uc3m | Universidad Carlos III de Madrid


**Universidad Carlos III de Madrid**
**Escuela Politécnica Superior**


**Computer Science**
**Bachelors Thesis**

# Intrusion Analysis System using Big Data Techniques.


**Author:** Rafael Garcia Olmedo

**Tutors:** Francisco Javier Garcia Blas


**July, 2017**

Título: Sistema de Análisis de Intrusión mediante técnicas Big Data.

Autor: Rafael García Olmedo

Tutor: Francisco Javier García Blas

EL TRIBUNAL

Presidente: ______________________________

Vocal: ______________________________

Secretario: ______________________________

Realizado el acto de defensa y lectura del Trabajo de fin de grado el día __ de _______ de 20__ en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de

VOCAL

SECRETARIO PRESIDENTE

# Agradecimientos

Quisiera agradecer en primer lugar a mi tutor Francisco Javier García Blas por toda la ayuda que me ha proporcionado durante todo el proyecto y la paciencia que ha tenido conmigo.

En segundo lugar agradecer a mis compañeros Luis y Álvaro por hacer de la época universitaria un gran capitulo de mi vida y los proyectos y exámenes más llevaderos.

Por supuesto agradecer a mis padres por la infinita paciencia que tienen en el dia a dia conmigo y por apoyarme siempre en todo lo que haga.

A la universidad por darme la oportunidad de formarme de cara al futuro y experiencas tan increíbles como lo fue el Erasmus.

# Index of Content

**Agradecimientos** .......... 3

**Index of Content** .......... 4

**Index of Figures** .......... 7

**Index of Tables** .......... 9

**1. Introduction** .......... 11
1.1. Motivation .......... 11
1.2. Objectives .......... 11
1.3. Architecture of the application .......... 12
1.4. Document structure .......... 12
1.5. Acronyms .......... 13

**2. State of the Art** .......... 14
2.1. Distributed Systems .......... 14
*2.1.1. Characteristics of distributed systems* .......... 14
*2.1.2. Distributed system vs Parallel system* .......... 14
*2.1.3. Main architectures* .......... 14
*2.1.4. Clusters* .......... 15
2.2. Big Data .......... 16
*2.2.1. Characteristics of Big Data* .......... 16
*2.2.2. The problem of Big Data* .......... 17
2.2.3. Architectures .......... 17
2.2.3.1. Store and Search .......... 17
2.2.3.2. Store and Scale .......... 18
2.2.3.3. Scale and Search .......... 18
2.3. Apache Spark .......... 19
*2.3.1. Resilent Distributed Dataset (RDD's)* .......... 20
*2.3.2. Parallel operations* .......... 20
*2.3.3. Shared variables* .......... 21
*2.3.4. Spark architecture* .......... 21
2.4. Monitoring systems .......... 21
2.5. Security threats .......... 22
*2.5.1. Security requirements* .......... 22
*2.5.2. Threat classification (Microsoft)* .......... 23
*2.5.3. Threat classification according to the nature of the attack* .......... 23
2.6. Hadoop .......... 23
*2.6.1. Hadoop Distributed File System* .......... 24
2.6.1.1. HDFS Architecture .......... 24
2.6.1.2. NameNode .......... 25
2.6.1.3. Datanodes .......... 25
2.6.1.4. HDFS Client .......... 26
2.6.1.5. Replication .......... 26
*2.6.2. Apache Zookeeper* .......... 26
2.6.2.1. Zookeeper architecture .......... 26
2.7. Apache Hadoop Yarn .......... 27

2.8. Apache Zeppelin ..... 28
2.9. Similar applications ..... 28
*2.9.1. Loggly ..... 29*
*2.9.2. Splunk ..... 29*
*2.9.3. GoAccess ..... 30*
*2.9.4. Logz.io ..... 31*

**3. Analysis of the application ..... 31**
3.1. Overview of the application ..... 31
3.2. Use cases ..... 32
3.3. System requirements ..... 33
*3.3.1. Functional requirements ..... 34*
*3.3.2. Access requirements ..... 37*
*3.3.3. Software requirements ..... 39*

**4. Installation manual for the different technologies ..... 41**
4.1. Apache Spark Installation ..... 41
4.2. HDFS installation ..... 43
4.3. Zookeeper installation ..... 45
4.4. Apache Yarn Configuration ..... 46
4.5. Alternatives ..... 47
*4.5.1. Cloudera ..... 47*
*4.5.2. Hortonworks ..... 48*

**5. Development environment characteristics ..... 49**
5.1. Topology of the cluster ..... 49
5.2. Single machine characteristics ..... 50
*5.2.1. Compute 1-2, 1-3 and 1-7 machines ..... 50*
*5.2.2. Rest of Compute-1-* machines ..... 50*
*5.2.3. Compute-3-* machines ..... 50*
*5.2.4. Compute-5-* machines ..... 51*
*5.2.5. Compute-6-* machines ..... 51*
*5.2.6. Compute-7-* machines ..... 51*
*5.2.7. Compute-8-* machines ..... 51*
*5.2.8. Compute-9-1 and 9-3 machines ..... 51*
*5.2.9. Compute-9-2 machine ..... 52*
*5.2.10. Compute-10-* machines ..... 52*
*5.2.11. Compute-11-* machines ..... 52*
*5.2.11. Storage-1 machine ..... 52*
5.3. Aggregated capabilities ..... 52

**6. Implementation of the application ..... 53**
6.1. Server application ..... 53
*6.1.1. Communication protocol ..... 53*
*6.1.2. Implementation ..... 54*
6.2. Client application ..... 58
*6.2.1. Keyconcepts ..... 59*
*6.2.2. Implementation of the Spark Streaming application ..... 60*
*6.2.2.1. Case classes ..... 60*
*6.2.2.2. Main function ..... 61*
6.3. Visualization application ..... 64
*6.3.1. Loading of data ..... 65*
*6.3.2. Invalid connections per day ..... 66*
*6.3.3. Invalid connections per IP ..... 66*
*6.3.4. Invalid users used to authenticate ..... 66*

*6.3.5. Existing users failing to authenticate* ... 67
*6.3.6. Authentication failures per day* ... 67
*6.3.7. Top 10 IP's that tried to authenticate as root* ... 68
*6.3.8. Number of IP's banned by the fail2ban application* ... 68
*6.3.9. Actual IP's that were banned* ... 68
*6.3.10. DNS Lookup's per date* ... 69
*6.3.11. IP's performing DNS Lookup* ... 69
*6.3.12. Disk writes per device and time* ... 70
*6.3.13. Time-series graph for write operations on devices sdf and sdf1* ... 71
*6.3.13. Time-series graph for read operations on devices sdf and sdf1* ... 71
*6.3.14. Time-series graph for overall I/O time on devices sdf and sdf1* ... 71
*6.3.14. Time-series graph for available RAM memory* ... 72
*6.3.14. Time-series graph for free RAM memory* ... 72
*6.3.14. Time-series graph for buffered RAM memory* ... 72
*6.3.14. Time-series graph for cached RAM memory* ... 73

**7. Evaluation ... 74**
7.1. Window duration ... 74
7.2. Conclusions of the evaluation ... 75

**8. Project planning ... 76**
8.1. Initial planning ... 76
8.2. Real planning ... 76

**9. Socioeconomic Environment ... 78**
9.1. Socioeconomic impact ... 78
9.2. Budget ... 78
*9.2.1. Human resources budget* ... 78
*9.2.2. Hardware budget* ... 79
*9.2.3. Software budget* ... 79
*9.2.4. Consumables budget* ... 79
*9.2.5. Overall budget* ... 79

**10. Regulatory framework ... 80**
10.1. Legal aspects ... 80
10.2. Technical standards ... 81
10.3. Intellectual property ... 81

**11. Conclusions and further work ... 82**
11.1. Conclusions ... 82
11.2. Further work ... 82

**11. References and bibliography ... 83**

# Index of Figures

Figure 1 Store and Search architecture. 18
Figure 2 Store and Scale architecture. 18
Figure 3 Scale and Search architecture. 19
Figure 4 Spark architecture [4]. 21
Figure 5 Hadoop ecosystem [7]. 24
Figure 6 HDFS architecture [8]. 25
Figure 7 ZooKeeper architecture [9]. 26
Figure 8 Loggly dashboard [12]. 29
Figure 9 Splunk dashboard [14]. 30
Figure 10 GoAccess dashboard [17]. 30
Figure 11 Logz.io interface [20]. 31
Figure 12. Detailed architecture of the application 32
Figure 13 Application usage use case. 33
Figure 14 Java verification. 41
Figure 15 Ubuntu update. 41
Figure 16 Install default JRE. 41
Figure 17 Install default JDK. 41
Figure 18 Download and installation of Scala. 42
Figure 19 Download and installation of sbt. 42
Figure 20 Download and installation of Spark. 42
Figure 21 Spark shell. 43
Figure 22 Installation of SSH. 43
Figure 23 Downloading of Hadoop. 43
Figure 24 Accessing core-site.xml. 43
Figure 25 Core-site.xml configuration. 44
Figure 26 HDFS-site.xml configuration. 44
Figure 27 Namenode formatting. 44
Figure 28 HDFS service starting. 44
Figure 29 HDFS service verification. 45
Figure 30 Downloading of zookeeper. 45
Figure 31 Zookeeper configuration file. 45
Figure 32 Creation of the data directory for Zookeeper. 46
Figure 33 Adding desired ID to Zookeeper data. 46
Figure 34 Starting Zookeeper service. 46
Figure 35 Yarn-site.xml configuration file example. 46
Figure 36 CDH architecture[24]. 47
Figure 37 HDP architecture[27]. 48
Figure 38 Cluster topology. 49
Figure 39 Communication protocol. 54

