates. To answer this, we check what was the date of the last release.

Interface with Visualization Service - One of our goals is to have a graphical representation of the data for interested people, so the interface with any visualization service is essential.

Query Performance - This criterion tells us how fast a query is made.

Supported Data Types - This criterion tells us what are the supported data types. This is important since we need to know if the database supports the data that we want to store.

Supported Programming Languages - This criterion tells us what are the supported programming languages. This can be important to choose one that has a language that we are familiar with. Although this is not a critical criterion. It only can untie the choice in case of doubt between two databases.

Metric Precision - This criterion tells us how big is the metric precision. In this case, the precision is the time precision. It can be important depending on the precision of the data that is being stored.

Server Operating System - This criterion tells us what are the supported operating systems. This is important to know to compare to the equipment available or when creating a virtual machine that can support the selected database.

Write Performance - This criterion tells us how fast can the database insert new data. This is very important since we are dealing with a significant amount of data.

Time Series Databases					
	InfluxDB	**Graphite**	**Riak TS**	**OpenTSDB**	**Prometheus**
Open-Source	yes	yes	yes	yes	yes
Community Size	Large	Medium	Medium	Medium	Large
Project Continuity	yes	yes	yes	yes	yes
Interface with Visualization Service	yes	yes	yes	yes	yes
Query Performance	Fast	Slow	Moderate	Moderate	Fast
Supported Data Types	Int64, Float64, bool and string	Float64	String, Int32, Int64, Float64 and bool	Int64, Float32, Float64	Float64
Supported Programming Languages	Java, JavaScript, Perl, PHP, Python, R, Haskell, etc.	JavaScript (Node.js), Python	C, C#, C++, Java, JavaScript, PHP, Python, etc.	Java, Python, R, Ruby	C++, Haskell, Java, JavaScript (Node.js), Python, etc.
Metric Precision	nanosecond	second	millisecond	millisecond	millisecond
Server Operating System	Linux and OS X	Linux and Unix	Linux and OS X	Linux and Windows	Linux and Windows
Write Performance	350k metrics/sec	50k metrics/sec	30k metrics/sec	20k metrics/sec	800k metrics/sec

Table 2.1: Time Series Database Comparison.

Chapter 3

Architecture

With all that has been said about the seismological and earth's magnetic field concepts and taking into consideration the solutions that already exist, we can now make some decisions about what we are about to implement. For that, in the next sections, we will remind what are our goals for this project, and we will talk about the requirements of the system and the proposed architecture to satisfy those goals. Finally, we go more into detail, and we talk about each specific component.

3.1 Objectives

The goal of this project is to automate of the acquisition of the data gathered by the seismograph and the magnetometer. By this, we mean to replace the current software that connects to these two sensors by one of ours to enable us to have full control of what we want to do and to not be attached to that software that limits us. The automatic acquisition would mean that we acquire the data directly from the sensors, pre-process it, and store it permanently and safely in a database. We want to store not only the data that is gathered with the system that we developed but also the data gathered before this project. For this, we processed the files containing the old data to store it in the database. Furthermore, for the data to have some utility, we needed a web visualization tool for interested people to visualize the data, analyze it, and even be able to export it for personal use.

3.2 Software Requirements

To design and implement a system that automatically acquires data, we established software requirements to achieve all the previously defined objectives:

Automatic data acquisition: We needed a software to directly connect with the sensors to communicate with them in order to configure and acquire data.

Permanent and safe data storage: Data should be stored in an appropriate database such as TSDB. Also, the system must guarantee that no data is lost.

Data available online for visualization and analysis: Acquired data must be available from the web for everyone that is interested to be able to visualize the data and analyse it. To do this, we need a tool that has some key features that enable the user to operate over data, highlight it, and choose what he wants to see.

Ability to export data for personal use: The user must be free to export the data for personal use, so our system must have the ability for the user to choose the time range and export all the data that he desires.

In summary, it was required to connect to each one of the sensors and be capable of acquiring data as well as being capable of setting the configuration. Since both sensors communicate through the serial port, we needed to either read from the serial port or use a USB adapter in order to establish a connection between the sensors and a single-board computer. This computer board was responsible for receiving data, parsing it and then sending it to a repository and for that, we needed Internet connection.

To store sensor readings, we require a database. The data needed to be received and pre-processed before being entered into the database. To do this process, it was needed Internet connection in order to be able to connect to the board and receive data.

Another requirement is to visualize data since these readings only have some value being visualized in some type of graph so that the users (e.g., experts or common people) can do data analysis. Users have to be capable of choosing measurements and time intervals that they want to visualize or even watch the data being captured in real-time. Moreover, they will have the option of exporting the selected data for personal use. For this purpose, we needed to be able to connect to the database hosted in the remote repository, and for that, we also needed Internet connection.

3.3 Proposed Architecture

In general, we want to develop a system that records, stores, and displays seismic and earth magnetic field data. Thus, for this purpose, we have three separated but complementary modules. In other words, we have three core tasks: gather, store, and visualize data, so in order to do that each one of the three core components is responsible for one of these critical tasks.

- Data Gatherer: responsible for the acquisition of data captured by the sensors (seismograph and magnetometer) and for sending data to a remote repository;
- Data Repository: responsible for receiving the data transmitted by the data gatherer module, parse it and then store it in a database;

- Data Visualization: responsible for filtering, processing, and displaying the data that the user wants and export it.

Figure 3.1, shows a high-level view of the proposed architecture.

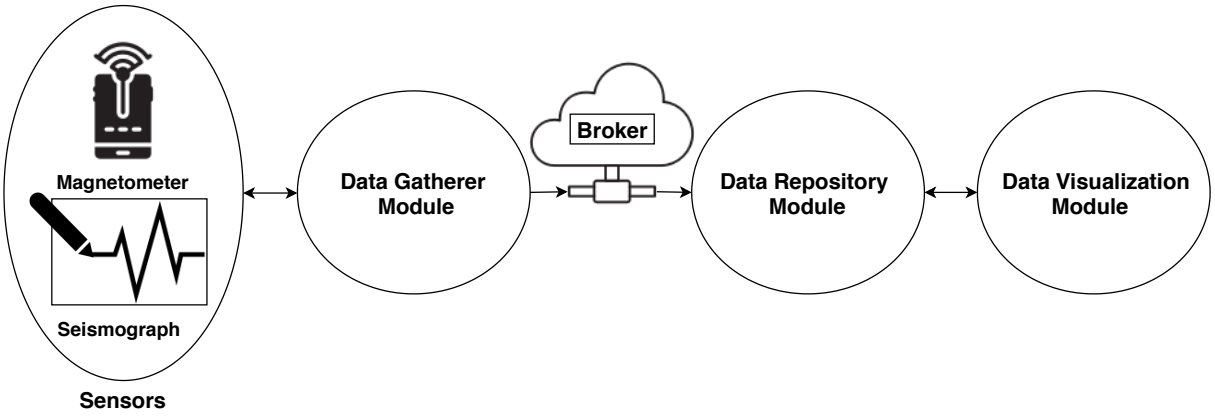


Figure 3.1: High-level view of the proposed architecture of the system representing the three existing modules.

The component that is responsible for the gathering of the data is the data gatherer module. This module is composed of two independent software submodules that connect to both sensors (seismograph and magnetometer) and are used to control and interact with them in order to configure and record data. They will be both running on a single-board computer and will be connected with each sensor by serial or by USB. Also, while recording data they are responsible for simultaneously sending that data to the repository.

The Data repository module is responsible for the store task and is composed by a client and a database. This client receives data from the data gatherer module, processes it, and stores it in the database.

The last task, data visualization, is executed by the visualization module. This module gets whatever data that is requested by the user from the database and displays it in a graphical way for easier data analysis.

For them to work together, they communicate with each other. As shown in Figure 3.1, data gatherer communicates with data repository and data visualization communicates with data repository, being this last one the center of this infrastructure. This system implements a publish/subscribe model. This type of model works by having a client sending data to a broker (publish) and have another client getting the data that was published in the broker (subscribe). Publishers publish data on a certain topic, and subscribers can subscribe to what topics they want. The broker is an intermediary between the publisher and the subscriber. In this case, the publisher is the data gatherer module, and the subscriber is the data repository module, and both of them will publish and subscribe respectively to the broker that will be responsible for receiving and sending data. Having this broker and this model of publish/subscribe will give us

the possibility of scalability without effort and without having to change components. Since if we want to add more sensors gathering data we just need to have more publishers and if we want to add more data repositories, we just need to have more subscribers and that will affect none of the components because the broker deals with the message forwarding by himself. This capability of scaling up by adding new sensors and repositories is important for future developments of this project. Also, the broker can deal with some problematic scenarios that will be discussed ahead. Towards data redundancy, we store data in a database in the data repository but also locally in the data gatherer module.

The data visualization connects to the repository to obtain measurements to represent, and this access is made directly to the database. Also, this module is responsible for sending alerts. These alerts can be configured and can alert to everything the user wants, it just needs a condition to be defined, and then if that condition is reached it sends an alert. An example of an alert is unusual values as the occurrence of outliers. When defining the alert conditions, we can define bounds, and when they are surpassed they can represent an anomaly and can be something to be analyzed and studied. Another example could be to alert by email, for example, if the sensor is not working properly.
This alert function should be straightforward for the average user to be able to use, so it will only be an operation as equal to or greater than and a value. These conditions can be defined in the data visualization interface at any time and in any amount.
Since this type of data is used for studies and analysis it has to be also possible to have certain operations over data like being able to calculate mean, upper bound, lower bound, etc.

Figure 3.2, represents the detailed overall architecture of this project.

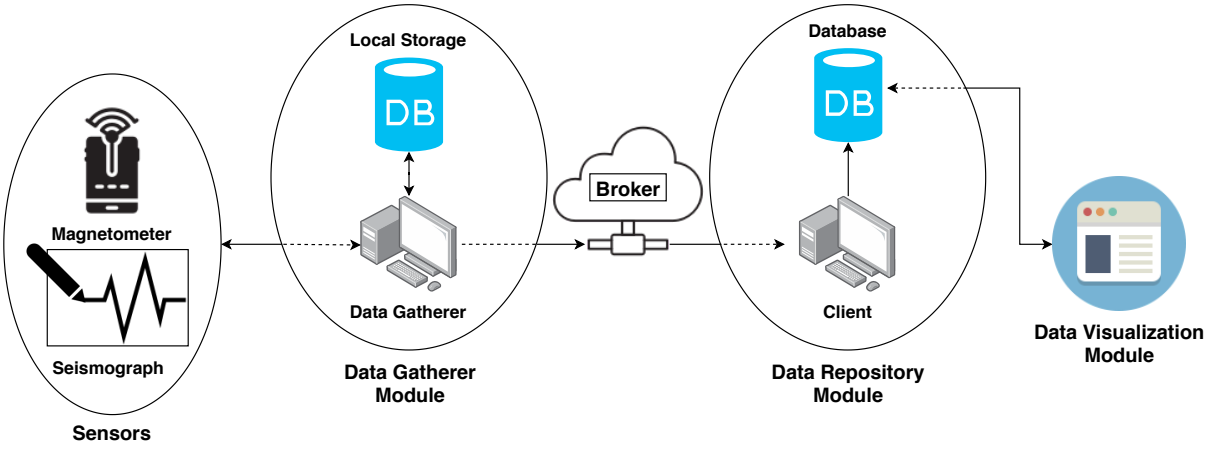


Figure 3.2: Detailed view of the architecture components.

3.4 Data Gatherer

The data gatherer module is divided into two very similar components, but since they work with two different interfaces (magnetometer and seismograph) they needed to have certain specificities.

Nevertheless, both have the same main purposes:

- provide an interface with both sensors in order to configure and retrieve data;
- process retrieved data and send it to a database;
- store data locally for redundancy.

In Figure 3.3, it is represented the data gatherer purposes and architecture.

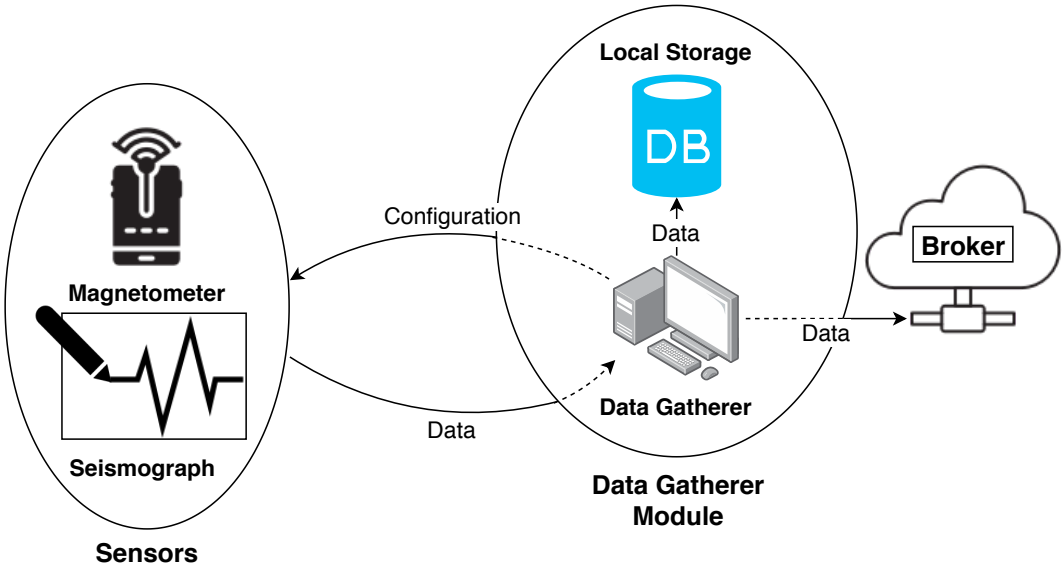


Figure 3.3: Diagram of the data gatherer architecture.

This module had to be capable of sending configurations, retrieve, process, and store data locally and remotely by sending it to a broker. For this to work without problems and human intervention, we must account for certain problematic situations that could occur. In those situations, we have two scenarios: no connection with the sensors or no Internet connection/broker offline (Figure 3.4).