Figure 40 Socket addresses. 55
Figure 41 Socket creation + tweaking. 55
Figure 42 Binding of the socket. 55
Figure 43 Listen + Accept functions. 56
Figure 44 Arguments to the threads function. 56
Figure 45: Identifier addition. 57
Figure 46 Dynamic allocation, copying and sending of the desired file. 57
Figure 47 Spark Context [33]. 59
Figure 48 Case class creation. 61
Figure 49 Initialization of the application. 61
Figure 50 Server connection call. 62
Figure 51 Filtering and mapping of fail2ban.log data. 62
Figure 52 Filtering of auth.log data. 63
Figure 53 Mapping of the data to Case classes. 63
Figure 54 Filtering and Mapping of diskstats information. 63
Figure 55 Filtering and Mapping of meminfo data. 64
Figure 56 Writiing to HDFS + starting of the application. 64
Figure 57 Applications on the cluster. 65
Figure 58 Data loading. 65
Figure 59 Invalid connections per day. 66
Figure 60 Invalid connections per IP. 66
Figure 61 Invalid users. 67
Figure 62 Existing users that failed authentication. 67
Figure 63 Authentication failures per day. 67
Figure 64 Top 10 IP's attacking root user. 68
Figure 65 Banned IP's per day. 68
Figure 66 Banned IP's. 69
Figure 67 DNS Lookup per day. 69
Figure 68 IP's performing DNS Lookup. 70
Figure 69 Disk writes per device and time. 70
Figure 70 Write time-series graph. 71
Figure 71 Read time-series graph. 71
Figure 72 I/O time time-series graph. 72
Figure 73 Available RAM time-series graph. 72
Figure 74 Free RAM time-series graph. 72
Figure 75 Buffered RAM time-series graph. 73
Figure 76 Cached RAM time-series graph. 73
Figure 77 Evaluation of the application. 74
Figure 78 Gantt diagram for the initial planning. 76
Figure 79 Gantt chart for the real planning of the project. 77

# Index of Tables

Table 1 System requirements schema. 33
Table 2 Functional requirement 1. 34
Table 3 Functional requirement 2. 34
Table 4 Functional requirement 3. 34
Table 5 Functional requirement 4. 35
Table 4 Functional requirement 5. 35
Table 5 Functional requirement 6. 35
Table 6 Functional requirement 7. 35
Table 7 Functional requirement 8. 35
Table 8 Functional requirement 9. 36
Table 9 Functional requirement 10. 36
Table 10 Functional requirement 11. 36
Table 11 Functional requirement 12. 36
Table 12 Functional requirement 13. 37
Table 13 Functional requirement 14. 37
Table 14 Functional requirement 15. 37
Table 15 Functional requirement 16. 37
Table 16 Access requirement 1. 37
Table 17 Access requirement 2. 38
Table 18 Access requirement 3. 38
Table 21 Access requirement 4. 38
Table 22 Access requirement 5. 38
Table 19 Access requirement 6. 38
Table 20 Software requirement 1. 39
Table 21 Software requirement 2. 39
Table 22 Software requirement 3. 39
Table 23 Software requirement 4. 39
Table 24 Software requirement 5. 40
Table 25 Software requirement 6. 40
Table 26 Software requirement 7. 40
Table 27 Machine characteristics schema. 50
Table 28 Compute 1-2, 1-3 and 1-7 characteristics. 50
Table 29 Rest of Compute 1-* characteristics 50
Table 30 Compute 3-* characteristics. 50
Table 31 Compute 5-* characteristics. 51
Table 32 Compute 6-* characteristics. 51
Table 33 Compute 7-* characteristics. 51
Table 34 Compute 8-* characteristics. 51
Table 35 Compute 9-1 and 9-3 characteristics. 51

Table 36 Compute 9-2 characteristics. ... 52
Table 37 Compute 10-* characteristics. ... 52
Table 38 Compute 11-* characteristics. ... 52
Table 39 Compute 11-* characteristics. ... 52
Table 40 Aggregated capabilities of the cluster. ... 53
Table 41 Initial Planning. ... 76
Table 42 Real planning. ... 77
Table 44 Human resources budget. ... 79
Table 45 Hardware budget. ... 79
Table 46 Software budget. ... 79
Table 47 Overall budget. ... 80

# 1. Introduction

In this first section of the report we will be giving a brief explanation of the development of the project as well as the different objectives that we would like to achieve.

## 1.1. Motivation

The main motivation behind the development of this project is to be able to use different emerging technologies to monitor and analyse different terminals in order to detect possible threats to the security of these systems. To do so we will be using Big Data techniques to capture and process the different sets of data received in order to detect if there has been some sort of security breach in the system being analysed. Nowadays Big Data is not only an emerging sector on the computing world but also relates many different concepts, from distributed systems (clusters) to visualization tools for this type of applications (*Zeppelin*, *ggplot* for *R-based* applications. *Kibana*), so the study of this type of technology is not only an opportunity to differentiate myself from other colleagues but to get into the Big Data world.

## 1.2. Objectives

The bachelor's thesis main objective is to develop a client-server application to monitor the activity of a system in order to detect possible intrusions by taking advantage of the capabilities of large-scale data processing engines such as *Apache Spark*, by having a flow of data sent to the possible clients and processing it using these techniques.

The objective of the thesis is to demonstrate the knowledge acquired on this type of technologies by explaining what was accomplished and how it was accomplished.

In order to complete the main objective of the thesis, the following smaller objectives are to be followed:

- Provide background on the selected technologies and justification on why they were chosen.
- Analysis of similar products available on the market.
- Analysis of the development environment of the application.
- Implementation of a *C-based* server that will send information to our client application.
- Implementation of an *Apache Spark* client application that connects to the *C-server* and cleans all the information and stores it in a distributed environment for latter visualization.

- Implementation of a visualization application using *Apache Zeppelin* to provide results of the application.

## 1.3. Architecture of the application

The client-server application is separated into three distinguishable parts:

- Server: The server for this application is a typical multithreaded C server, which will be reading information regarding security of the system (different types of logs) and also some system information.. This multithreaded server will be creating one thread per file being read and sent to the client.

- Client: The client side of the application consists of an *Apache Spark Streaming* client connecting to our implemented C server and receiving the different data sets, splitting them and processing them. After the processing of the received data, the apache Spark Client will be writing the final information to a distributed file system (*Hadoop Distributed File System*) for it to be stored in a distributed manner, replicated, and handled easily.

- Visualization of the application: Once the data has been correctly analysed and stored into our distributed file system, we will have another application called *Zeppelin* which will be the one in charge of reading the stored data from *HDFS* and displaying different graphs to visualize the effectiveness of the overall application.

## 1.4. Document structure

The document is structured in the following parts:

- 2. State of the Art: This section covers all the necessary background information related to the development of the project. The final section is a series of similar technologies to the one being developed.

- 3. Analysis of the application: This section includes the analysis and design of the application being developed, including use cases and requirements of the application.

- 4. Installation manual for the different technologies: This section is a manual covering the step-by-step installation of the different technologies on a machine.

- 5. Development environment characteristics: This section analyses the environment in which the application was carried out (Tucan cluster).

- 6. Implementation of the application: This section explains the actual implementation of the application.

- 7. Evaluation: This section covers the evaluation plan carried out to test the application.

- 8. Project planning: This section covers the planning performed to carry out the project, both the initial planning and the real planning.

- 9. Socioeconomic Environment: This section covers the impact on social and economic fields as well as the budget for the project.

- 10. Regulatory framework: This section covers the legal aspects concerning the application as well as programming standards for the different technologies used.

- 11. Conclusions and further work: This section contains the conclusions of the project and the future lines of work to improve the application.

### 1.5. Acronyms

This section provides a table with the acronyms used through the document.

| Acronym | Meaning |
|---|---|
| **HDFS** | Hadoop Distributed File System |
| **OS** | Operating System |
| **JVM** | Java Virtual Machine |
| **DAG** | Directed Acyclic Graph |
| **HA** | High Availability |
| **SPF** | Single Point of Failure |
| **RDD** | Resilient Distributed Dataset |
| **RAM** | Random Access Memory |
| **DNS** | Domain Name System |
| **I/O** | Input / Output |
| **kB, GB** | Kilobyte, gigabyte |
| **Gbps** | Gigabyte per second |

# 2. State of the Art

In order to understand the objective of this project it is necessary to give a brief explanation of the main technologies being used during the development of it.

## 2.1. Distributed Systems

We have many definitions for distributed systems, one of these could be the one defined by Tanenbaum : *"A collection of independent computers that appear to the users of the system as a single computer"* [1].

### 2.1.1. Characteristics of distributed systems

The characteristics of distributed systems are:

- Concurrency of components: as the previous definition said we have a set of computers that aggregate its hardware components so that we can use them as if it was a single one.

- Lack of a global clock: each of the computers has their own internal clock and synchronization among them is necessary for correct behaviour of the distributed programs that will be run along our system.

- Independent failure components: each of the components of the different machines is subtle to failure (in distributed systems the probability of a component failing is much higher than in conventional systems) and it's the designer's job to handle these failures.

### 2.1.2. Distributed system vs Parallel system

We must differentiate between distributed systems and parallel systems as these two terms often tend to be confused and overlapped. Whereas parallel systems communicate through a common memory space, distributed systems communicate through message passing between the different machines in our system.

### 2.1.3. Main architectures

There are various types of software and hardware architectures used for distributed systems but these usually fall into one of the following:

- Client-Server: architecture in which clients contact the server for data which will be then format and display it to the users. Input at the client side is sent back to the server when it represents a permanent change.

- Three-tier: architecture in which the client side of the application is moved to a middle tier so that stateless clients may be used. This simplifies deployment of the application. Most of the web applications are three-tier based.

- N-tier: architecture similar to three-tier that extends it to forward its requests to other enterprise services. Highly used for application servers.

- Peer-to-peer: architectures in which there are no privileged components, all machines have the same responsibilities and are interconnected among them. Machines in this type of system are known as peers and can act both as server and client. Torrent applications use this type of architecture.

### 2.1.4. Clusters

Cluster is a word used to refer to a group of computers that are interconnected between them and appear to be a single machine. Clusters are usually interconnected with very fast LAN networks, and they appeared because the computing power of single machines got obsolete and their growth (as described by Amdahl's law [2]) was limited, so in order to gain processing power computer clusters were created. Clusters have a series of keywords to describe their components; nodes of a cluster are the smallest unit of computation (a computer used as a server).

The main purposes of clusters are:

- Performance: clusters offer way higher processing power than that of single computers as each node in the cluster aggregates its hardware capabilities to that of the cluster (as its is seen as a single unit), and so they can provide better performance when dealing with problems that can not be solved using conventional sole computers.

- High availability: clusters are designed so that service (to the purpose they may have) can be served by using redundancy, so that if in a certain point in time a node is busy by other user, service to another one may be served. The main purpose of HA clusters is to avoid SPF on the system by the usage of redundancy.

- Load balancing: because nodes in a cluster share hardware resources, load balancing of an application can be done effectively distributing the computation among the different nodes. Load-balancing clusters objective is that all the nodes of the system cooperate to achieve a common goal (for example each of

the node is performing a different algorithm, and the final result will be aggregated.

- Scalability: clusters provide the best type of scalability (known as horizontal scalability) when the problem we are dealing with gets bigger. To scale our system all we have to do is add more nodes to our system (and generally computers that make up nodes of a cluster are computers which have low hardware characteristics that, when aggregated, generate high amount of resources). There is also the possibility of escalating vertically (meaning to add more resources to a node), but this would take to escalate every node of the cluster (not completely necessary) as the purpose of a cluster is to have similar hardware capabilities to prevent differences when being assigned to a node of the cluster, as this is done automatically by the cluster manager application.

There are more requirements when building up a cluster like the operating system we want the cluster to work under, this OS must be both multithreaded and allow multiple users to connect to it.

Clusters are in need of middleware (software operating between the OS and application running on the cluster) to provide a single system image, and thus when a user accesses the cluster it is perceived as a single very powerful computer. This middleware will also optimize the different processes on the system (load-balancing, fault tolerance…) and provide the scalability desired (this middle ware is in charge of handling resources of the system).

## 2.2. Big Data

Big data is a term used to refer to datasets that are so large or complex that the traditional processing applications are inadequate to deal with them. It is a field of study of the analysis, capturing, normalization, search, storage, transfer, visualization, querying and privatization of these large datasets.

The term is widely used to refer to different concepts and there is no general definition for what the term "Big data" covers but there are several hints and characteristics of applications that may take advantage of Big Data technologies.

### 2.2.1. Characteristics of Big Data

The main characteristics of Big Data application could be summarized in the following:

- Volume: The amount of stored and generated data.

- Variety: The type and nature of data, this information is vital so that people who will analyse the results of our application can better understand them.

- Velocity: The speed at which we generate and process data must meet the expected requirements for that application.

- Variability: Inconsistency of the dataset of our application might disturb processes computing this data.

- Veracity: The quality of captured data can vary greatly, affecting accurate analysis.

## 2.2.2. The problem of Big Data

Big data is a problem, as we cannot offer a solution that is efficient for every type of system. Depending on what we want to achieve our solution will adapt to one of the general types of architectures big data applications deal with. These architectures depend on three variables (we cannot offer the three at the same time):

- Store
- Scale
- Search.

### *2.2.3. Architectures*

Depending on which of the three variables we want to focus on, we have three types of architectures that are fit to solve the problem:

#### *2.2.3.1. Store and Search*

In this type of application we have a single user that will be accessing a distributed storage system and we will prioritize on storing and searching the data stored and not on scaling of the application. This can be seen on Figure 1 in which our application would only allow a single privileged user access to the system.

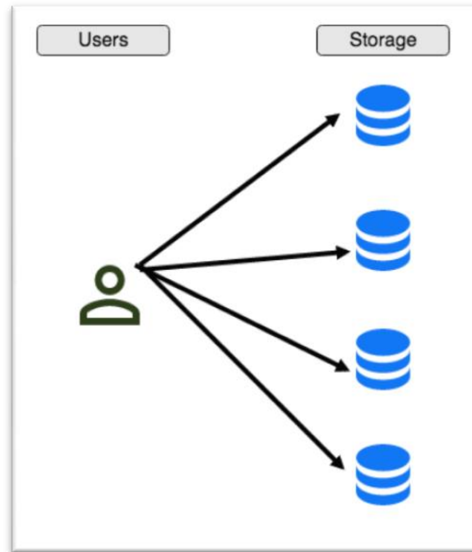


Figure 1 Store and Search architecture.

#### *2.2.3.2. Store and Scale*

In this type of application we have multiple users accessing a distributed storage system, we will prioritize on storing and scaling of the application and not on searches on the data stored. This can be seen on Figure 2, searches on the dataset will go slower as any user can access any storage system and the data we want to obtain might be in any of them.

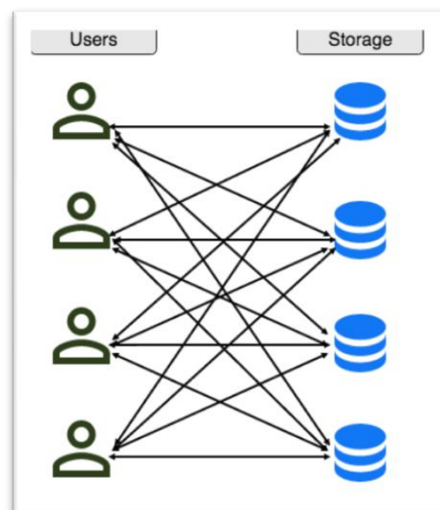


Figure 2 Store and Scale architecture.

#### *2.2.3.3. Scale and Search*

In this type of application we have multiple users accessing a single storage system, we will prioritize on scaling of the application and retrieval of data and not on storage capacity (as it is not distributed). This can be seen on Figure 3 in which we have multiple users accessing a central storage system.

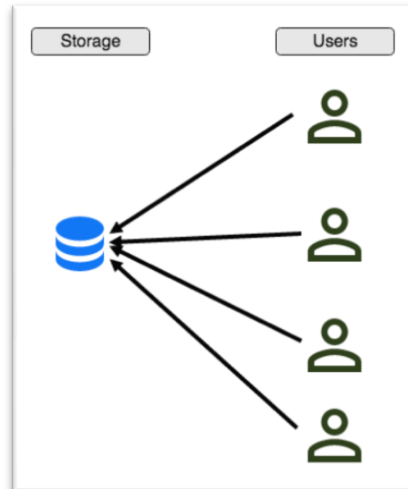


**Figure 3 Scale and Search architecture.**

All these architectures can also be referenced to the CAP theorem by Eric Brewer (Consistency, Availability and Partition tolerance), which states that in any distributed computing system we cannot assure simultaneously these three characteristics.

- Store and Search → provides consistency and partition tolerance (does not provide availability as only one user is able to access the application).

- Store and Scale → provides availability and partition tolerance (as at a time t data among users might not be consistent if a change by user A has been made on the storage $S_1$ and user B might not see this modification on the data).

- Scale and Search → provides consistency and availability (no partition tolerance as it is not distributed).

## 2.3. Apache Spark

This section will provide some of the key concepts of the Apache Spark [3] engine for distributed processing.

From now on we will refer to Apache Spark as Spark for simplicity.

Spark is an open source computing framework oriented to cluster computing and large-scale data processing. We will explain the main functionalities of it as well as why it was chosen as the computing framework for this project. Spark is a processing engine that can work with different programming languages (mostly *Scala*, *Java* and *Python*). For the development of the application we chose *Scala* as it is an emerging programming language and highly used in the Big Data environment, as well as for the

many features that make it a very good programing language (data types inferring, map-reduce operations, case class implementations…).

The programming model is functional oriented (treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data) and is composed of a *driver program* that invokes parallel operations on the data sets applying a function that will be distributed along the infrastructure in which we will deploy the application. The *driver program* generates for each application a directed acyclic graph representing each of the steps of the application.

Spark has two main options for data processing depending on the speed and the availability of the data being processed. We have either batch processing or stream processing. In Spark Streaming, processing is not really what we understand as stream processing but rather micro batch processing. This is because data is not processed record-by-record as in other streaming applications such as *Apache Storm* or *SQLstream Blaze*, but instead it processes in small batches of data that are aggregated and treated as a single processing unit (called *Dstreams* for the Spark engine).

Spark's programming model is composed of three main elements:

### 2.3.1. Resilent Distributed Dataset (RDD's)

They are the main data structure in the programming model of Spark, a definition would be a read-only set of data items that will be distributed along the infrastructure of our application. Their main advantage is that they support fault-tolerance through a capability known as lineage (the set of operations that produced an *RDD* through the life-cycle of the application). *RDD's* are immutable which means that in order to alter an existing *RDD* Spark will generate a new one with the desired transformation.