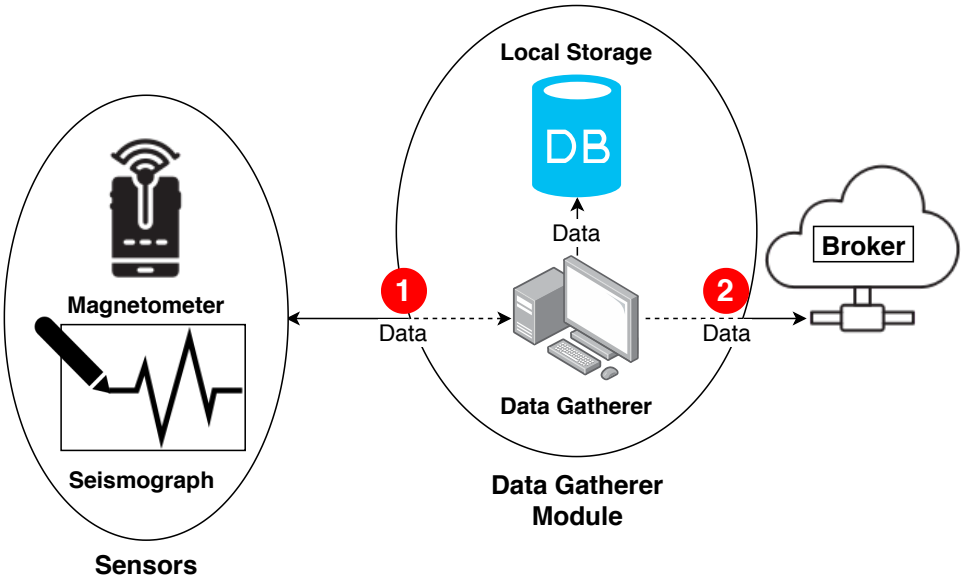


Figure 3.4: Data gatherer problematic situations.

In the first scenario (1), no communication with sensors, there is no perfect solution, because in this case, the problem will be in the sensor or in the communication cable. To minimize the negative outcome of this scenario, the software attempts to reconnect to the sensor, and an error message is published, and an email is sent to the administrator with the error message.

In the second scenario (2), no Internet connection or broker offline, both situations produce the same effect, because the problem is the broken link between the data gatherer and the broker. That will result in no data being stored remotely and consequently, no data to be shown in the data visualization. Also, if there is no mechanism to retransmit data when the link is restored, data will not be available in the remote repository and consequently will result in data loss. To prevent this situation, we had to create a mechanism that detects when there is no Internet connectivity or when the broker is offline, to store that data locally, and retransmit it when possible, in other words, when the Internet connection is re-established or when the broker turns back online. Also, as in scenario 1, it will be published an error message, and a notification will be sent to the administrator via email.

In both cases, it results in a temporary loss of live data so it will be visible in the display and since the visualization module has an alert function to notify about certain defined conditions, we would be aware that something went wrong. This alert function complements the solutions above described and will be discussed in Section 3.7.

3.5 Broker

The broker is responsible for being an intermediary between transactions of data between the data gatherer module and the client in the data repository. As mentioned in Section 3.3, the

broker implements a publish/subscribe model where it is possible to add data producers (data gatherer) that publish information, and we can add clients (data repositories) that subscribe to that data streams.

Besides dealing with all transactions, the broker would allow for scalability for future developments of this project, in other words, it would be possible to add new sensors collecting data and new data repositories without the necessity of changing other, already running, components.

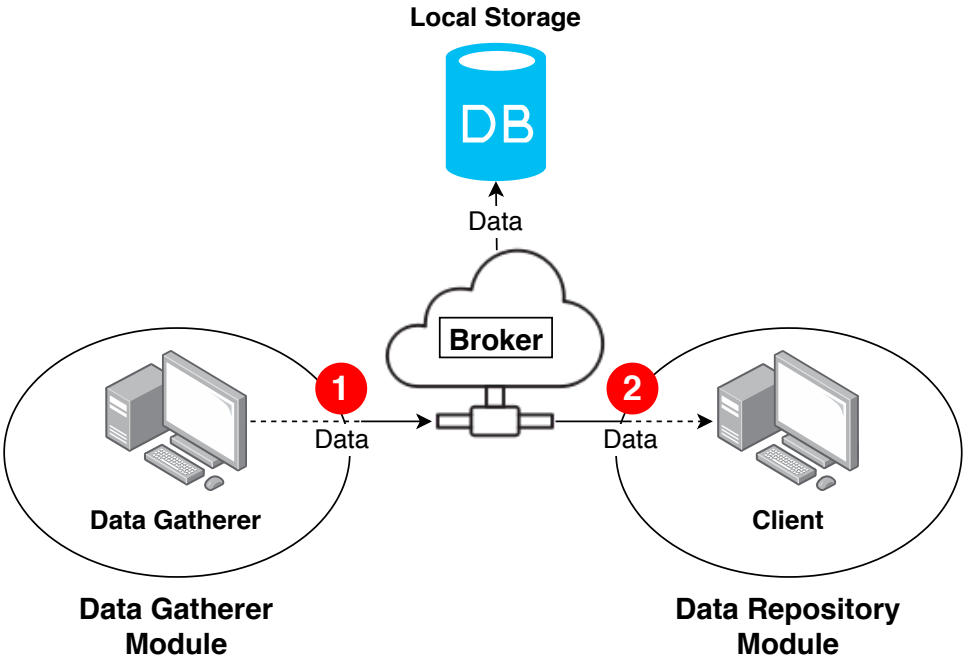


Figure 3.5: Diagram of the broker architecture.

Looking at Figure 3.5, if one of the links with the data gatherer or the data repository module fails, it could be critical, so the broker has to be prepared to deal with that situation. In case of no connection with the data gatherer (1), the broker has to try to reconnect automatically when possible and also sends an error message to the data repository notifying that the data gatherer has disconnected unexpectedly. In the other case, if the link to the client in the data repository is broken (2), the broker will have to store the data that is receiving from the data gatherer module locally and try to reconnect when possible. When the reconnection is successful, the broker has to transmit all data stored locally in the downtime. The data is stored locally in a database and will have a retention policy that was defined in the configurations of the broker to prevent from using all disk in case of the data repository stays offline indefinitely.

3.6 Data Repository

The data repository is where all data obtained by the sensors is stored. This component is the center of the whole system and can be considered the core of the solution since both the

components depend on it. The data gatherer module sends data to the repository to be stored, and the data visualization module requests data in order to represent it graphically.

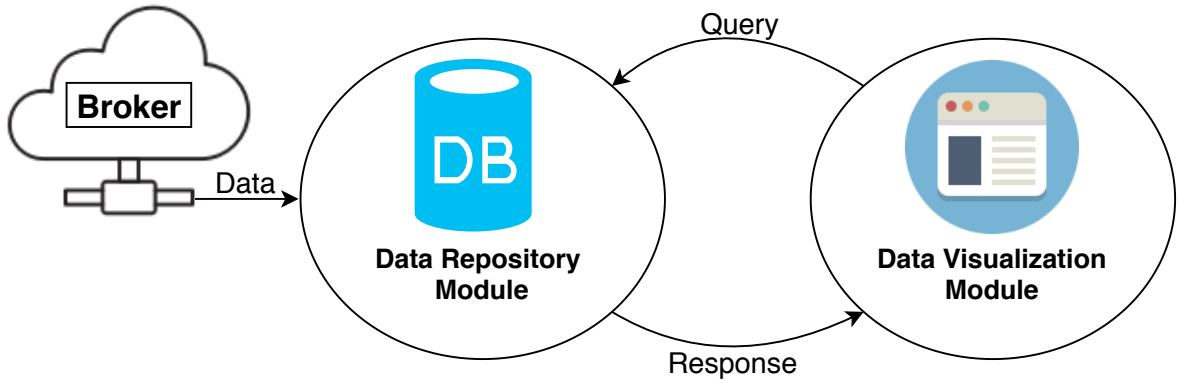


Figure 3.6: Diagram of the data repository architecture.

This module is divided into two parts: client and database. These two components will be addressed next.

3.6.1 Client

The client is responsible for connecting, receiving, and storing the data from the broker. This component has to deal with all the incoming data whether is seismograph or magnetometer data or even error messages. In the case of data, it has to process it and build the query to insert the measurement into the database. In case of error message, it has to notify the administrator by email with the correspondent error message.

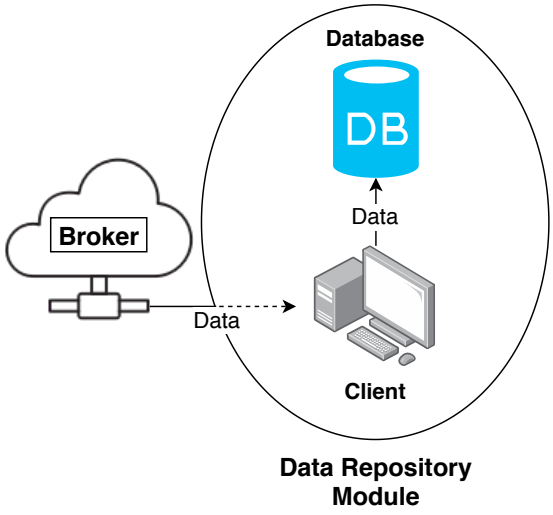


Figure 3.7: Diagram of the client architecture.

3.6.2 Database

The database is responsible for storing data transmitted by the client and providing the data requested by the data visualization module.

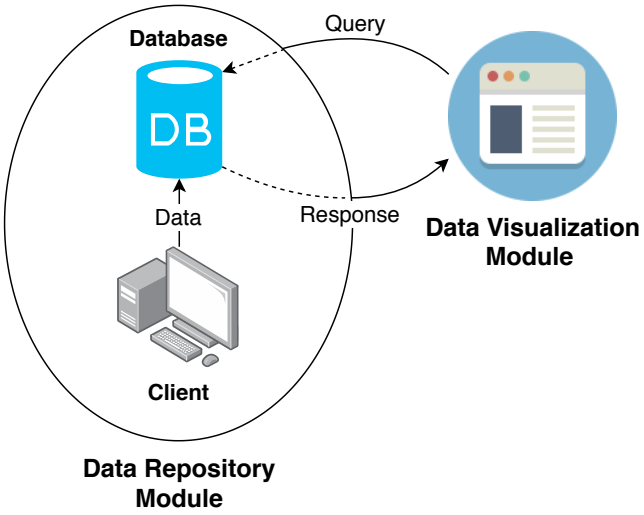


Figure 3.8: Diagram of the database architecture.

Since we are dealing with series of temporal data, it will produce large amounts of data and large amounts of writes to the database. Also, it will have to deal with queries of large chunks of data. With this, we needed to use a database that is optimized for this type of data that has fast read/writes operations and that has scalability, i.e., that does not suffer from the fast growth of the amount of data and workload.

3.7 Data Visualization

The data visualization module is responsible for:

- graphical representation of the data captured by the sensors;
- features to make the process of data analysis easier and possible;
- sending alerts in order to notify for specific conditions;
- export data.

We did not develop a visualization tool from scratch because of the research made about this type of tool. In Chapter 2, we have specified some well suited visualization tools that are robust and versatile enough to achieve our requirements for this module. Also, it has to be as powerful and flexible as possible so that in the future, this project can grow and suit all future goals and developments.

Figure 3.9, shows the data and control flow for this module.

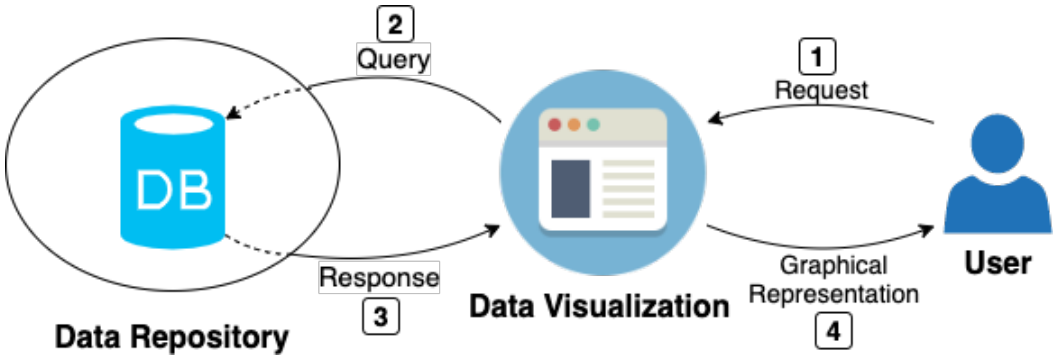


Figure 3.9: Diagram of the data visualization architecture.

Mainly this module is about showing data, and for that, it will need to have access to it. This access is made directly to the remote database in the data repository module.

This module will serve the user so it had to have a very customizable interface and graphical representation. It had to have several programmable modules and customizable data representations. It had to be a modular system, thus already having a rich feature base, but with the possibility of the user creating, for example, graphical representations using a scripting language.

For the graphical representation to be made, data is obtained through queries to the database. So the user just has to choose what measurement he wants to see, and the interface will build the graphic. When choosing the measurement the user also has to pick the time window that wants to visualize, so it has the power to see live data or any other data in the past as long as there are records of that time window. On top of that, after choosing a measurement and time window that he wants to visualize, the user can export data for personal use. Also, it had to have a dashboard to customize things like color, units, zoom, scale, and many others.

The data visualization module also had to have the ability to send alerts via email to notify for events like abnormal readings representing either natural phenomena or sensor failure, in case of no readings at all. These alerts are not hard-coded since they have to be defined by the user because he is the one that chooses what he wants to be notified about, so it has to be defined in the interface and in a simple way for the common user to be able to create alerts seamlessly.

Chapter 4

Implementation

This Chapter will describe how this project was implemented by specifying all the details and addressing the technologies that were used.

For implementing what was defined in the architecture, we have deployed a Raspberry Pi in IGUP serving as data gatherer module, a virtual machine hosted on DCC serving as data repository, broker and data visualization module. It is important to mention that every component can be hosted on different machines, as long as all machines have Internet connection. We adopted the setup shown in Figure 4.1 because of the resources we had available and for simplicity in the process of developing and testing.

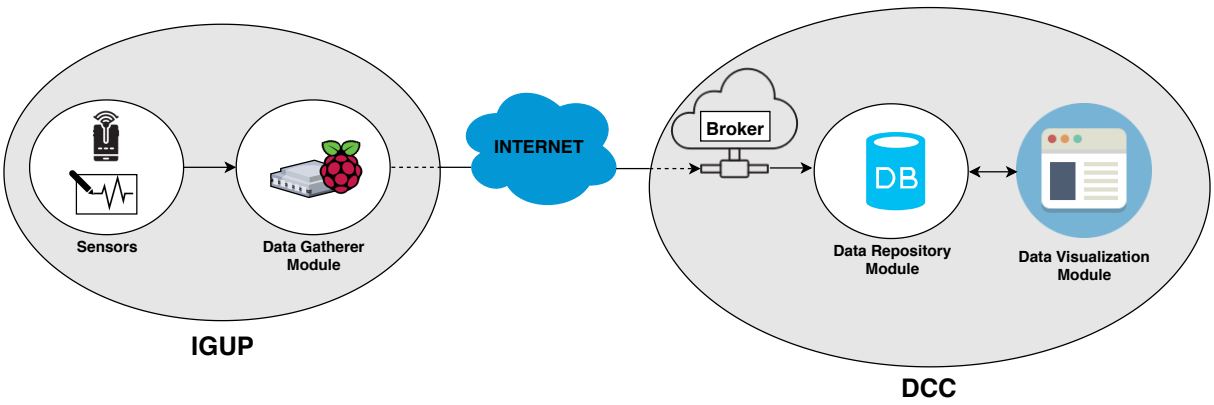


Figure 4.1: Current deployment of the architecture.

With regard to the technologies used, the data gatherer module, as already mentioned has two components, one for each sensor. To develop each component, we used Python for the magnetometer data gatherer software and for the seismograph it was used C++. For the data repository module, it was also used Python to develop the client, and it was implemented a time-series database (TSDB) using InfluxDB. The data visualization module uses an open-source software called Grafana to display all data. The broker uses an open-source message broker called Mosquitto that implements the MQTT protocol. The reason why we chose these technologies is

explained in each respective Section.

For the communication, it was used MQTT protocol between the broker and both data gatherer and data repository module. Between the data visualization module and the database, Grafana uses HTTP to retrieve data in order to show it.

All messages are structured and serialized using JSON, except in the sensor case, there they are transmitted in plain text.

In summary, Table 4.1 shows all the technologies used in this project and where they are used.

Components	**Use**	**Technology**
Data Repository	Database	InfluxDB
Data Visualization	Web visualization	Grafana
Broker	Broker	Mosquitto
Data Gatherer, Broker and Data Repository	Communication Protocol	MQTT
Database and Data Visualization	Communication Protocol	HTTP
Data Gatherer, Broker, Data Repository and Data Visualization	Structured Data Serialization	JSON

Table 4.1: Summary of all the technologies used and their purpose.

Figure 4.2 represents the system with all the technologies that are used.

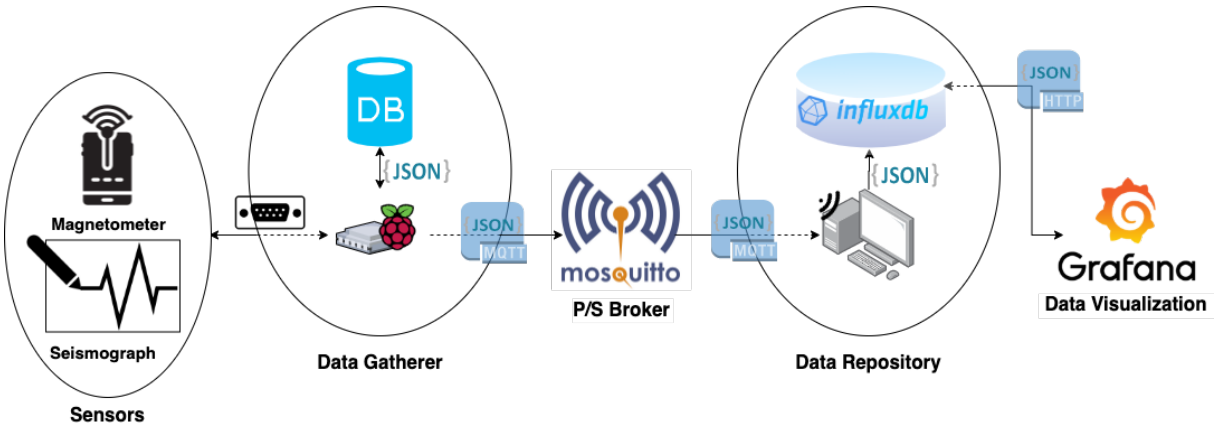


Figure 4.2: System with all the technologies used.

Throughout the rest of this Chapter, we will talk about the implementation of every component.

4.1 Data Gatherer

To start the development of the data gatherer for each sensor, we had to study the communication protocol that was used by the sensors to send data. For that, we wrote a simple Python script that reads the USB port and prints everything that is transmitted. To communicate with the sensors we used a RS232 to USB adapter (Figure 4.3).

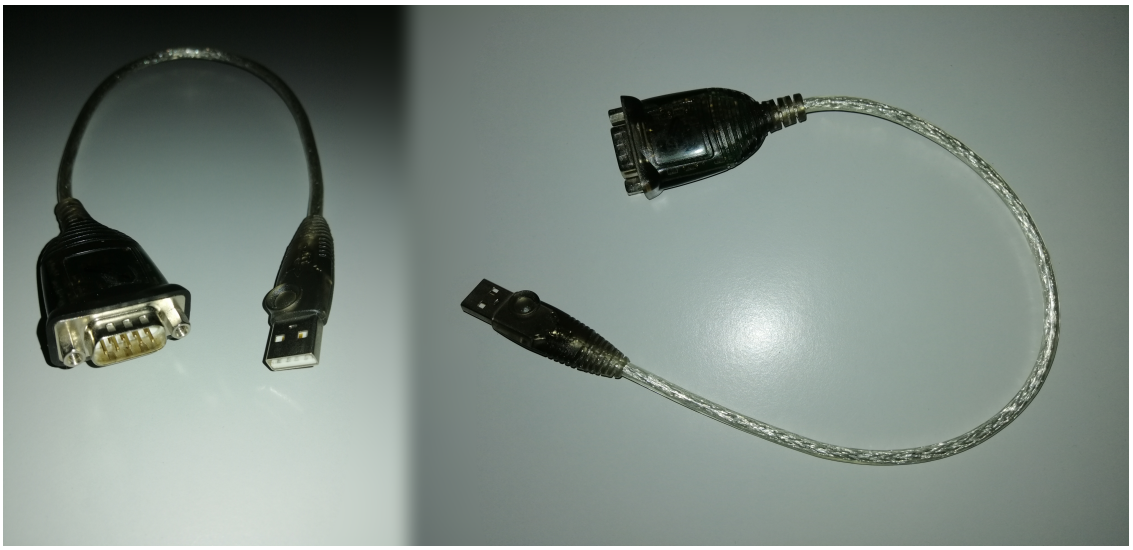


Figure 4.3: RS232 to USB Adapter.

This adapter is plugged to the serial cable in one end, and the other end is a common USB which we connect to the computer running the Python script (Listing 4.1).

```
import serial

def main():
    ser = serial.Serial(port = '/dev/pts/ttyUSB0', baudrate = 9600)

    print 'connected to: ' + ser.portstr

    while True:
        line = ser.readline()
        if line:
            print line

    ser.close()
```

Listing 4.1: Python program to analyze the sensor communication protocol.

Access to USB ports usually requires privileged access. We either execute the scripts with sudo, or we can alternatively add the user to the group that has permissions to use the port.

After running the script, we had two different scenarios. In the case of the magnetometer, our Python script started to print the sensor readings, as shown in Listing 4.3.

```
$ cat 20161210.txt

10.12.16 00:00:00: X,15597,Y,-4481,Z,-43034
10.12.16 00:00:02: X,15600,Y,-4481,Z,-43035
10.12.16 00:00:04: X,15600,Y,-4481,Z,-43035
10.12.16 00:00:06: X,15600,Y,-4478,Z,-43035
10.12.16 00:00:08: X,15600,Y,-4478,Z,-43036
10.12.16 00:00:10: X,15600,Y,-4481,Z,-43036
10.12.16 00:00:12: X,15595,Y,-4483,Z,-43036
10.12.16 00:00:14: X,15602,Y,-4483,Z,-43036
10.12.16 00:00:16: X,15602,Y,-4481,Z,-43037
...
```

Listing 4.2: Sample of data received from the magnetometer.

```
$ cat 20161210.txt

ID=7 Time=17/05/2019 11:18:17.000  Time Ref Status=y [339,680,
-554,127,-23,852,-802,-113,-24,888,-899,886,-380,-205,-70,
-308,-853,170,-934,-460,715,-71,565,318,-582,246,402,9,-480,
456,-595,-684,-476,-693,798,-947,-407,953,410,-172,-934,-661,
217,-742,698,-538,979,576,-65,-143,-727,679,-328,528,-733,236,
330,-729,317,-759,-111,-426,415,581,610,-220,-699,346,581,-609,
810,-636,273,-667,-789,883,583,-825,232,756,-246,897,-183,-256,
559,-724,385,619,577,459,928,633,-536,985,-448,717,899,154,866,
-410,374,-263,-971,359,562,937,-36,425,353,398,483,889,747,-108,
...
```

Listing 4.3: Sample of data received from the magnetometer.

With this, we realized that the magnetometer sends data in plain text without any type of handshake. That did not happen with the seismograph, because we could not get any data with our Python script.

In the case of the seismograph, it had to have some kind of handshake to start sending data. To solve this, we used a library that was developed by the creators of the seismograph and the software to capture and visualize the data (WinSDR 2.2.2).

4.1.1 Magnetometer

To develop the magnetometer data gatherer, we used Python. This component parses each line splitting it into several fields: date, hour, X component, Y component and Z component. Thereafter, it is necessary to serialize these fields, store locally in a file and send it to the broker.

For connecting we used a script like in Listing 4.1, but before connecting we fetch the initial configuration in `MagnetometerConfiguration.ini` (Listing 4.4).

```
[Magnetometer]
Serial_Port = /dev/pts/ttyUSB0
Baudrate = 9600
Magnetometer_Units = Tesla
Magnetometer_Model = Simple Aurora Monitor SAM

[MQTT]
Broker_Address = 127.0.0.1
Broker_Port = 1883
Broker_Username = igup
Broker_Password = *******
MQTT_ClientID = Magnetometer_Publisher
MQTT_Topic = IGUP/Magnetometer
MQTT_Topic_Error = IGUP/Magnetometer/Error
QoS = 1
Will_Message = Magnetometer Data Gatherer gone offline

[LOG]
Offline_Log_File = MagnetometerOfflineData.log
Local_Data_Storage_File = Magnetometer
Max_Offline_Storage = 0
```

Listing 4.4: MagnetometerConfiguration.ini file.

This configuration file generally allows us to set variables from the serial baud rate and port, from MQTT information like IP address and port of the broker to more specific things like log names and the max number of messages stored when the broker is offline. This number is defined to 0 to not discard messages and to store as long as there as memory. If we want to restrain the number of messages stored we just have to define a number and if that number is reached the messages begin to be discarded.

Thereafter we connect to the broker and define callback functions that will be explained later in this section. Listing 4.5 shows the code for dealing with the broker.

```
def InitializeBrokerConnection():
    client = mqtt.Client(MQTT_ClientID, clean_session = False) # Create new
        instance
    client.username_pw_set(username = Broker_Username, password =
        Broker_Password) # Set credentials
    client.on_connect = on_connect # Define connect callback function
    client.on_disconnect = on_disconnect # Define disconnect callback function
    client.will_set(MQTT_Topic,Will_Message,qos=QoS) # Set the last will
    client.connect(Broker_Address, Broker_Port) # Connect to the broker
    client.loop_forever()
```

Listing 4.5: Function to connect to the broker and to define callback functions.

After receiving a message from the magnetometer, we parse that message using the `Parse()` function. This function receives the raw magnetometer message and returns a JSON to be sent to the broker.

```
def Parse(line):
    correctMessage = re.search('\d+\.{1}\d+\.{1}\d+
        \d+[:]{1}\d+[:]{1}\d+[:]{1}(X,){1}-?[0-9]\d*(\.
        \d+)?(,Y,){1}-?[0-9]\d*(\.\d+)?(,Z,){1}-?[0-9]
        \d*(\.\d+)?',line).group(0)
    splittedLine = correctMessage.split(' ')
    date = splittedLine[0]
    date = date.replace('.', '/') # For example 17.02.19 to 17/02/19
    date = date[: 6] + '20' + date[6: ] # 17/02/19 to 17/02/2019
    hour= splittedLine[1]
    hour = hour[: 8] + '.' + hour[9: ] + '000' # 17:45:20 to 17:45:20:000
        (including miliseconds)
    measurements= splittedLine[2].split(',')
    x = measurements[1]
    y = measurements[3]
    z = measurements[5]
    z = z.replace('\r', '')
    z = z.replace('\n', '')
    json_body = GenerateJSON(date, hour, x, y, z)

    return json_body
```

Listing 4.6: Function that parses the information and generate the JSON.

As shown in Listing 4.6, there is a call to a function that will create the JSON object. The JSON, has the date, hour, units of the readings, information about the magnetometer and an array with three elements representing each component of the sensor. The JSON looks like this:

```
{
   "Date": date,
   "Hour": hour,
   "Magnetometer_Units": Magnetometer_Units,
   "Magnetometer_Model": Magnetometer_Model,
   "Data": [
      x,
      y,
      z
   ]
}
```

Listing 4.7: Magnetometer data in JSON.

Finally, we send the JSON to the broker publishing in IGUP/Magnetometer topic with `QoS=1`.

For redundancy, we store data locally in files to maintain the old standard of IGUP. If someone wants to use the old way, it can go there physically and access the files. Each day creates a

new file with the name of the sensor and the date, for example, `Magnetometer.15-08-2019`. Each entry is stored in JSON to make it easier to parse. To do this, we used a Python library called logging that has a module denominated `TimedRotatingFileHandler` that creates these log files automatically by defining when they are supposed to be created e.g., at midnight. In this case, we defined midnight to be the time when a new log file is created, because it is when it represents a new day.