### 2.3.2. Parallel operations

Spark supports different type of operations to be performed on an *RDD* that will generate a new *RDD*. We have two different types of operations that can be invoked over our *RDD's*:

- Transformations: these operations create a new *RDD* from an existing one applying a function (map, filter…). These transformations are lazy and do not really occur until an "action" operation is performed over that *RDD*.

- Actions: these operations return a value to the *driver program* after applying a function to the RDD (reduce, collect…).

### 2.3.3. Shared variables

Spark supports two types of limited shared variables:

- Broadcast variables: this type of variable is a read-only variable that is cached in each of the machines of the system. It is mainly used to create copies of large datasets to reduce the overhead in communication generated by sending these datasets when they are needed.

- Accumulators: this type of variable is a special type of variable to which we can only "add" by using associative and commutative operations. They are mainly used to implement counters. Only the *driver program* can read this value (the different computation nodes can not access its value).

### 2.3.4. Spark architecture

On Figure 4 we can observe the architecture of a spark application:

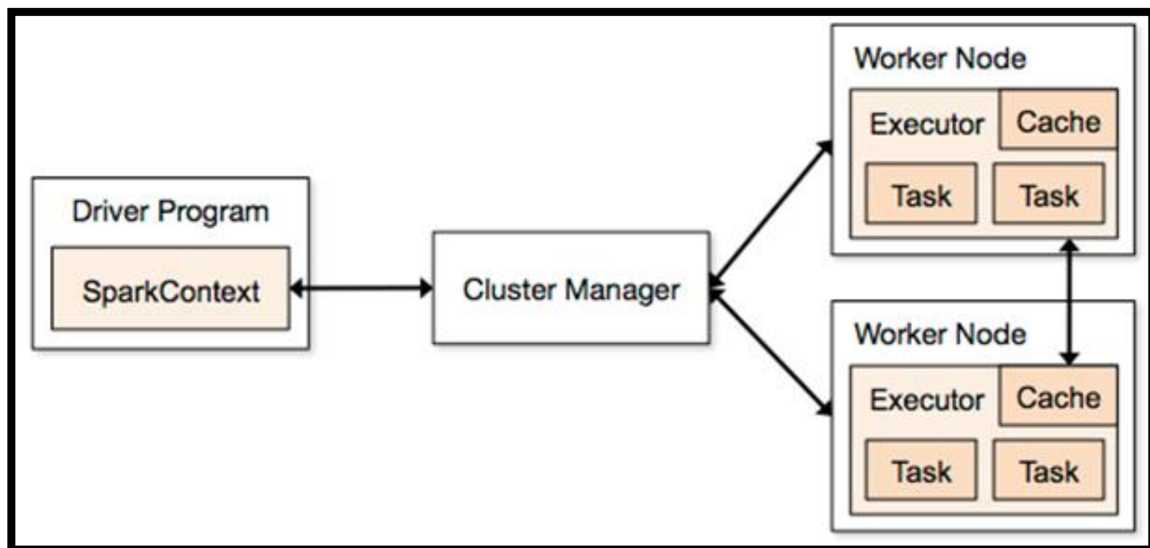


**Figure 4 Spark architecture [4].**

As we can observe the Spark architecture is composed of the spark context contained on the *driver program*, which interacts with the worker nodes that are the ones in charge of performing all the computation.

## 2.4. Monitoring systems

Monitoring systems are software application whose purpose is to monitor activity on a certain system so that if a failure arises in this system, its detection and recovery is easier for the system manager. We can also use monitoring systems as a security element in our system so that once we detect a possible threat, we can inform of it and keep a record of these threats and establish a backup policy.

The monitoring application needs information on what to monitor (elements we will be constantly checking) and how to monitor (if something is detected what to do).

Performance of these monitoring systems is crucial, and we have two main aspects on performance:

- Impact on the system domain: the monitoring application must never have a high impact on the performance of the system we are monitoring (must not degrade the functionality of the system being monitored).

- Efficient monitoring: the monitoring of the system must handle the monitoring goals in a timely manner, must meet the desired time requirements. This is strongly related to scalability (if we are monitoring a big system the monitoring application must scale with the application being monitored).

## 2.5. Security threats

The definition of a threat in computing is a possible danger that might take advantage of a vulnerability of our system and cause some harm to the application. The definition from ISO 27005(International Organization for Standardization) is:*" A potential cause of an incident, that may result in harm of systems and organization"* [5]. Threats might be either intentional or accidental.

### 2.5.1. Security requirements

Security in every system must meet the following requirements, being the three firsts ones the most important:

- Confidentiality: only authorized users can access the data.

- Integrity: Data must always be accurate and third parties must not modify them.

- Availability: Data must always be available to authorized users.

- Non-repudiation: Ensuring that the originators of messages cannot deny that they in fact sent the messages.

- Authentication: Ensuring that users are the persons they claim to be.

- Auditability: keeping track of everything that happens within the system so audition might be performed.

### 2.5.2. Threat classification (Microsoft)

Microsoft proposed a classification for possible threats according to five different categories, the DREAD risk assessment model [6]:

These five categories are the Damage that represents how bad an attack can be (the damage it can produce in our system). The Reproducibility of the attack, or how easy it is to reproduce that attack to our system. The Exploitability of the attack that stands for how much work does it take to launch that attack in our system. The amount of Affected users, if we are a big application will the attack affect to all the users of that application or will it affect to only a small portion of them. And finally the last category of the risk assessment model would be the Discoverability or how easy it is to discover that threat.

### 2.5.3. Threat classification according to the nature of the attack

Depending on which of the three main requirements of a secure system (Section 2.5.1) the attack is exploiting we have four different attack classifications:

- Interception: obtaining data that is being transferred. Attack on **confidentiality**.
- Interruption: denying access to a certain resource. Attack on **availability**.
- Modification: modification of data or packages. Attack on **confidentiality** and **integrity.**
- Fabrication: generation of data, it's a type of modification in which the modification is total and we generate a new set of data. Attack on **confidentiality** and **integrity.**

## 2.6. Hadoop

In order to explain the Hadoop Distributed File System (from now on referred to as HDFS) we need to give a brief introduction to what *Hadoop* is and which problems was it designed to solve. *Hadoop* is a collection of tools intended for both storing and processing of Big Data applications, it is part of the Apache Software Foundation and has a lot of different technologies on it (although we will only be using HDFS here).
On Figure 5 we can see part of the *Hadoop* ecosystem.

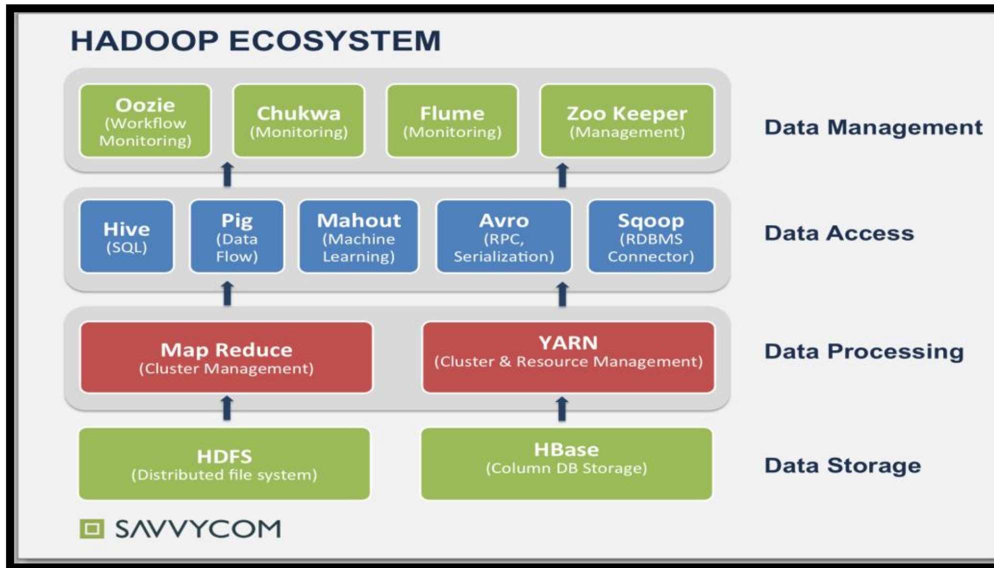


**Figure 5 Hadoop ecosystem [7].**

The core components of *Hadoop* are HDFS: a distributed fault tolerant and highly available storing system; *MapReduce*: the processing engine for distributed computing and *Apache YARN*: a cluster resource manager.

### 2.6.1. Hadoop Distributed File System

HDFS is a distributed file system developed by the *Hadoop* foundation intended to work over clusters to distribute data access and storage along them. It was written in *Java*, it is based on an abstraction called *Namespace*, which is a file and directories hierarchy. Files along HDFS are divided on big data blocks (128MB) and the *Namespace* is separated from the data. Apart from the *Namespace* we have other abstract entities on the HDFS environment, we have *Namenodes* and *Datanodes*, *Namenodes* are the ones in charge of storing in memory (RAM) the namespace hierarchy, metadata and blocks resident on HDFS; and *Datanodes* are the ones in charge of storing the File data blocks on disks local to each node on the cluster. As it is a fault tolerant system, data blocks are replicated along the cluster (typically a data block is replicated on 3 instances.