About the problematic scenarios described in Section 3.4, in the case of no connection to the broker, we will save data locally in a file until the reconnection is successful. After the connection is reestablished, we will flush all data from the file to the broker. To develop this function, we used Paho MQTT callback functions to detect when the data gatherer is connected to the broker or not, and we used the Python library used to store data locally to create the file in which we store the offline data. This library deals with the creation of the file and the appends of new data. Also, in case the sensors disconnect an error message is published on `IGUP/Magnetometer/Error` and in case the data gatherer disconnects, it sends the last will that is a function of MQTT that is a pre-defined message automatically transmitted in case of unexpected disconnection. This last will also represent an error message, and both these messages will be treated in the data repository where the administrator is notified via email with the error message.

4.1.2 Seismograph

For developing the seismograph data gatherer, we used a library (PSNADBoard.DLL 2.2.6). The library was developed by the creators of the seismograph to enable any user of making its own data gatherer. Since the library is in C++ and it has a demo app also in C++ we stuck with this programming language to implement this component. With this, we have a mechanism to communicate with the seismograph that enables us to receive data and configure it.

The procedure is similar to the procedure of the magnetometer. First, we connect to the seismograph and to the broker. After that, we fetch the configuration from `SeismographConfiguration.ini` file (Listing 4.8) and send the configuration to the seismograph.

```
[Seismograph]
Def_Port_Number = 0
Def_Port_Str = "/dev/ttyusb0"
Def_Port_Speed = 38400
Def_Time_Ref = 0
Def_Sample_Rate = 100
Def_Number_Channels = 6
Def_High_To_Low_PPS = false
Def_No_Led_Status = false
Def_12Bit_Flags = 0
Def_Time_Offset = 0
Def_Check_Time = 1
Def_Set_PC_Time = 0
Set_PC_Time_Normal = 1
Set_PC_Time_Large_ok = 2
Seismograph_Units = m/s
Long_Period_Model = Sprengnether Ewing-Press type
Short_Period_Model = Geotech Benioff

[MQTT]
Broker_Address = 127.0.0.1
Broker_Port = 1883
Broker_Username = igup
Broker_Password = iguppassword
MQTT_ClientID = Seismograph_Publisher
MQTT_Topic = IGUP/Seismograph
MQTT_Topic_Error = IGUP/Seismograph/Error
QoS = 1
Will_Message = Seismograph Data Gatherer gone offline

[LOG]
Offline_Log_File = SeismographOfflineData.log
Local_Data_Storage_File = Seismograph
Max_Offline_Storage = 0
```

Listing 4.8: SeismographConfiguration.ini file.

This configuration file, almost like the one to configure the magnetometer, has the seismograph, MQTT and log information but this has more fields about the seismograph itself because this sensor has more variables available to configure.

Finally, data begins to flow and is parsed in order to send it to the broker. We use Rapidjson [Yip, 2019] to create a JSON object. The JSON has the date, hour, units of the readings, information about both seismic sensors and an array of 600 elements representing 100 readings by each of the six sensors and looks like this:

```
{
   "Date": date,
   "Hour": hour,
   "Seismograph_Units": Seismograph_Units,
   "Long_Period_Model": Long_Period_Model,
   "Short_Period_Model": Short_Period_Model,
   "Data": [600]
}
```

Listing 4.9: Seismograph data in JSON.

All 600 readings correspond to a period of one second. We have six sensors capturing seismic data and the sample rate is 100 samples per second. This means that each sensor gets 100 values and each one of them is acquired every 10 milliseconds.

Since we were not able of making the Paho MQTT library for C++ work, we were forced to send the output of the C++ program to a python script similar to the script used in the magnetometer data gatherer. To do this, we used a pipe to redirect the output of the C++ script to the input of the python code. There we connect with the broker, we store the data locally and send it to the broker.

4.2 Broker

For the broker, we used a message broker that implements the MQTT protocol called Mosquitto (Section 2.2.9). As already mentioned, the broker will serve as an intermediary between the data gatherer and the data repository module dealing with the forwarding of messages.

Our use of Mosquitto was almost plug and play, this means that we only installed it and it was already fully functional. We did little to none configurations, but it was possible to configure several things. One of the modified configurations was the authentication to prevent unwanted subscribers or publishers from using our broker. To do this, we created `password_file.txt` in `/etc/mosquitto`, and we wrote the desired ID and password in this format: ID:PASSWORD. After that, we run the following utility command:

```
mosquitto_passwd -U /etc/mosquitto/password_file.txt
```

Listing 4.10: Mosquitto command to encrypt password in authentication file.

The command shown in Listing 4.10 modifies the `password_file.txt` file by encrypting the PASSWORD field. After that, we just had to go to the Mosquitto configuration file and change two fields. These fields are shown in Listing 4.11.

```
allow_anonymous false
password_file /etc/mosquitto/password_file.txt
max_queued_messages 172800
```

Listing 4.11: Modified fields in Mosquitto configuration file (mosquitto.conf) to enable authentication.

The first field is to enable and make the authentication mandatory in each connection. The second field is the path to the list of users and respective passwords that are allowed to establish a connection. The last field is to define the number of queued messages. In other words, when a subscriber with persistent connection disconnects, the broker stores locally the messages that were not delivered and while the client does not reconnect the broker queues all messages received. After the client reconnects, the broker automatically transmits all queued messages. This field is important since we do not want to fill the memory of the device where the broker is hosted. In the case that a client becomes offline indefinitely, the broker would store all the messages and would fill the memory. For this, we defined a max number of queued messages of 172800 that represents two days in seconds, since our worst case is the seismograph that sends the data every second. We can also set that value to 0 representing indefinitely storage, in other words, until the disk gets filled.

Finally, we just need to restart the Mosquitto service. In our case, we just stop the service and then we run Mosquitto with the following command in Listing 4.12.

```
mosquitto -c /etc/mosquitto/mosquitto.conf
```

Listing 4.12: Command to run Mosquitto with the defined configuration file.

This will run Mosquitto with the configuration file that we created with the authentication enabled.

Mosquitto and the MQTT protocol let us add more data gatherers and data repositories without having to modify any of the previous data gatherers and data repositories already operational. This is good for future developments of this project since new sensors and new data repositories can be added. As mentioned, this project uses a publish/subscribe protocol that works on top of topics. In our solution, we have four topics: IGUP/Magnetometer, IGUP/Seismograph, IGUP/Magnetometer/Error, and IGUP/Seismograph/Error. MQTT enables us to have a hierarchy in our topics similar to a file system where there are folders and subfolders. In this case, for organization and to be easy to add more topics and stay organized and easy to understand, we put our specific topics under IGUP topic. In Figure 4.4 can be seen our implementation of the broker publish/subscribe protocol.

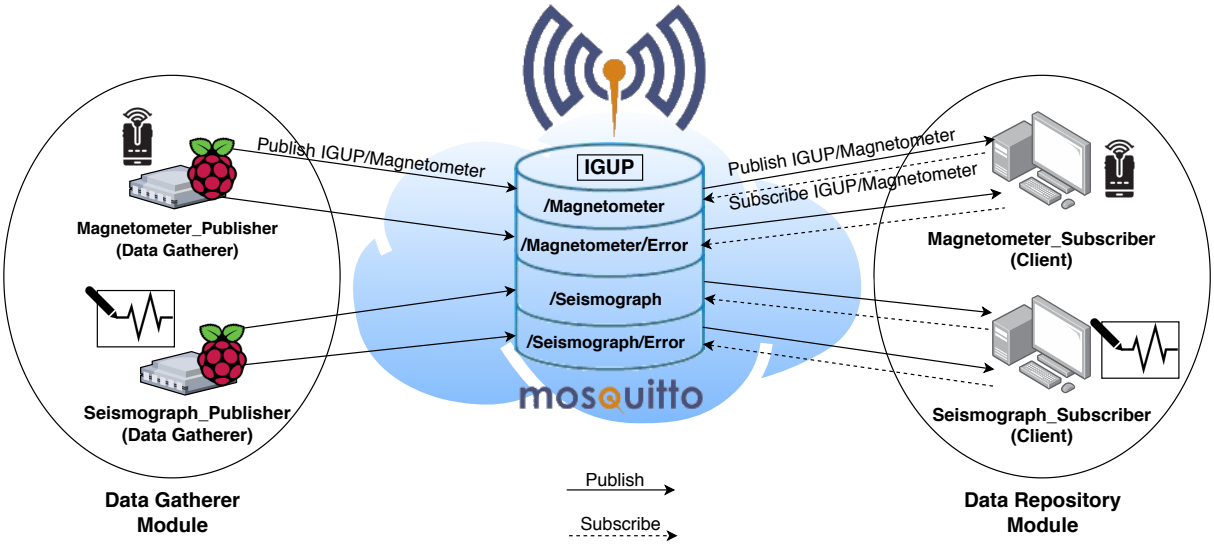


Figure 4.4: Publish/Subscribe diagram of our solution.

The configurations made in the clients were the client ID, the quality of service (QoS), last will, and whether the session is persistent or not (clean session). The client ID is used to identify each client that connects to the broker and its state. Therefore, it has to be unique, but it is not necessary to have it since the MQTT protocol accepts clients without ID. Not sending the ID has some negative aspects that we will approach after the other configurations are explained. The QoS, as already mentioned in Section 2.2.8, also known as Quality of Service is a level that is specified between clients (sender and receiver) that defines the guarantee of the delivery of a message. There are three levels of QoS: levels 0, 1, and 2. Level 0 sends the message once and does not care if it is delivered. Level 1 ensures that the message is delivered at least once, but it is possible to receive the same message multiple times. Level 2 ensures that the message is delivered exactly one time being this the safest, but the slowest QoS. Each client defines the QoS they want and if for example the publisher defines a QoS of 2 and the subscriber defines a Qos of 1 the message will be transmitted to the subscriber with the Qos of 1. The last will is a message that is defined when the connection with the broker is made. The broker stores the message and sends it to the subscribers in the case of unexpected disconnection. Finally, the clean session (also mentioned in Section 2.2.8) lets us define whether we have a persistent session or not. In other words, if the clean session is set to false, the broker creates a persistent connection and stores all information about the client like subscriptions and not delivered messages. For example, if a client disconnects for any reason and if the clean session is set to false all the messages with QoS equal to 1 or 2 sent when the client is offline are stored and are transmitted when the connection is reestablished. Otherwise, if the flag is set to true after the client disconnects all information and messages not sent are forgotten and erased by the broker.

In our solution, we combine these four configurations in order to solve the problem of the offline client and consequent loss of data that was described in section 3.5. We set QoS equal to 1 in both clients (sender and receiver) to ensure that the message is delivered. We

do not worry about duplicates since when storing in the database it will overwrite the value causing no harm to the data integrity because both values are equal and the database does not allow equal values with the same timestamp. We also set clean session flag to false to have a persistent connection and we also define an ID for each client. The IDs like we can see in Figure 4.4 are `Magnetometer_Publisher`, `Seismograph_Publisher`, `Magnetometer_Subscriber`, and `Seismograph_Subscriber`. All these configurations combined solve the problem of offline client and data loss because with clean session set to false we have a persistent connection so all information and unsent messages with QoS greater or equal to 1 are stored in the broker database and are sent when the connection is back online. When the connection is restored the broker checks that a session was already created with the same ID of the client that is connecting, so it will retransmit all messages queued. Finally, we also have the last will that is used to notify the data repository (subscriber) about an unexpected disconnection by the data gatherer (publisher) and this module, as a response to this disconnection notifies the administrator via email.

4.3 Data Repository

As already mentioned in Subsection 3.6, this module is where all the information is stored and therefore available to be shown in the data visualization module. For this to happen, we need two components: the database to store all data and the client to subscribe to the information that is published in the broker.

4.3.1 Database

To meet all the requirements mentioned in Subsection 3.6.2, we need to make a choice about which database we wanted to use. For this choice, firstly we looked up to the type of data that we were dealing with. We were working with a type of data that fits in the IoT paradigm where data is being recorded over time, being this type of data called time-series data.

Time series data is time-stamped data that represents how something changes over time. This type of data has three major features:

- Typically data arrives ordered by time;
- Each reading is unique since each timestamp is different and in most cases will be recorded as a new entry, being almost always an append;
- Any query to this type of data will be almost always a chunk of continuous entries.

With this in mind, there were created databases specialized in this kind of data, and they are called Time-Series Databases (Subsection 2.4).

After knowing this, we made a comparison between some TSDBs in order to choose the one that best fits our needs. The comparison is in Table 2.1 and the description of each comparison

parameter is above the Table. After comparing those TSDBs, we decided that the database that most suited us was InfluxDB.

As already mentioned, InfluxDB is an open-source schemeless time-series database developed by InfluxData. Being schemaless means that it will store everything that we want without any problem. The only requirement is to first create a database.

Although InfluxDB is schemaless, we will have a schema, in other words, our entries in the database will always have the same fields. In the eventuality that there is a need to add new fields, there is no problem, because of the schemaless feature of InfluxDB.
Before describing the schema used, we will describe some InfluxDB concepts and how things are done in this database because some decisions about the schema were made on top of these concepts.

In InfluxDB every entry in the database is a point. A point has a measurement, time (UNIX-timestamp), fields, and tags. Measurement is mandatory to be defined. Also, it has to have at least one field and tags are optional. If time is not explicitly defined, InfluxDB will use the local time as timestamp. Conceptually, you can think of a measurement like being a table in an SQL database. The time is the primary index of that table and fields and tags are columns. The measurement is used to define what that point represents (e.g., `Meteorology`). The field is the actual reading of a variable (e.g., `Temperature=25`) and a tag is used for metadata (e.g., `Station=IGUP`, `Temperature_Unit=Celsius`). Listing 4.13 represents the structure of a point in InfluxDB.

```
measurement, timestamp, (tag_key=tag_val)*, (field_key=field_val)+
```

Listing 4.13: Structure of a point in InfluxDB (+ means one or more occurences and * means zero or more accurences of that field).

In Listing 4.14, we show examples of points that can be inserted in the database.

```
Measurement=CPU,Time=1567184511,Fields={value=77},Tags={Machine=Server57}

Measurement=Meteorology,Time=1567184511,Fields={Temperature=25}
```

Listing 4.14: Examples of valid points that can be inserted in InfluxDB.

The difference between fields and tags is that the tags are indexed. This means that when doing a query to a specific field (in the WHERE clause, e.g., `WHERE Temperature=17`) it will sequentially check every value of that field in every entry. This will increase the workload and response time, especially in large datasets. Tags are supposed to store metadata as they are used to describe data in a measurement. We can think of tags like additional information about our data and not the values itself. However, fields, are the data, in our case, actual sensor readings.

With this in mind, we had to be careful when choosing which data is a tag and which is a

field.

In our solution, we have two measurements: Seismograph and Magnetometer. For each measurement, we have the sensor readings, equipment information, and the units of the readings.

In Seismograph measurement we have: `Long-Period_Instrument`, `Short-Period_Instrument`, `Units`, `LP_NorthSouth`, `LP_EastWest`, `LP_Vertical`, `SP_NorthSouth`, `SP_EastWest`, and `SP_Vertical`. `Long-Period_Instrument` and `Short-Period_Instrument` is where is stored the name of the equipment of the long and short period sensor of the seismograph. `LP_NorthSouth`, `LP_EastWest`, `LP_Vertical`, `SP_NorthSouth`, `SP_EastWest`, and `SP_Vertical` are the readings of each sensor, the Long Period (LP) and the Short Period (SP) in each component, North-South, East-West and Vertical component.

Following the knowledge that we had about fields and tags, we decided that `Long-Period_Instrument`, `Short-Period_Instrument`, and `Units` were tags since this is metadata about the measurement. The remaining variables were designated as fields (`LP_NorthSouth`, `LP_EastWest`, `LP_Vertical`, `SP_NorthSouth`, `SP_EastWest`, and `SP_Vertical`) since they are the actual sensor readings. We can see the scheme of the Seismograph measurement in Figure 4.5.

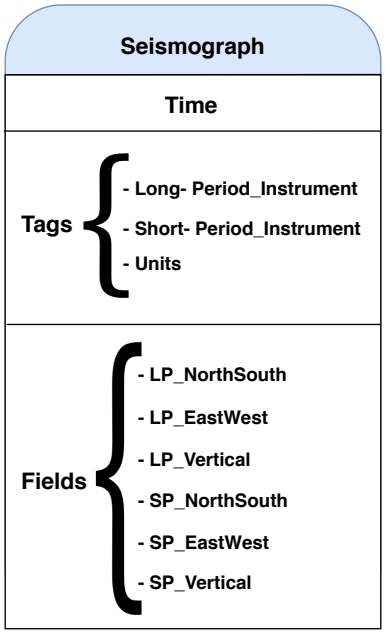


Figure 4.5: Database Seismograph measurement scheme.

In the Magnetometer measurement, we have: `Instrument`, `Units`, `X`, `Y`, and `Z`. `Instrument` is where the information about the magnetometer is stored and `Units` is where we store the units of the reading. `X`, `Y`, and `Z` are the readings of each sensor of the magnetometer in each correspondent axis depending on its orientation.

As mentioned in the Seismograph measurement, we defined information about the measurement as tags, so in this case, the magnetometer case, `Instrument`, and `Units` are tags, since they

are metadata and `X`, `Y` and `Z` are fields because of being sensor readings. We can see the scheme of the Magnetometer measurement in Figure 4.6.

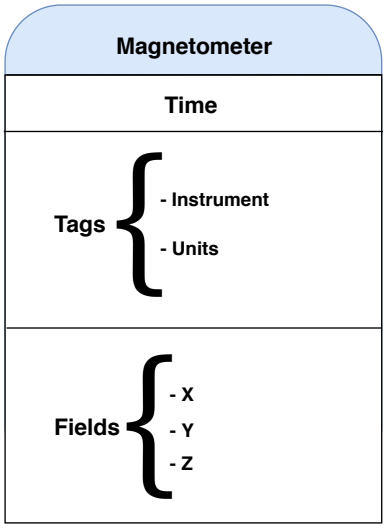


Figure 4.6: Database Magnetometer measurement scheme.

Also, InfluxDB has a feature called retention policy, in which it defines how many time the database will keep that data. By default, it sets the retention policy to always keep the data and never delete it. In our case, we keep the default value because this project is about acquiring and keeping the data, so we do not have the intention to erase it.

4.3.2 Client

For developing both clients, we used Python as the programming language. These components are about connecting to the broker and subscribing to each topic (IGUP/Seismograph, IGUP/-Magnetometer, IGUP/Seismograph/Error, and IGUP/Magnetometer/Error). After that, they just wait for messages to be delivered by the broker, parse them, and store them in InfluxDB. Throughout this Subsection, we will only describe the magnetometer data gatherer since the seismograph data gatherer performs the same process.

Similar to the data gatherers we also have configuration files for the clients: `SeismographRepositoryConfiguration.ini` (Listing 4.15) and `MagnetometerRepositoryConfiguration.ini` (Listing 4.16).

```
[ADMIN]
Admin_email = example@example.com

[MQTT]
Broker_Address = 127.0.0.1
Broker_Port = 1883
Broker_Username = igup
Broker_Password = iguppassword
MQTT_ClientID = Seismograph_Subscriber
MQTT_Topic = IGUP/Seismograph
MQTT_Topic_Error = IGUP/Seismograph/Error
QoS = 1

[InfluxDB]
InfluxDB_Adress = 127.0.0.1
InfluxDB_Port = 8086
InfluxDB_Database = igup
InfluxDB_Time_Precision = ms
```

Listing 4.15: SeismographRepositoryConfiguration.ini file.

```
[ADMIN]
Admin_email = example@example.com

[MQTT]
Broker_Address = 127.0.0.1
Broker_Port = 1883
Broker_Username = igup
Broker_Password = iguppassword
MQTT_ClientID = Magnetometer_Subscriber
MQTT_Topic = IGUP/Magnetometer
MQTT_Topic_Error = IGUP/Magnetometer/Error
QoS = 1

[InfluxDB]
InfluxDB_Adress = 127.0.0.1
InfluxDB_Port = 8086
InfluxDB_Database = igup
InfluxDB_Time_Precision = ms
```

Listing 4.16: MagnetometerRepositoryConfiguration.ini file.

These configuration files enable us to configure the MQTT information like address and port of the broker and topics as well as information about the connection to the InfluxDB like the address, port, and database name. Also, we specify the email of the administrator to notify in case of errors.

The connection and respective subscriptions are shown in Listing 4.17.

```
def InitializeBrokerConnection():
    client = mqtt.Client(client_id = MQTT_ClientID, clean_session = False) #
        Create new instance
    client.username_pw_set(username = Broker_Username, password =
        Broker_Password) # Set credentials
    client.on_connect = on_connect # Define callback function
    client.on_message = on_message # Define callback function
    client.connect(Broker_Address, Broker_Port) # Connect to the broker
    client.subscribe(MQTT_Topic, qos = QoS) # Subscribe to topic
    client.message_callback_add(MQTT_Topic, on_message_from_Magnetometer) #
        Define callback function
    client.subscribe(MQTT_Topic_Error, qos = QoS) # Subscribe to topic
    client.message_callback_add(MQTT_Topic_Error,
        on_message_from_MagnetometerError) # Define callback function
    client.loop_forever()
```

Listing 4.17: Function that connects to the broker, subscribes to a topic and define callback functions.

In this function, besides connecting to the broker, we define two callback functions to know when there is a new message sent by the broker about the topic that we subscribed to. That callback functions are defined by `message_callback_add(MQTT_Topic, on_message_from_Magnetometer)` and `message_callback_add(MQTT_Topic_Error, on_message_from_MagnetometerError)`. These functions tell that when we receive a message from the broker about the defined topic in the first argument, it will call the function defined in the second argument. The callback function can be seen in Listing 4.18.

```
def on_message_from_Magnetometer(client, userdata, message):
    print("Message Received from Magnetometer: " + message.payload.decode())
    Parse(message.payload.decode())

def on_message_from_MagnetometerError(client, userdata, message):
    print("Message Received from Magnetometer/Error: " +
        message.payload.decode())
    NotifyAdministrator(message.payload.decode())
```

Listing 4.18: Callback functions accessed when a new message is received from the broker.

All these MQTT procedures done in the client use the same Python library called Eclipse Paho MQTT [Eclipse, 2019b] just like all the procedures done in the data gatherers.

In the callback function (`on_message_from_MagnetometerError()`), we have a call to the function `NotifyAdministrator()` that will send an email with the error message to the administrator. In the callback function (`on_message_from_Magnetometer()`), we have a call to another function (`Parse()`) to parse the message. This function just split the received JSON into each corresponding field.

Afterwards, function `GenerateJSON()` shown in Listings 4.19 and 4.20 is called to generate a

new JSON respecting the structure that InfluxDB accepts in order to write the point successfully into the database.

```
def GenerateJson(x, y, z, ut):
    json_body = \
        {
            "measurement": "Magnetometer",\
            "tags": {
                "Instrument": Magnetometer_Model,
                "Units": Magnetometer_Units
            },
            "time":int(ut),
            "fields": {
                "X": int(x),
                "Y": int(y),
                "Z": int(z)
            }
        }

    return json_body
```

Listing 4.19: Function that generates the magnetometer JSON to insert into the database.

```
def GenerateJson(value, ut):
    json_body = \
        {
            "measurement": "Seismograph",\
            "tags": {
                "Long-Period_Instrument": Long_Period_Model,
                "Short-Period_Instrument": Short_Period_Model,
                "Units": Seismograph_Units
            },
            "time":int(ut),
            "fields": {
                "LP_X": int(value[0]),
                "LP_Y": int(value[1]),
                "LP_Z": int(value[2]),
                "SP_X": int(value[3]),
                "SP_Y": int(value[4]),
                "SP_Z": int(value[5])
            }
        }

    return json_body
```

Listing 4.20: Function that generates the seismograph JSON to insert into the database.

Finally, after the JSON is generated successfully, it is stored in the database in function `StoreInDB()` as shown in Listing 4.21. For this, we use the InfluxDB library.

```
def Store_in_DB(json_body):
    client = InfluxDBClient(InfluxDB_Adress, InfluxDB_Port, InfluxDB_Database)
    client.write_points(json_body, time_precision = InfluxDB_Time_Precision,
        protocol = 'json')
```

Listing 4.21: Function that stores the point in the InfluxDB database.

4.4 Data Visualization

After data is acquired and stored in the data gatherer and the data repository module respectively, it only has some real meaning being displayed graphically, and for that, we need a data visualization platform. With this in mind, we opted for using Grafana instead of building our own data visualization software. We chose Grafana mainly because it supports InfluxDB and because of all the features and functionalities it has, especially the ones that favor time-series analysis. A brief description of Grafana was made in Subsection 2.3, but along this Subsection, we will go more in-depth about all the functionalities and configurations.

To use this platform, after we installed Grafana we looked into the configuration file (`grafana.ini`) to make the adjustments that we needed. In this case, we only did one change because the default defined parameters suited to us. Mainly, there were three parameters that were important to us. These parameters are the administrator credentials (username and password) and the port for the Grafana server. These parameters are represented in Listing 4.22.

```
[server]
    http_port = 3000

[security]
    admin_user = igup
    admin_password = iguppassword
```

Listing 4.22: Modified Grafana parameters from the configuration file.

We kept the default port and changed the credentials to only privileged users are allowed to have administrator privileges. There is the possibility to create more users with certain privileges.

After logging in as administrator with the defined credentials, we have the home page, and the first step is to add our data source (in this case InfluxDB).

Figure 4.7, shows the configuration to define the data source.

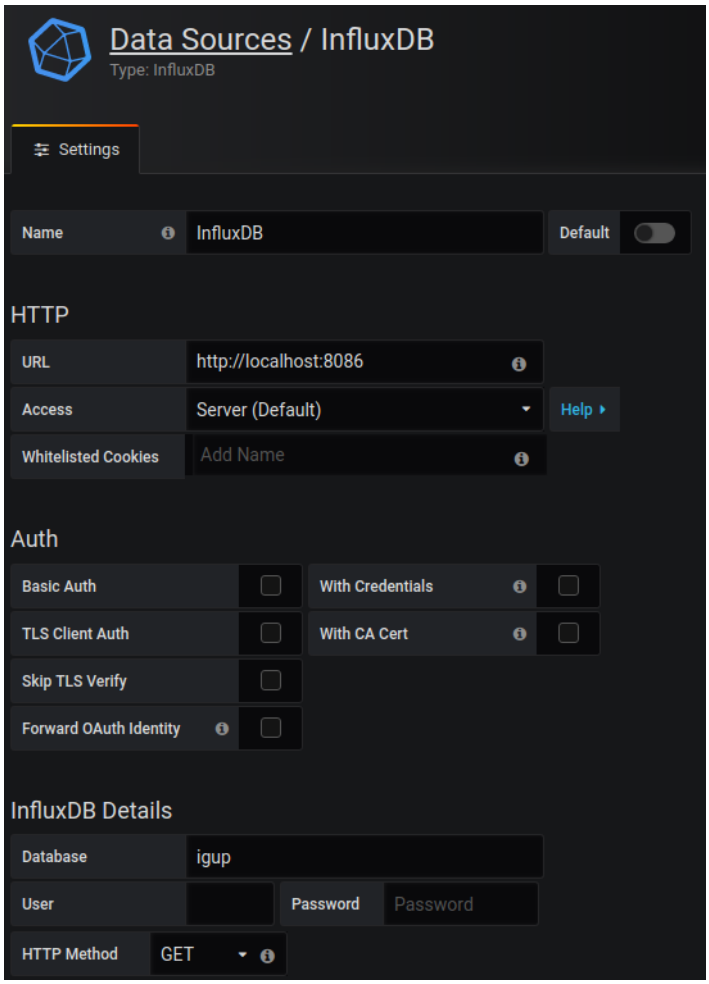


Figure 4.7: Grafana page to add InfluxDB as data source.

To configure the data source, we filled each field as shown in Figure 4.7. We filled the name as InfluxDB, the HTTP URL to http://localhost:8086 and the access we set to the default value (Server). We set HTTP URL to that value because the database and the Grafana service are running on the same machine and we set that port because it is the InfluxDB default port number (8086). We maintained the Access to the option "Server" due to the same reason mentioned about the two services being on the same machine. With the server option, Grafana backend server will deal with the requests to the database. On the other hand, the "Browser" option the browser will communicate with the database itself. If the database and the browser are not in the same machine, the localhost IP will not work. The other field that we filled was "Database" in "InfluxDB Details". This field allows Grafana to know which database to query. In summary, Grafana API connects with InfluxDB API whenever some data is requested by the user.