#### *2.6.1.1. HDFS Architecture*

On Figure 6 we can observe a diagram showing the architecture of how HDFS works:

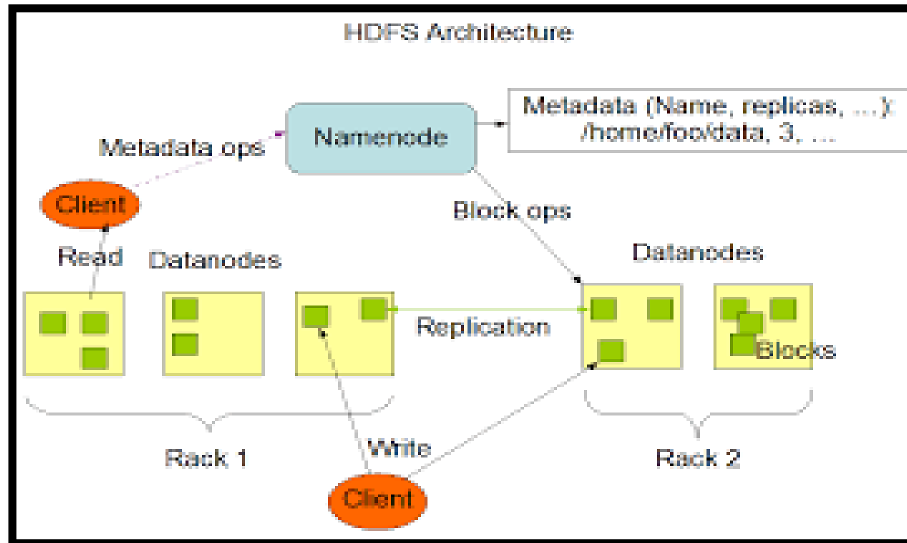


**Figure 6 HDFS architecture [8].**

As we can observe on the image there are 3 main components of the HDFS architecture, the client is the one accessing the HDFS in order to perform operations on the different files stored across the file system. When a client wants to access a file (for either reading or writing to HDFS), first it access the *Namenode* which is the one storing all the metadata necessary to find where a data block is being held, once the client has "queried" the *Namenode* to know where to find the data being requested, it will go to the specific *Datanode* and perform the operation.

### 2.6.1.2. NameNode

As stated previously *Namenodes* are the central piece of the HDFS, it contains all the information regarding the file systems hierarchy, the block manager and a list with all the *Datanodes* that are available. HDFS also has a secondary *Namenode* that stores checkpoints on the state of the system; the primary *Namenode* creates a log of transactions storing all the operations being made over the file system. The downside of the *Namenode* is that as it is the one storing all the information regarding on how to access data blocks if this one fails, then there is no way to access the data (Single point of failure), although we can recover it from checkpoints and it is recommended to keep other replicas of the *Namenode*.

### 2.6.1.3. Datanodes

Every *Datanode* once is created registers itself to the *Namenode* by sending a list of available blocks. In order to keep the consistency across the system, *Datanodes* send a heartbeat to the *Namenode* every 3 seconds and if the *Namenode* has not received this heartbeat from a *Datanode* then that node will be marked as unavailable (and thus all its available blocks).

#### *2.6.1.4. HDFS Client*

The HDFS client supports typical file system operations: create, read and write files, create and delegate directories, permission, replication, configuration…

#### *2.6.1.5. Replication*

Replication on the file system is handled by proximity and thus the three basic replicas will be created according to the following: 1st replica will be on the same node as the HDFS client, 2nd replica will be on a different rack from the 1st replica and the 3rd replica will be on the same rack as the 2nd one but in a different node. This is to maximize both latency when searching for data blocks and to assure fault tolerance.

### 2.6.2. Apache Zookeeper

In order to coordinate all the operations that must be done on the *Hadoop* environment we have a service named *Zookeeper* (needed for various reasons), this service allows distributed processes to coordinate through the hierarchical name space of *Hadoop*. The main purpose of *Zookeeper* is to automate all the synchronization and communication among the *Hadoop* ecosystem.

#### *2.6.2.1. Zookeeper architecture*

In order to explain the architecture of *Zookeeper* and how it works we can observe Figure 7.

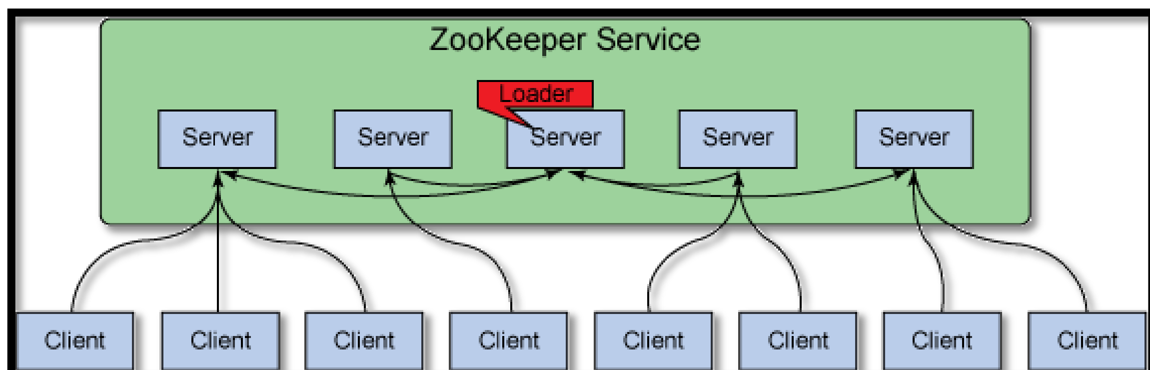


**Figure 7 ZooKeeper architecture [9].**

All the information in the system is replicated through the *Zookeeper* servers, and each one has a copy of the data. The leader is selected when the *Zookeeper* service is started and is the one in charge of handling the synchronization amongst the rest, to do so a client using the *Zookeeper* service may read any of the *Zookeeper* servers (as it is a non-blocking operation) but if a write operation is to be made this will be sent to the leader.

The main reason for using *Zookeeper* on our system is that in order to provide high-availability on HDFS the best way to do it is to use the *Zookeeper* service and abstract the communication when clients want to access HDFS.

## 2.7. Apache Hadoop Yarn

*Yarn* is a cluster resource administration tool, it can be seen as a distributed operating system intended to work over clusters. The *Yarn* application consists of a central *Resource Manager*, which has information on all the available resources along the cluster and a series of node managers that are in charge of handling and monitoring processing on each cluster node. *Yarn* makes automatic the handling of cluster resources, and there are some differences on how the application works from what we talked about in the *Spark* architecture, *Yarn* has three different execution modes and depending on each of them there are some slight differences on how it works. To make easier to comprehend this differences we need to recap a series of keywords regarding *Spark* applications:

- Application: the application is the final purpose of what we want to execute, it can either be a single shell application, a series of nodes communicating and collaborating to reach a final purpose, etc.

- Spark driver: the spark driver is the one in charge of creating and executing the spark context (representing the application), this driver is the one responsible of generating the DAG representing the application.

- Spark application master: this abstract entity is the one responsible of handling the requests of resources by each spark executor as well as finding the appropriate nodes on which to launch the spark application. Spark creates one application master per application.

- Spark executor: each instance of a JVM on a node that is performing computations for a single spark application. This must not be confused with cluster nodes as a single cluster node can have multiple Spark executors. Each executor on the system is in charge of performing a series of tasks in order to fulfil the purpose of the application.

- Task: a task represents the minimum unit of computation on a dataset, (each of the steps in the defined DAG can produce one or more tasks).

Below we give a brief explanation on the three different Yarn modes and their differences according to these keywords regarding Spark Yarn applications.

<u>Yarn Cluster Mode:</u> in this mode the spark driver program resides on the application master, who is isolated from the client, the application master requests resources to the YARN *ResourceManager*. In this mode, the YARN *NodeManager* starts executor processes, which together with the previously mentioned YARN *ResourceManager* are the permanent services that are "alive" during the application. In this Yarn mode there is

no support for spark shell (only self-contained applications may be launched in this mode).

Yarn Client Mode: in this mode the spark driver program resides on the client node, so output of the application will be visible to the client. As in the cluster mode, the application master handles resource requesting; also the same persistent services are "alive" during the application, and the YARN *NodeManager* is in charge of starting executor processes. Client mode does give support for the spark shell.

Spark Standalone: In this mode the client is responsible of managing everything of the application. It also provides support for spark shell.

## 2.8. Apache Zeppelin

As stated by its web page, *"Apache Zeppelin is a web-based notebook that enables interactive data analytics"* [10], to further understand the purpose of Zeppelin we need to define what is a web-based notebook.

A web-based notebook is an abstraction of how to work on a determined environment instead of working with interactive shells. It was first introduced by *iPython* as a web-based application in which to create *Python* interactive programs.

In the end Zeppelin is a tool to ingest, discover, analyse and visualize data on a simple common environment. Zeppelin supports many different interpreters (*Apache Spark*, *Python*, *R*, *Shell*, etc). It also has the possibility to add and create more interpreters to the system.

The main reason to be using Apache Zeppelin on our application is the fact that once we have properly saved our desired data, loading it into zeppelin and visualizing it is quite straightforward. Also it has some serious advantages, it's simplicity is one of them, the interface is integrated with easy technologies such as hive or *sparkSQL*. Again the variety of opportunities it gives with the plugin architecture regarding interpreters. The fact that in the same notebook you are able to use different programming languages is also a fact to take into consideration.

## 2.9. Similar applications

In this section we will be introducing a few applications available on the market whose functionality and purpose are similar to the one being developed. These applications are mainly log-management applications.

## 2.9.1. Loggly

This is a cloud-based logging management and analytics application, it records log data from any device and reports this data in a real-time management platform. Loggly [11] offers both free and paid plans, but the free plan has some serious limitations (only 1 user can acess the application at the same time, only 200 MB a day, 7 day retention of data…). On Figure 8 we can observe a view of the dashboard Loggly provides.