The next step was to create the graphic displays for the data stored in the database. We already had an idea of what we wanted and for that, Grafana, already had templates to make the graphical representation. To build the desired layout, we created a new dashboard. Dashboards are the core of Grafana as they will contain every graph that we want. In our case, we created

two different dashboards: Seismograph and Magnetometer. The seismograph dashboard has eight graphs each representing one component from each sensor (LP and SP) and two showing the three components of each sensor together. The magnetometer dashboard has four graphs representing also each component (X, Y, and Z) and one with all the components together.

To create a graph, we need to select the data that we want to show. For that, we have to query the database for data. That is done by editing the graph. When editing, we will have a query editor and some fields to fill to build the query. This is shown in Figure 4.8 and Figure 4.9.

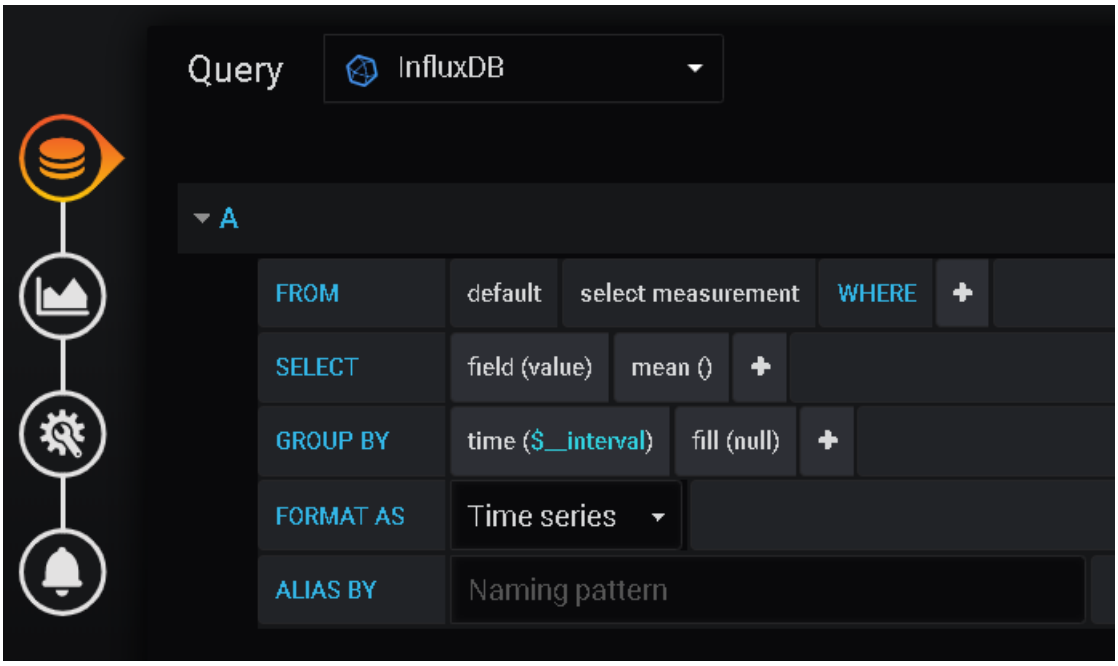


Figure 4.8: Grafana query editor mode.

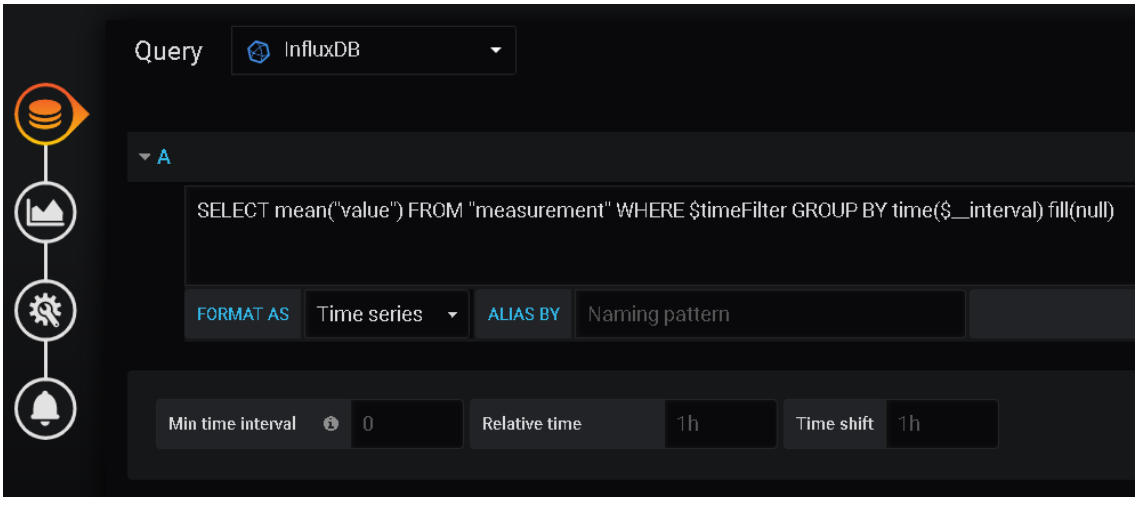


Figure 4.9: Grafana text editor mode to make a query.

The query can be done in two modes: using query editor (Figure 4.8) or writing the query in

the text editor mode (Figure 4.9). In both modes, the queries are done using SQL syntax. Before writing the query, we need to select the data source to query to in the dropdown menu "Query".

Finally, we can customize the graph as we want. Grafana has multiple customizable options.

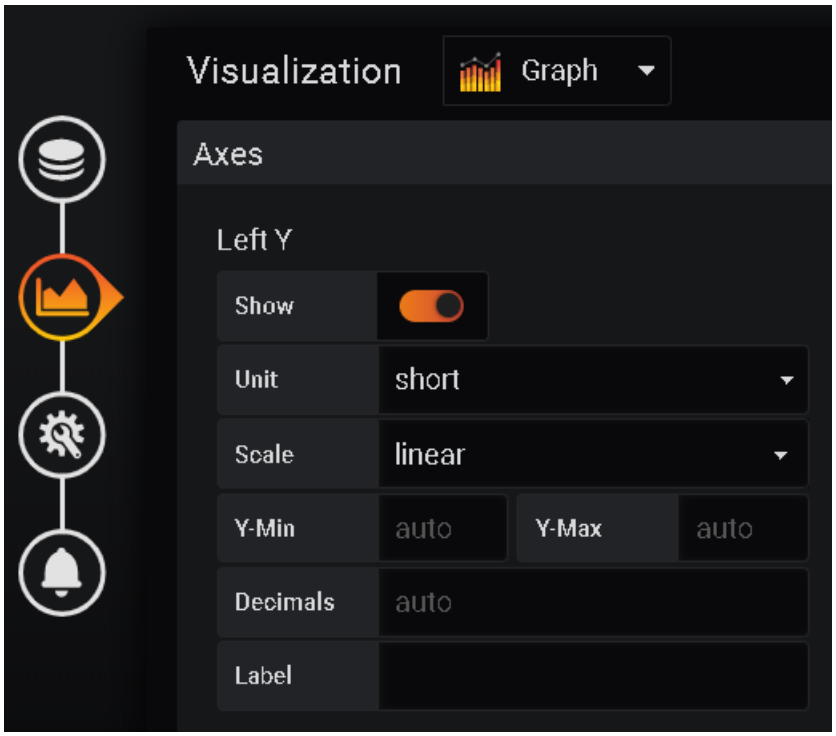


Figure 4.10: Grafana tab to customize the axes of the graph.

In the axes section, as we can see in Figure 4.10, we customized the units to short since the units that we needed was not available, we set the scale to linear and Y-min and Y-max to auto for the graph to choose its min and max according to the incoming data. Decimals we kept the auto option, and in the Label, we wrote our desired units, because this field is to define a text that will appear next to the axis and with this, we can define our units.

Besides the graph, we added extra information. That information is defined in the Legend tab shown in Figure 4.11.

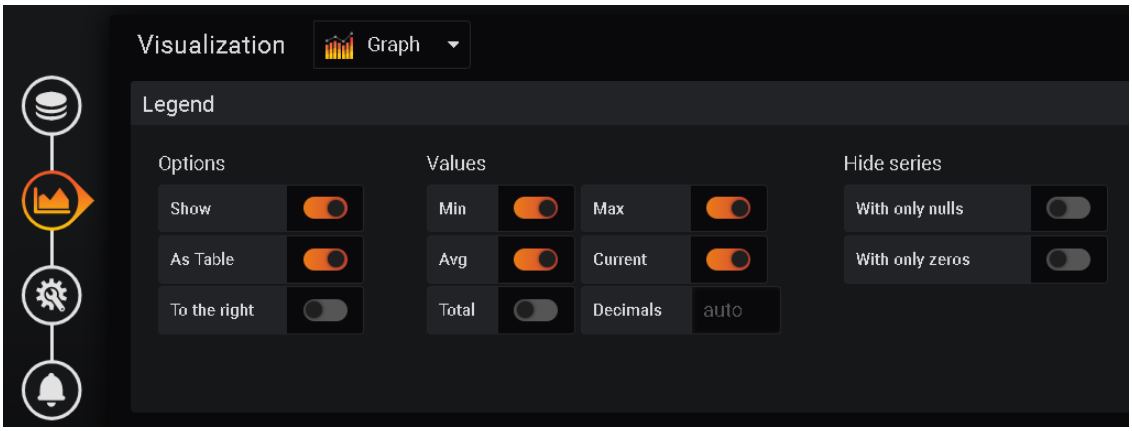


Figure 4.11: Grafana tab to customize the additional information below the graph.

Here we added min, max, avg, and current values to be displayed below the graph.

Another important thing that can be configured are the alerts. This feature is to alert whoever is specified about certain events. Those events are programmed as conditions, and if those defined conditions are reached, the system throws an alert. The alerts are defined as we can see in Figure 4.12.

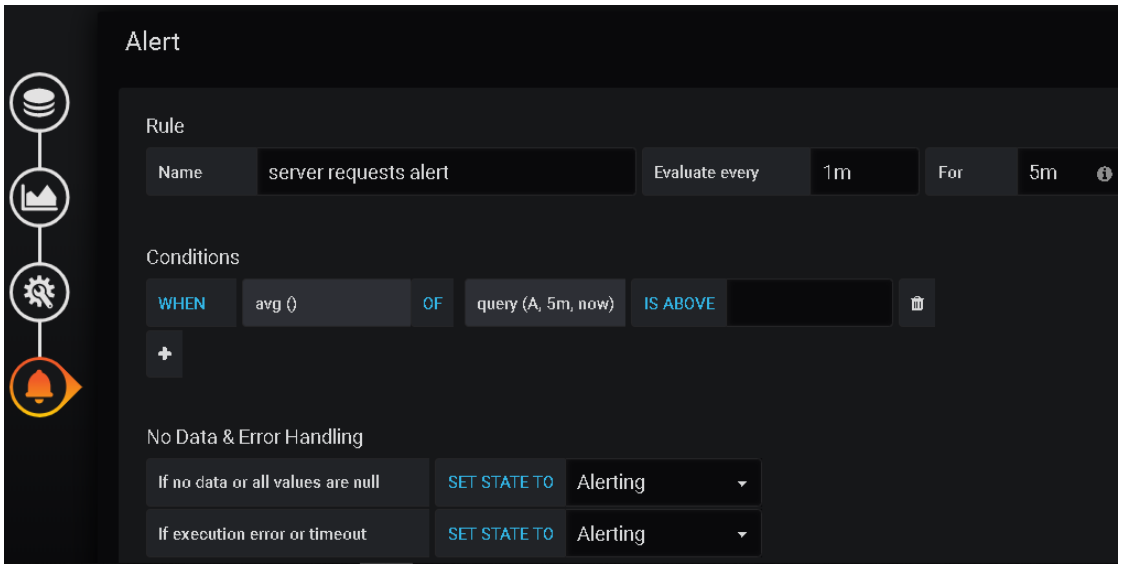


Figure 4.12: Grafana tab to customize the alerts.

For setting up the condition, we need to define the aggregation function (e.g., avg, min, max) then we have to define where will be done the query to check the condition and in what time interval. This is done in the `query()` field. In this function we have three fields: the first is for defining which query to execute, the second and the third are the time range, for example, if we have `query(A,5m,now)` it will use query A in the time interval of 5 minutes ago to now. After the query is made, Grafana checks if the condition is reached. The condition is defined by choosing the operation (e.g., is above, is below, is outside range) and defining the threshold

value. Besides these defined alerts, it has other ones for the case of no data or all values are null, and for execution error or timeout. For these two cases, we can choose if we want to alert, do nothing or keep the last state.

After the alert is specified, it will manifest in three ways: by email (if configured), visually on the graph or by sending a notification to a platform. By email, if we set the message, the destination and if we enable SMTP in the Grafana configuration file. The other type of manifestation is a visual warning on the graph. This visual warning creates a line in the graph representing the assigned threshold and every point above or below, depending on the defined condition, triggers the alert. Also, it is possible to send a notification to a platform like Slack, Discord, etc.

The combination of all the configurations already mentioned can be seen in Figure 4.13.

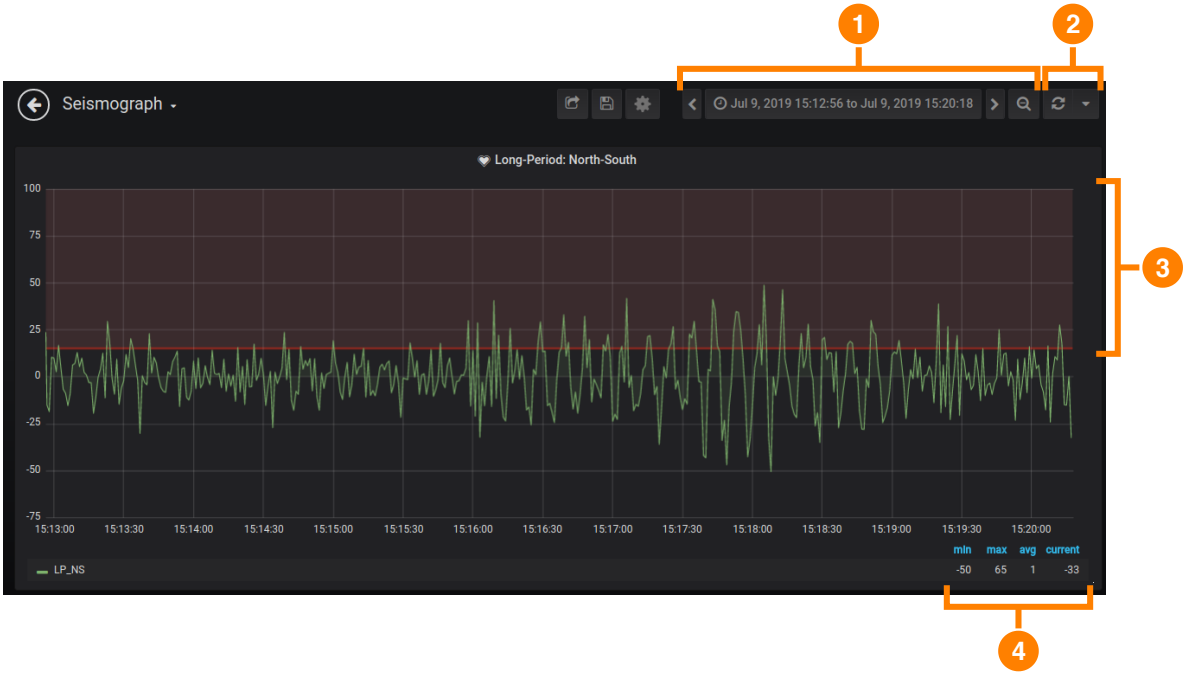


Figure 4.13: Example of a configured graph in Grafana.

Figure 4.13, shows the final version of a graph after the entire configuration has been completed.

1. Time picker. This button enables us to choose what is the time interval in which we want data to be displayed. Also, there is a zoom out button that will enlarge the defined time interval.

2. Manual refresh button. This button will refresh the graph and fetch new data. Also, it has a dropdown menu that lets us pick a timer in which the panel refreshes automatically.

3. Visual warning about the defined alert. It creates a line representing the threshold and, in

this case, a shade above the line representing the range that was stipulated in the condition of the alert.

4. Legend. This shows us the specified values min, max, avg, and current about the graph.

In the dropdown menu in the manual refresh button, we can configure it to have live data visualization. We achieve this by choosing the refresh time to one second (minimum refresh time supported) and by defining the time interval from some value to the current time. For example, if we choose the last five minutes, we would have a time range from five minutes ago to now, and that enables us to see live data by having the graph refreshing every second.

After the graph is ready and displaying data, it is possible to export the data that the graph is showing. By clicking on the title of the graph a dropdown menu will show and in the "More" option it will have the option "Export CSV". This procedure is shown in Figure 4.14.

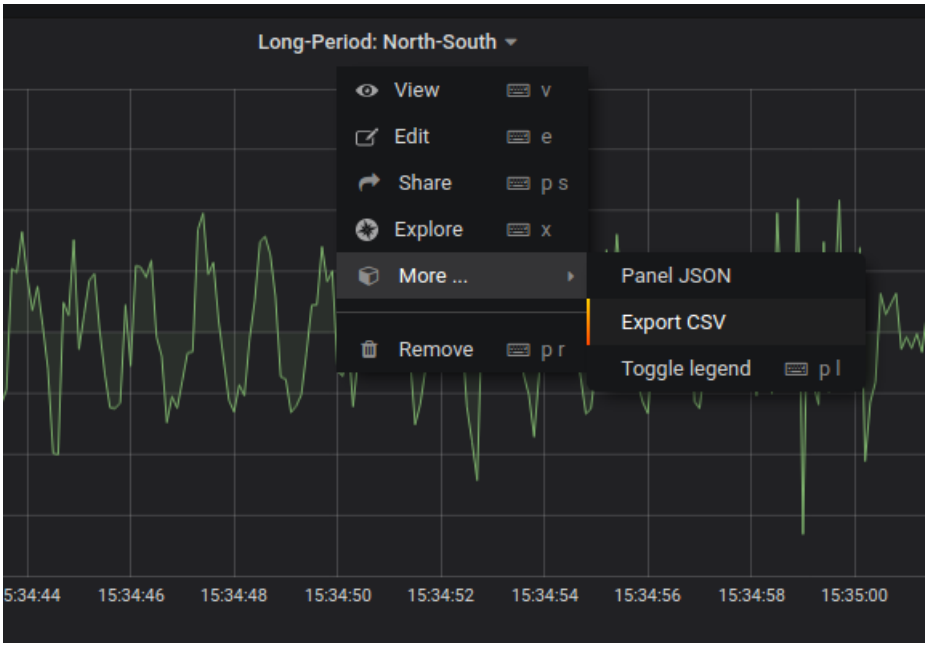


Figure 4.14: Export data in Grafana.

Also, it is possible to get the JSON of the panel to create an equal panel in another dashboard or it is also possible to share it using a link or even with HTML code to embed the graph on a website.

Grafana also has another feature called "Annotations". This feature lets us create an annotation on a specific point or in a time interval of the graph and write a description. Besides that, it enables us to add tags to that annotation, which is very helpful to categorize annotations and to make it easier to find which annotations we want to visualize. These annotations are very useful to describe events that are noticed in the graph. These events can be anomalies in data or even in our case, earthquakes or abnormal Earth magnetic field values. Figure 4.15, shows an example of an annotation and how she looks like in the graph.

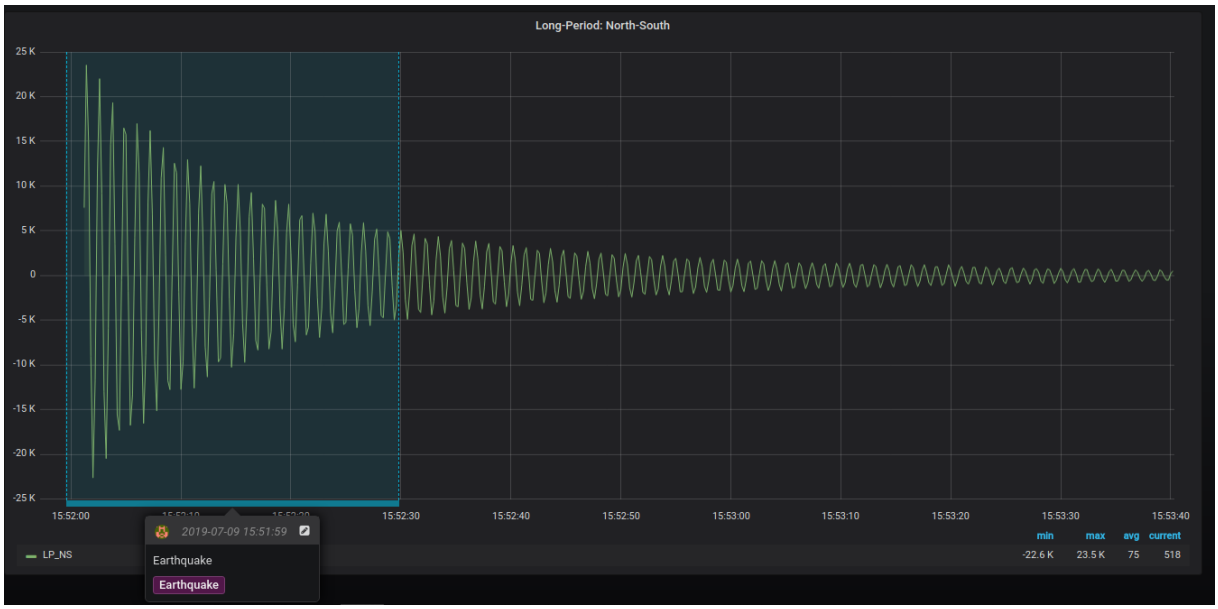


Figure 4.15: Example of an annotation in the graph.

It is important to refer that to write an annotation in a single point of the graph it is necessary to press the control key (CTRL) and click the point. To write an annotation in a time interval, it is also necessary to press the control key (CTRL) and to select the desired time interval.

The final version of one of the dashboards can be seen in Figure 4.16.

Figure 4.16: Partial final version of the Seismograph dashboard.

As previously mentioned, for security reasons, it is possible to create users with certain privileges. The only user that can create and grant permissions to users is the administrator.

There are three categories of users, each one with specific permissions. The viewer can only view dashboards that he has permission to (by default he can see all dashboards). The viewer is not able to create or edit dashboards nor data sources. The Editor can create and modify dashboards but cannot create or edit data sources. The admin is the superuser and can do everything.

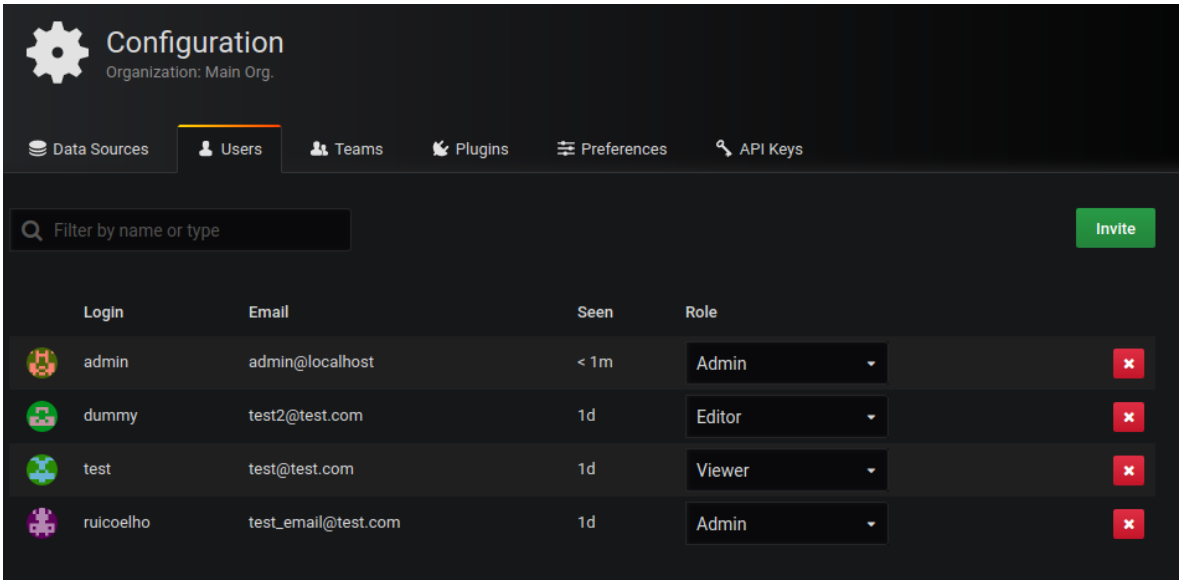


Figure 4.17: Grafana tab with all users and its respective permissions.

Chapter 5

Historic Data Set

Another goal is to process and store all the archival data for both instruments in the database. The archival data available is accessible in two formats: analog and digital. The seismic analog data is available from 1963 to 1993; and 1996. However, some values are missing. For the magnetometer we only work with samples. We do not have complete information about the extent of the dataset and which parts are in digital and analog format. Same happens with the digital seismic data, we only have samples and, also we do not know the extent of this dataset. In this case, we will only deal with the digital data since the analog is not in the scope of this project. The digital data was acquired by the software already described in Section 2.2.2 (Seismograph) and 2.2.5 (Magnetometer). Both software stored data in files and each file correspond to one day, and in the seismograph case we have a sample rate of 100 meaning that we have a reading every 10 milliseconds and in the magnetometer case, we have a reading every two seconds. For retrieving all data from the files, we had to parse each file and save the data into the database. To develop the parser, we had to previously know what is the template in which the file was written. In the case of the magnetometer, we looked into the file and it was equal to the data shown in Listing 4.3. In the case of the seismograph, we were not able to read the files since they were in a binary format.

For the Magnetometer data files, we have a script that works like this:

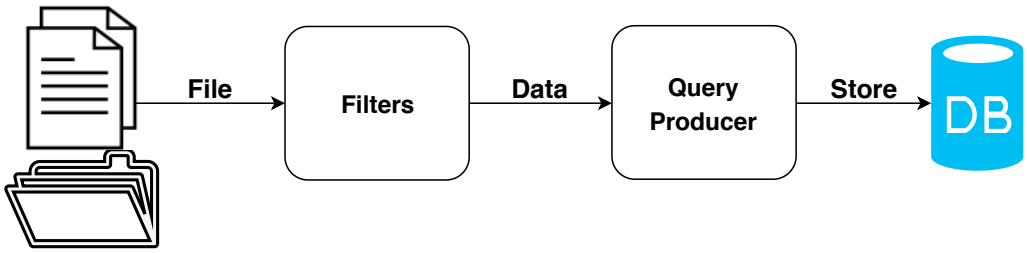


Figure 5.1: Diagram of the historic data script.

We iterate over all data files and parse each one of them. In the parse operation, we split each line in its respective fields but before that, we have a regular expression checking if each line is correctly formed because prior to the process of developing this tool we analyzed the files and noticed some errors. Errors like unfinished messages or overlap of messages in the same line produced by the old software when writing to the file. Listing 5.1, shows some errors on the magnetometer data files.

```
10.12.16 00:00:10: X,15600,Y,-4481,Z,-43036

10.12.16 10.12.16 00:58:17: X,15646,Y,-4492,Z,-43029

10.12.16 00:59:15:  10.12.16 00:59:17: X,15648,Y,-4493,Z,-43028

10.12.16 03:10.12.16 03:01:18:  X,15661,Y,-4454,Z,-43028
```

Listing 5.1: Errors in old magnetometer data files.

To avoid this problem, we apply a regular expression shown in Listing 5.2 to each entry of the file to only catch the correct message. After this, we split each field into a variable, generate the JSON, and insert it into the database. The parser, JSON generator, and database insertion are similar to Listings 4.6, 4.19 and 4.21, respectively.

```
\d+\.\d+\.\d+ \d+[:]\d+[:]\d+[:](
    X,)-?[0-9]\d*(\.\d+)?(,Y,)-?[0-9]\d*(\.\d+)?(,Z,)-?[0-9]\d*(\.\d+)?
```

Listing 5.2: Regular expression to only catch correct data.