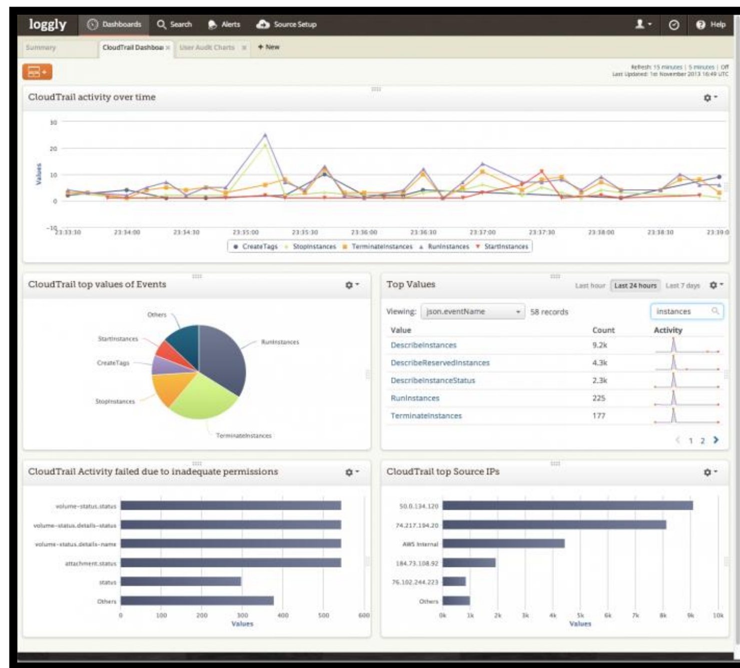


**Figure 8 Loggly dashboard [12].**

The main differences of this application with ours are (apart from it being more powerful) the Big Data technologies used to implement it (*Elasticsearch*, *Apache Lucene* and *Apache Kafka*).

## 2.9.2. Splunk

Splunk [13] is an application for log treatment, it searches, monitories and analyses machine-generated data. It is a very powerful application, allowing to generate graphs, dashboards, reports, alerts and visualizations over the log-based data. Splunk offers many different products going from those aiming to big companies to the ones for small IT infrastructures; Splunk also offers cloud-based products. On Figure 9 we can observe an example of the interface Splunk provides to its users.

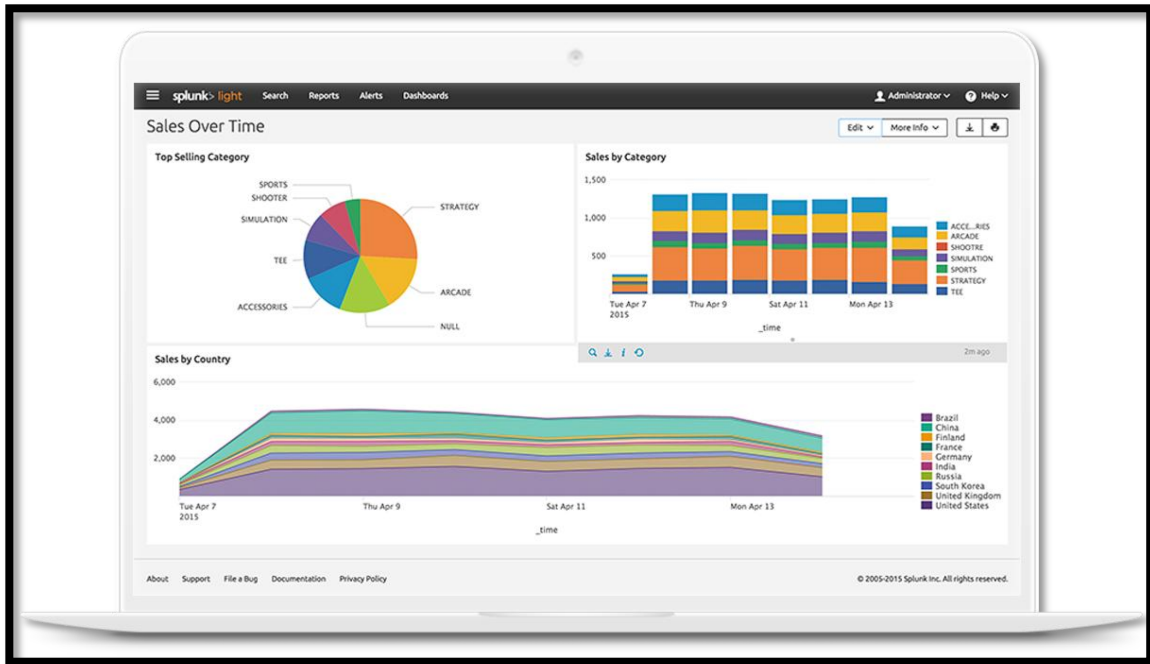


Figure 9 Splunk dashboard [14].

## 2.9.3. GoAccess

GoAccess [15] is an application entirely open-source and available on GitHub [16]. It is a log-analyser designed to be fast and terminal-based so that it does not require using the web-browser. It is a very powerful application for system analysers that are fluent with terminal commands and ssh. GoAccess also offers the possibility to generate reports in various formats (being the terminal output its default information source) like HTML, JSON or CSV. On Figure 10 below we can observe the dashboard output for the GoAccess application.

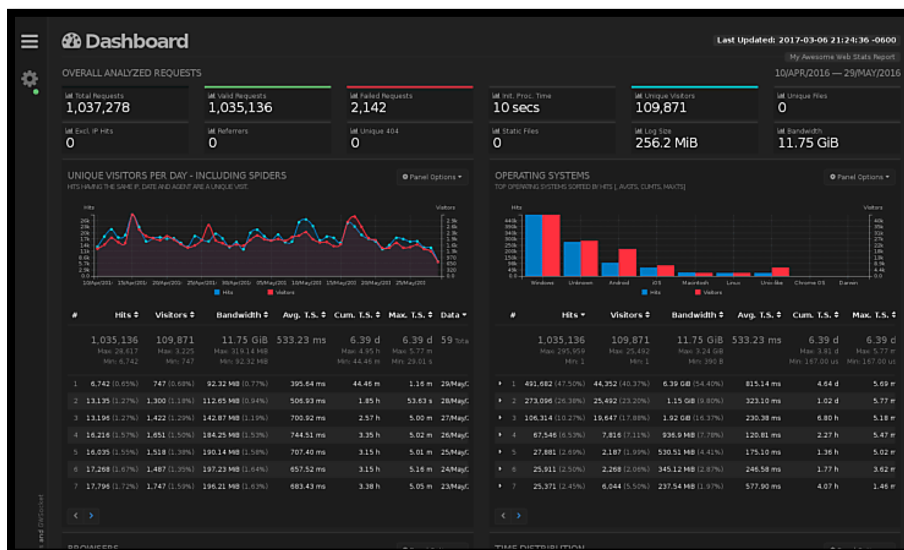


Figure 10 GoAccess dashboard [17].

GoAccess does not use Big Data technologies, it is entirely written in C; that would be the main difference between our application and GoAccess.

### 2.9.4. Logz.io

Logz.io [18] is another log-analysis software; it is based on Big Data technologies offered by the *Elastic Stack* [19], these technologies include, *ElasticSearch* (search engine acting as a NoSQL database), *Logstash* (processing pipeline log-oriented) and *Kibana* (visualization platform), all these technologies are easily integrated among themselves. It offers both paid and free plans (limited to 1GB of data and 3 day retention). On Figure 11 we can observe an example of the dashboard Logz.io offers.

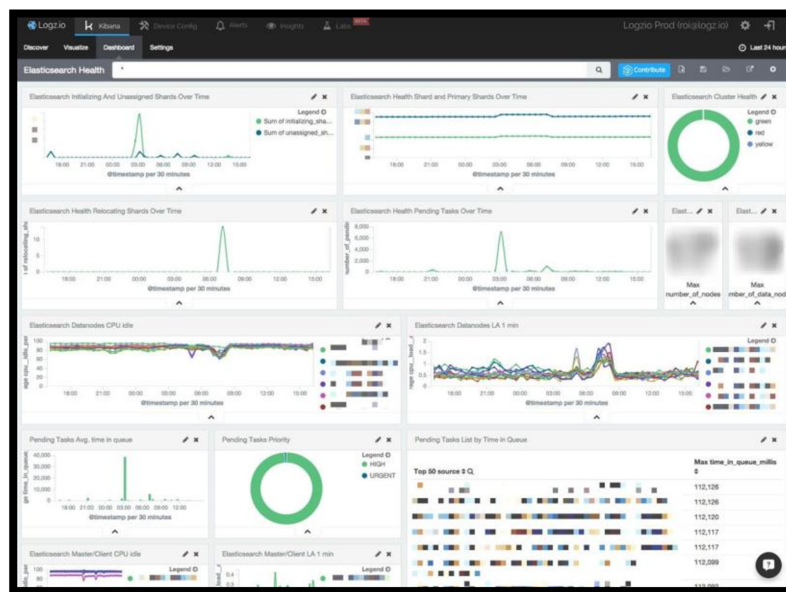


Figure 11 Logz.io interface [20].

# 3. Analysis of the application

In this section we will proceed with a previous analysis of the application, stating the desired functionality, use cases and requirements the application must fulfil. The objective of this section is to show the different components of the application in a more detailed way, to do so we will first give a brief glimpse of the application, then we will provide the use cases for the application, and finally we will show Tables following the format of Table 1 to show the different requirements of the application.