Chapter 6

Conclusions and Future Work

With the importance of data in a world where everything is interconnected and recording information, IGUP had the necessity to upgrade its system to be able to contribute with its data to the world. IGUP has a system that gathers seismic and earth's magnetic field data for posterior analysis, but this system is not fully automated since to have access to those gathered measurements someone has to go there physically and download the data. Thus, that data is only available to the person that downloaded it instead of being available to the world for people to study it. Also, this has another inherent problem because all data is stored in only one place that is in the machine where the recording software is running. This is extremely dangerous since the machine can be damaged and all data can be lost. Besides making the current system automated, we also stored the old data that was already in a digital format. In summary, our mission was to make an automation of the process of data acquisition of the seismograph and the magnetometer, to preserve the old records and to make that data available online for anyone interested to be able to make data analysis. For that, we would need to develop a system that replaces the old data acquisition software. A system that gathers data processes it, stores it safely and visualizes it in a tool with features to make the process of data analysis easier. Also, we would need to write scripts to process and store old records. With this in mind, we deployed a Raspberry Pi with a software that gathers the sensor readings, processes them, and sends them to a database. Also, we want to store that data in the cloud for the safety of it, and for that, we will use the UP Digital for this purpose. Currently, the system is not yet storing data in the cloud but the process of hosting a database there is almost completed. Parallel to this, we have a visualization software that connects to that database and is used by the user in order to graphically display the data that he selects.

Overall, we accomplished what we proposed to, but during this project, we encountered some setbacks that made this process harder and made us not able to achieve everything we wanted. Working with systems that were established in the second world war was not an easy task because of the lack of documentation. For this reason, we had to do a considerable amount of research and reverse engineering to understand how everything works. The seismograph was our major difficulty because the communication protocol was not documented and, as mentioned,

the seismograph just did not instantly debit data proving that there was some kind of handshake before starting sending data. After a long time of trying to reverse engineer everything that went through the USB cable and a long time of research, we discovered a library that enabled us to make our own data logger and communicate with the seismograph. After that, we had to study that library and understand everything that it was capable to do, and only then we manage to read our first measurements from the seismograph. With the magnetometer, it was simpler because, in our first connection, we obtained plain text data. Although this task was straightforward, we could not accomplish one thing that we proposed, and it was the capability of configuring the magnetometer. To configure it there was a specific protocol that was not documented. The only thing that we knew was that we had to press a button on the sensor to turn him into configuration mode. Therefore, we were not able to discover what was the configuration protocol, and we were not able to configure the sensor with our own software. The current solution is to configure it with the magnetometer software and then use our solution to gather, store, and display the data. Also, we were not able to let the user choose the format of the file in which he would want to export data because we did not manage to include this feature into Grafana. In this case, there only exist seismic formats, so we would want to include a feature for the user to choose between seismic formats when exporting seismic data.

Regarding the data gathered previous to this project, we were capable of parsing the files of the magnetometer because they were plain text files with the same structure as the data transferred through USB. In the seismograph files, we could not achieve the same result. These files were in a binary format, and we were not capable of reading them, thus not capable of storing them in the database. Currently, we are now parsing seismic files, because the developer of the datalogger of the seismograph gave us a software to convert those binary files to CSV. The converted CSV files are similar to the magnetometer files.

Throughout this project, one of our concerns was future projects and how we could develop this in order to make the integration with our solution and a future improvement possible and simple. With this in mind, we chose the MQTT protocol that makes the integration of both sensors and new data repositories an easy process providing scalability to this project. Also, we use InfluxDB that is a schemeless database, that provides the ability to change the defined schema without corrupting the old one. In the data visualization component, we exceeded our expectations because of the versatility that Grafana offers. In the beginning, we pictured a web page in which a user chooses the time range and after that a graph showing the data. Instead, Grafana is a powerful tool to analyze data since it has numerous functions to simplify this process.

As future work, we would want to add the analog data. In the case of the seismograph, we would need to develop a software that transforms the waves into values to store them in the database. Firstly, we scan every seismogram, and after that, we run the script to get the values. This is hard since the waves overlap each other when there is a stronger reading. Figure 6.1, shows a seismogram in which the waves overlap each other.

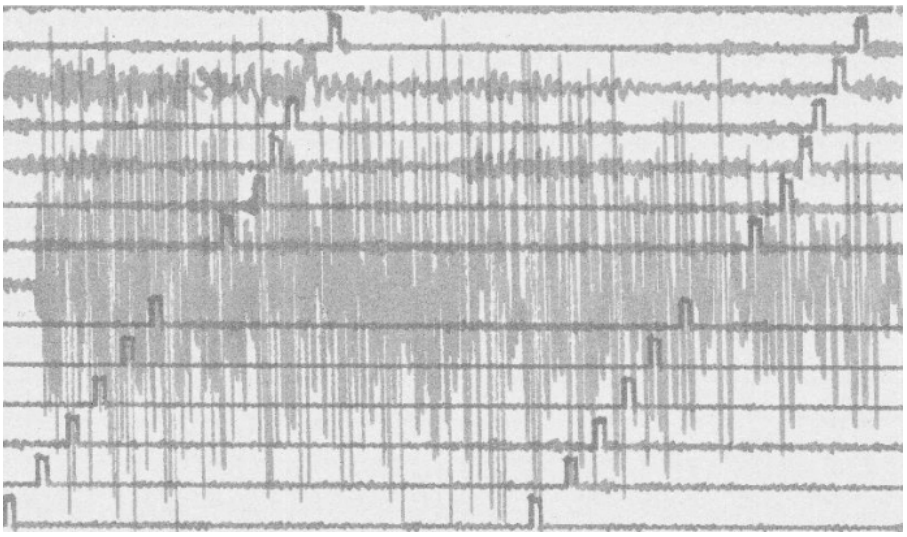


Figure 6.1: Example of a problematic seismogram [Pintore et al., 2005].

To accomplish this task, it would be necessary to develop a sophisticated algorithm capable of differentiating between waves, even if overlapped. This is out of the scope of this project, but it would be an interesting challenge in the computer vision paradigm and a big improvement in the data analysis field since a very large portion of data is stored in the analog format.

Since we implemented our solution thinking in future projects and scalability, a new improvement would be to attach new sensors or data repositories and make this embedded system bigger, and more complete including all type of sensors (geophysical, meteorological, etc.). Finally, we would want to be able to configure the magnetometer with our software and to store the digital historic data from the seismograph.

Bibliography

[Aljawarneh et al., 2016] Aljawarneh, S., Radhakrishna, V., Kumar, V., and Janaki, V. (2016). A Similarity Measure for Temporal Pattern Discovery in Time Series Data Generated by IoT. In *Engineering & MIS (ICEMIS), International Conference on*, pages 1–4. IEEE.

[Andreassen et al., 2015] Andreassen, O., Fuente, A., and Charrondière, C. (2015). Monitoring mixed-language applications with elastic search, logstash and Kibana (elk). In *Proceedings, 15th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS 2015): Melbourne, Australia, October 17-23, 2015*.

[Bader et al., 2017] Bader, A., Kopp, O., and Falkenthal, M. (2017). Survey and comparison of open source time series databases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*.

[Bellavista and Zanni, 2016] Bellavista, P. and Zanni, A. (2016). Towards better scalability for iot-cloud interactions via combined exploitation of mqtt and coap. In *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–6. IEEE.

[Betke and Kunkel, 2017] Betke, E. and Kunkel, J. (2017). Real-time i/o-monitoring of hpc applications with siox, elasticsearch, grafana and fuse. In *International Conference on High Performance Computing*, pages 174–186. Springer.

[Boppidi, 2014] Boppidi, R. (2014). *A Mobile Tool About Causes and Distribution of Dramatic Natural Phenomena*. PhD thesis, Sciences.

[Caruso et al., 1998] Caruso, M., Bratland, T., Smith, C., and Schneider, R. (1998). A New Perspective on Magnetic Field Sensing. *SENSORS-PETERBOROUGH*, 15:34–47.

[Cochrane, 2019a] Cochrane, L. (2019a). Psnadboard.dll. http://psn.quake.net/PSNADBoardDLL.html. Accessed in December 2019.

[Cochrane, 2019b] Cochrane, L. (2019b). Winquake. http://psn.quake.net/wqdocs/. Accessed in December 2019.

[Cochrane, 2019c] Cochrane, L. (2019c). WinSDR Version 4.x Documentation. http://psn.quake.net/winsdr/. Accessed in December 2019.

[Donalek et al., 2014] Donalek, C., Djorgovski, G., Cioc, A., Wang, A., Zhang, J., Lawler, E., Yeh, S., Mahabal, A., Graham, M., Drake, A., et al. (2014). Immersive and collaborative data visualization using virtual reality platforms. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 609–614. IEEE.

[Dost et al., 2012] Dost, B., Zednik, J., Havskov, J., Willemann, R., and Bormann, P. (2012). Seismic Data Formats, Archival and Exchange. *New Manual of Seismological Observatory Practice 2 (NMSOP2)*.

[Eclipse, 2019a] Eclipse (2019a). Mosquitto. https://mosquitto.org/. Accessed in March 2019.

[Eclipse, 2019b] Eclipse (2019b). Paho mqtt. https://www.eclipse.org/paho/. Accessed in March 2019.

[Elmasri and Navathe, 2010] Elmasri, R. and Navathe, S. (2010). *Fundamentals of Database Systems*. Addison-Wesley Publishing Company.

[Grafana, 2019] Grafana (2019). Grafana. https://grafana.com/. Accessed in January 2019.

[Gunnarsdóttir et al., 2012] Gunnarsdóttir, E. et al. (2012). *The Earth's Magnetic Field*. PhD thesis.

[Gupta, 2015] Gupta, Y. (2015). *Kibana Essentials*. Packt Publishing Ltd.

[Guralnik and Srivastava, 1999] Guralnik, V. and Srivastava, J. (1999). Event Detection From Time Series Data. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 33–42. ACM.

[Hunkeler et al., 2008] Hunkeler, U., Truong, L., and Stanford-Clark, A. (2008). Mqtt-s—a publish/subscribe protocol for wireless sensor networks. In *2008 3rd International Conference on Communication Systems Software and Middleware and Workshops (COMSWARE'08)*, pages 791–798. IEEE.

[InfluxData, 2019] InfluxData (2019). Influxdb. https://www.influxdata.com/. Accessed in February 2019.

[IT, 2019] IT, S. (2019). Db-engines ranking of time series dbms. https://db-engines.com/en/ranking/time+series+dbms. Accessed in February 2019.

[Kibana, 2019] Kibana (2019). Kibana. https://www.elastic.co/pt/products/kibana. Accessed in January 2019.

[Kono and Roberts, 2002] Kono, M. and Roberts, P. (2002). Recent Geodynamo Simulations and Observations of the Geomagnetic Field. *Reviews of Geophysics*, 40(4):4–1.

[Krischer et al., 2015] Krischer, L., Megies, T., Barsch, R., Beyreuther, M., Lecocq, T., Caudron, C., and Wassermann, J. (2015). ObsPy: A Bridge for Seismology into the Scientific Python Ecosystem. *Computational Science & Discovery*, 8(1):014003.

[Lampkin et al., 2012] Lampkin, V., Leong, T., Olivera, L., Rawat, S., Subrahmanyam, N., Xiang, R., Kallas, G., Krishna, N., Fassmann, S., Keen, M., et al. (2012). *Building smarter planet solutions with mqtt and ibm websphere mq telemetry*. IBM Redbooks.

[Langenbach and Hansky, 2019] Langenbach, D. and Hansky, K. (2019). Simple Aurora Monitor (SAM). http://www.sam-magnetometer.net/. Accessed in December 2019.

[Lee et al., 2013] Lee, S., Kim, H., Hong, D.-k., and Ju, H. (2013). Correlation analysis of MQTT loss and delay according to qos level. In *The International Conference on Information Networking 2013 (ICOIN)*, pages 714–717. IEEE.

[Light et al., 2017] Light, R. et al. (2017). Mosquitto: server and client implementation of the mqtt protocol. *J. Open Source Software*, 2(13):265.

[Manandhar, 2017] Manandhar, S. (2017). MQTT based communication in IOT. Master's thesis.

[Moura et al., 2014] Moura, R., Sant'Ovaia, H., Simão, B., Santos, C., Freitas, J., Teixeira, L., and Ferreira, R. (2014). IGUP and Nuclear Seismology in Portugal. *Comunicacoes Geologicas*, 101:361–364.

[Naqvi et al., 2017] Naqvi, S., Yfantidou, S., and Zimányi, E. (2017). Time Series Databases and InfluxDB.

[Nayak et al., 2013] Nayak, A., Poriya, A., and Poojary, D. (2013). Type of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*, 5(4):16–19.

[ObsPy, 2017] ObsPy (2017). Obspy. https://github.com/obspy/obspy/wiki. Accessed in December 2019.

[Pintore et al., 2005] Pintore, S., Quintiliani, M., and Franceschi, D. (2005). Teseo: A vectoriser of historical seismograms. *Computers & Geosciences*, 31(10):1277–1285.

[Raykova and Nikolova, 2000] Raykova, R. and Nikolova, S. (2000). Review of Digital Formats for Exchange and Processing of Seismological Data. *Bulgarian Geophysical Journal*, 26:1–4.

[Reeve, 2010] Reeve, W. (2010). Geomagnetism Tutorial. *Reeve Observatory Anchorage, Alaska-USA*.

[Sanaboyina, 2016] Sanaboyina, T. (2016). Performance Evaluation of Time series Databases Based on Energy Consumption. Master's thesis.

[Shearer, 2009] Shearer, P. (2009). *Introduction to seismology*. Cambridge University Press.

[Silva et al., 2016] Silva, A., Breitenbücher, U., Képes, K., Kopp, O., and Leymann, F. (2016). Opentosca for iot: automating the deployment of iot applications based on the mosquitto message broker. In *Proceedings of the 6th International Conference on the Internet of Things*, pages 181–182. ACM.

[SoundCloud, 2019] SoundCloud (2019). Prometheus. https://prometheus.io/. Accessed in February 2019.

[StumbleUpon, 2019] StumbleUpon (2019). Opentsdb. http://opentsdb.net/. Accessed in February 2019.

[Torres et al., 2020] Torres, A., Rocha, A., and de Souza, J. (2020). Análise de desempenho de brokers MQTT em sistema de baixo custo. In *Anais do XV Workshop em Desempenho de Sistemas Computacionais e de Comunicação*, pages 47–58, Porto Alegre, RS, Brasil. SBC.

[Udias and Buforn, 2017] Udias, A. and Buforn, E. (2017). *Principles of Seismology*. Cambridge University Press.

[Ullman, 1997] Ullman, J. (1997). *A First Course in Database Systems*. Pearson Education India.

[Yip, 2019] Yip, M. (2019). Rapidjson. http://rapidjson.org/. Accessed in March 2019.