## 3.1. Overview of the application

This section aims to provide a small recap of what the applications final purpose is.

As stated before, this application consists on a monitoring system to detect possible threats to the system being monitored. The application works as a server-client application in which the server-side will be sending (as a stream of data) different log and system information to the client-side of the application. Then the client-side will filter that data and store it in a distributed file system (HDFS), finally we have a visualization application (Zeppelin) that will be the one displaying the gathered information.

On Figure 12 we can observe the overall architecture of the application, explanation of each the points:

- 1. Server application reads the files.
- 2. Server application sends data to the Spark application as a stream of data.
- 3. Spark application filters incoming stream of data and stores necessary information on HDFS.
- 4. Zeppelin runs over Spark.
- 5. Zeppelin loads data from HDFS and provides the visualization tools.

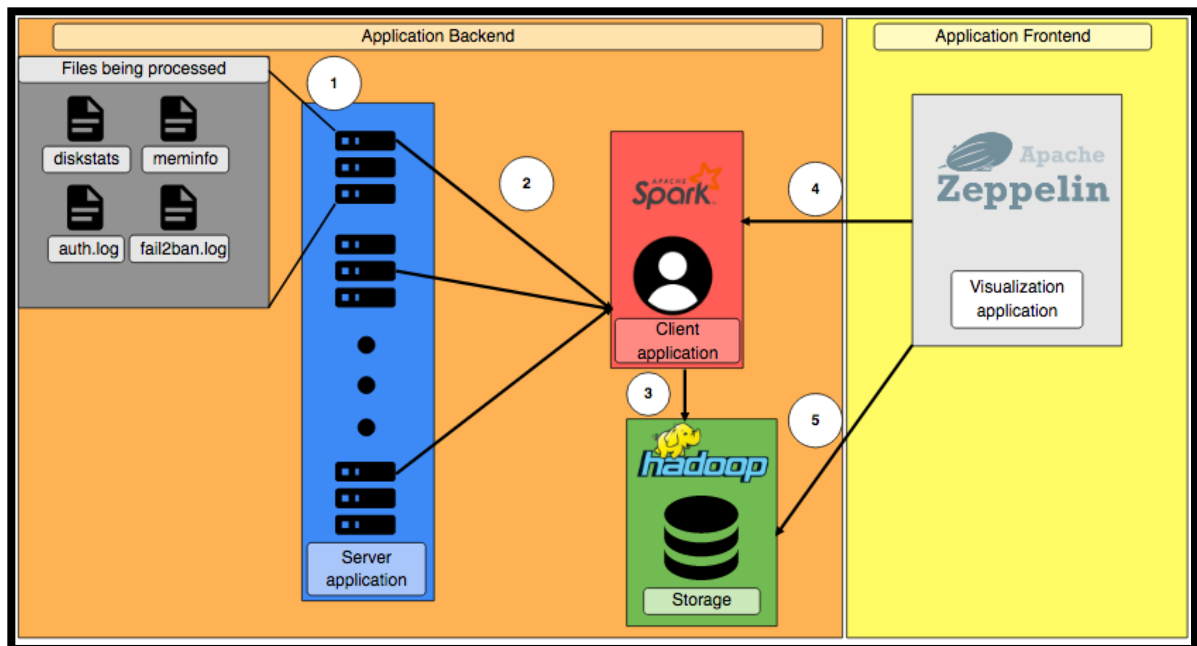


**Figure 12. Detailed architecture of the application**

## 3.2. Use cases

In this case we will use *UML use case diagrams* [21] to represent the different steps to be followed by a user to use the application.

As our application is a monitoring system application there is only one use case related to our software application, this use case can be seen on Figure 13.

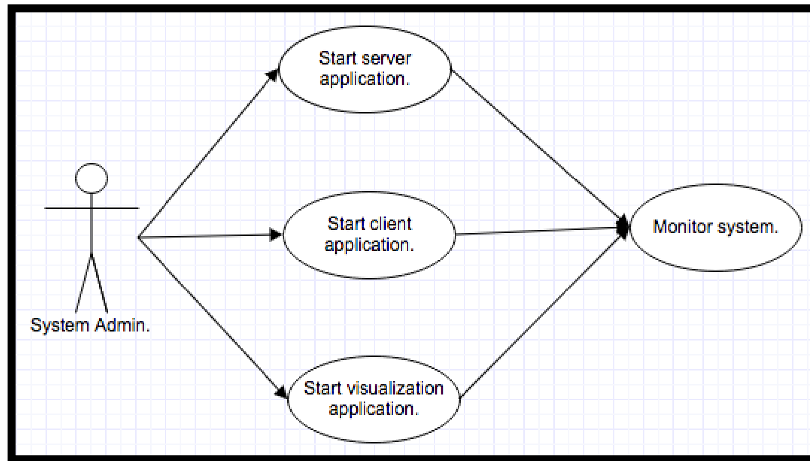


**Figure 13 Application usage use case.**

This is the only use case for our application; the system administrator is in charge of both having permissions over the required files, and starting the three main components of the application. Once the three elements are running the system administrator will be able to see in a real-time fashion the different graphs shown on the visualization application to monitor malicious activity on the system.

## 3.3. System requirements

The aim of this section is to provide all the requirements the application must fulfill for its proper implementation. All requirements will be provided on a table according to the one shown in Table 1.

| ID | |
|---|---|
| NAME | |
| DESCRIPTION | |
| PRIOTITY | |
| NECESSITY | |

**Table 1 System requirements schema.**

Where each field on the table represents the following:

- ID: Identifier of the requirement, it is a unique identifier for each of the requirements that will help the understanding of the type of requirement it represents, we have three different types of requirements:
    - FR-XX: this type of identifier matches functional requirements of the application, that is, the desired functionality for the different parts of the application.
    - AR-XX: these are access requirements; they represent rights that must be obtained by the user of the application so that it works properly.

- SR-XX: these identifiers are the ones representing software requirements, which include software that must be available, libraries, versions, etc.

- NAME: a brief name given to the requirement.
- DESCRIPTION: a detailed description of the requirement.
- PRIORITY: this value represents the priority of the requirement; its values can be three (HIGH, MEDIUM and LOW).
- NECESSITY: this value represents how necessary that requirement is for the correct functioning of the application, it also has three values (ESSENTIAL, DESIRABLE and OPTIONAL)

### 3.3.1. Functional requirements

This section contains the different functional requirements of the application.

| FR-01 | |
|---|---|
| **NAME** | Auth.log existence |
| **DESCRIPTION** | The file auth.log must exist on our system. |
| **PRIOTITY** | HIGH |
| **NECESSITY** | ESSENTIAL |

**Table 2 Functional requirement 1.**

| FR-02 | |
|---|---|
| **NAME** | Fail2ban.log existence |
| **DESCRIPTION** | The file fail2ban.log must exist on our system. |
| **PRIOTITY** | HIGH |
| **NECESSITY** | ESSENTIAL |

**Table 3 Functional requirement 2.**

| FR-03 | |
|---|---|
| **NAME** | diskstats existence |
| **DESCRIPTION** | The file diskstats must exist on our system. |
| **PRIOTITY** | HIGH |
| **NECESSITY** | ESSENTIAL |

**Table 4 Functional requirement 3.**

| FR-04 | |
|---|---|
| NAME | meminfo existence |
| DESCRIPTION | The file meminfo must exist on our system. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 5 Functional requirement 4.

| FR-05 | |
|---|---|
| NAME | Real-time application. |
| DESCRIPTION | The application must work on a real-time fashion. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 6 Functional requirement 5.

| FR-06 | |
|---|---|
| NAME | Accept connections. |
| DESCRIPTION | The server application must be able to support client connections. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 7 Functional requirement 6.

| FR-07 | |
|---|---|
| NAME | Accept multiple connections. |
| DESCRIPTION | The server application must support multiple clients connecting. |
| PRIOTITY | MEDIUM |
| NECESSITY | DESIRABLE |

Table 8 Functional requirement 7.

| FR-08 | |
|---|---|
| NAME | Infinite execution. |
| DESCRIPTION | The server application must be continuously sending data to the client application. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 9 Functional requirement 8.

| FR-09 | |
|---|---|
| NAME | Multithreading |
| DESCRIPTION | The server application must support multiple threads to run. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

**Table 10 Functional requirement 9.**

| FR-10 | |
|---|---|
| NAME | Filtering. |
| DESCRIPTION | The client application must filter the stream of data so that only required information is stored. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

**Table 11 Functional requirement 10.**

| FR-11 | |
|---|---|
| NAME | Storing. |
| DESCRIPTION | The client application must correctly store the filtered information for the visualization application to load them. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

**Table 12 Functional requirement 11.**

| FR-12 | |
|---|---|
| NAME | Distributed computing. |
| DESCRIPTION | The client application must perform its processes in a distributed fashion. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

**Table 13 Functional requirement 12.**

| FR-13 | |
|---|---|
| NAME | Independent storing. |
| DESCRIPTION | Each of the desired information will be stored on separate files according to the nature of the data. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 14 Functional requirement 13.

| FR-14 | |
|---|---|
| NAME | Loading of data. |
| DESCRIPTION | The visualization application must correctly load the data stored by the client application. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 15 Functional requirement 14.

| FR-15 | |
|---|---|
| NAME | Clearness. |
| DESCRIPTION | The visualization application must provide clear meaningful graphs to show the data. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 16 Functional requirement 15.

| FR-16 | |
|---|---|
| NAME | Replication. |
| DESCRIPTION | The storage of the data made by the client-application must be automatically replicated. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 17 Functional requirement 16.

### 3.3.2. Access requirements.

This section describes the different permissions and authority requirements that must be taken into account.

| AR-01 | |
|---|---|
| NAME | Access to development cluster. |
| DESCRIPTION | The user must be able to access (and authenticate) the cluster where the application is implanted. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 18 Access requirement 1.

| AR-02 | |
|---|---|
| NAME | Auth.log |
| DESCRIPTION | The user must have (at least) read permissions over the auth.log file. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 19 Access requirement 2.

| AR-03 | |
|---|---|
| NAME | Fail2ban.log |
| DESCRIPTION | The user must have (at least) read permissions over the fail2ban.log file. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 20 Access requirement 3.

| AR-04 | |
|---|---|
| NAME | diskstats |
| DESCRIPTION | The user must have (at least) read permissions over the diskstats file. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 21 Access requirement 4.

| AR-05 | |
|---|---|
| NAME | meminfo |
| DESCRIPTION | The user must have (at least) read permissions over the meminfo |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 22 Access requirement 5.

| AR-06 | |
|---|---|
| NAME | Stored data. |
| DESCRIPTION | The user must have both read and write permissions over the directory in which the filtered data stored by the client application writes. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 23 Access requirement 6.

### 3.3.3. Software requirements

This section will cover the different software requirements needed for the application to correctly function.

| SR-01 | |
|---|---|
| **NAME** | Java. |
| **DESCRIPTION** | The development environment must have Java installed with version 1.7.0_67 or higher. |
| **PRIOTITY** | HIGH |
| **NECESSITY** | ESSENTIAL |

Table 24 Software requirement 1.

| SR-02 | |
|---|---|
| **NAME** | Scala. |
| **DESCRIPTION** | The development environment must have Scala installed with version 2.11.7 or higher. |
| **PRIOTITY** | HIGH |
| **NECESSITY** | ESSENTIAL |

Table 25 Software requirement 2.

| SR-03 | |
|---|---|
| **NAME** | Sbt. |
| **DESCRIPTION** | The development environment must have sbt installed with version 0.13.13 or higher. |
| **PRIOTITY** | HIGH |
| **NECESSITY** | ESSENTIAL |

Table 26 Software requirement 3.

| SR-04 | |
|---|---|
| **NAME** | Apache Spark. |
| **DESCRIPTION** | The development environment must have Apache Spark installed with version 1.6.0 or higher. |
| **PRIOTITY** | HIGH |
| **NECESSITY** | ESSENTIAL |

Table 27 Software requirement 4.

| SR-05 | |
|---|---|
| NAME | Apache Yarn. |
| DESCRIPTION | The development environment must have Apache Yarn installed. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 28 Software requirement 5.

| SR-06 | |
|---|---|
| NAME | Apache Hadoop. |
| DESCRIPTION | The development environment must have Apache Hadoop installed with version 2.6.0-cdh5.11.1 or higher. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 29 Software requirement 6.

| SR-07 | |
|---|---|
| NAME | Apache Zeppelin. |
| DESCRIPTION | The development environment must have Apache Zeppelin installed with version or higher. |
| PRIOTITY | HIGH |
| NECESSITY | ESSENTIAL |

Table 30 Software requirement 7.

# 4. Installation manual for the different technologies

In this section we will be giving a manual on how to install and configure the different technologies we will be using on the development of the application. It will be divided by each technology (as they are all open source). The manual serves as a guide on how to install all the technologies on a new environment, but our application was in the end carried on the universities cluster (see Section 5).

## 4.1. Apache Spark Installation

In order to install apache Spark we need to have a series of pre-requisites:

1. Verify Java Installation: to work with apache spark we need to have java installed, (we recommend to have the latest java version), so first of all we should check we have java installed in our machine.

```
➜ TFG_install java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
```

Figure 14 Java verification.

If we do not have Java installed then simply run the following commands to get the default installation (although this will install Java 7 instead of 8):

a. Update Ubuntu:

```
➜ TFG_install sudo apt-get update
```

Figure 15 Ubuntu update.

b. Download default Java JRE:

```
➜ TFG_install sudo apt-get install default-jre
```

Figure 16 Install default JRE.

c. Download default Java JDK:

```
➜ TFG_install sudo apt-get install default-jdk
```

Figure 17 Install default JDK.

2. Downloading and installing Scala: in order to download and install Scala in our Debian-based machine run the following commands:

```
sudo apt-get remove scala-library scala
sudo wget www.scala-lang.org/files/archive/scala-2.10.4.deb
sudo dpkg -i scala-2.10.4.deb
sudo apt-get update
sudo apt-get install scala
```

Figure 18 Download and installation of Scala.

3. Downloading and installing sbt: sbt is an open source building tool for Java and Scala projects, its use in our project is to compile our Spark code (written in Scala). In order to install sbt in our Debian-based machine run the following commands:

```
wget http://scalasbt.artifactoryonline.com/scalasbt/sbt-native-packages/org/scala-sbt/sbt/0.12.4/sbt.deb
sudo dpkg -i sbt.deb
sudo apt-get update
sudo apt-get install sbt
```

Figure 19 Download and installation of sbt.

4. Downloading and installing Apache Spark: the final step is to download apache spark from its web page [22] and installing it, to do just run the following commands:

```
tar -xvzf spark-1.6.1-bin-hadoop2.4.tgz
sudo mv spark-1.6.1-bin-hadoop2.4 /opt

sudo ln spark-1.6.1-bin-hadoop2.4 spark
//Setting environment for Spark; ADD THE FOLLOWING LINE TO ~/.bashrc file, NEEDS SUDO PERMISSION
export PATH = $PATH:/opt/spark/1-bin-hadoop2

sudo source ~/.bashrc
```

Figure 20 Download and installation of Spark.

5. Finally verify that the installations has been correctly made by executing the spark-shell:

```
rgarcia@urraca:~$ spark-shell
WARNING: User-defined SPARK_HOME (/opt/cloudera/parcels/CDH-5.10.1-1.cdh5.10.1
0.10/lib/spark) overrides detected (/usr/lib/spark).
WARNING: Running spark-class from user-defined location.
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/

Using Scala version 2.10.5 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_121)
Type in expressions to have them evaluated.
```

**Figure 21 Spark shell.**

## 4.2. HDFS installation

In order to install HDFS we have to fulfil the following steps:

1. Installing ssh:

```
➜  TFG_install sudo su -
[sudo] password for utad:
utad-virtual-machine# apt-get install ssh
```

**Figure 22 Installation of SSH.**

2. Installing Hadoop: we will be installing hadoop on our */opt* directory:

```
➜ TFG_install cd /opt
➜ /opt wget http://ftp.cixug.es/apache/hadoop/common/hadoop-3.0.0-alpha1/hadoop
-3.0.0-alpha1.tar.gz
```

**Figure 23 Downloading of Hadoop.**

3. Configuration of HDFS: in order to configure HDFS we will need to modify the content of some configuration files:

   a. *Core-site.xml*: file that will configure where :

```
➜  /opt sudo ln -s hadoop-3.0.0-alpha1 hadoop
[sudo] password for utad:
➜  /opt cd hadoop/etc/hadoop
➜  hadoop vim core-site.xml
```

**Figure 24 Accessing core-site.xml.**

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

**Figure 25 Core-site.xml configuration.**

Where value will be where HDFS will be residing.

b. *Hdfs-site.xml*: file that will configure replication and data nodes and namenodes.

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.namenode.name.dir</name>
        <value>/var/dfs/my_namenode</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/var/dfs/my_datanode</value>
    </property>
</configuration>
```

**Figure 26 HDFS-site.xml configuration.**

4. Running HDFS: in order to be able to start using HDFS we have to run the necessary services:

   a. First of all we have to format the namenode, to do so run the following command (from the root directory of hadoop):

```
➜ hadoop sudo bin/hdfs namenode -format
```

**Figure 27 Namenode formatting.**

   b. Next we have to run the HDFS service, to do so:

```
➜ hadoop sudo sbin/start-dfs.sh
```

**Figure 28 HDFS service starting.**

5. Verifying the service is up: in order to finally demonstrate that both namenodes (primary and secondary) are up we run the following command:

```
➜  hadoop sudo jps
14594 Jps
14068 SecondaryNameNode
13710 NameNode
➜  hadoop
```

Figure 29 HDFS service verification.

Now we can proceed with the different commands HDFS supports.

## 4.3. Zookeeper installation

The following manual covers how to install and configure Zookeeper:

1. First of all we download and uncompress zookeeper.

```
➜  ~ cd /opt
➜  /opt  wget http://apache.rediris.es/zookeeper/stable/zookeeper-3.4.7.tar.gz
➜  /opt  ln -s zookeeper-3.4.7 zookeeper
➜  /opt sudo tar -xvzf zookeeper-3.4.9.tar.gz
```

Figure 30 Downloading of zookeeper.

2. Now we must configure the *zoo.cnf* file on the *"/conf "* directory of Zookeeper.

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/var/lib/zookeeper
# the port at which the clients will connect
clientPort=2181

server.1=localhost:2888:3888
```

Figure 31 Zookeeper configuration file.

It is important to note that server.(number) must always be an integer as we will have to refer to it in the next step.

3. This previously mentioned number must be included on the data directory as follows:

   a. First we create the following directory:

```
→ conf sudo mkdir /var/lib/zookeeper
```

Figure 32 Creation of the data directory for Zookeeper.

   b. Then we issue the following command to finish the configuration of zookeeper:

```
→ conf echo 1 > /var/lib/zookeeper/myid
```

Figure 33 Adding desired ID to Zookeeper data.

4. Starting Zookeeper: finally the last step is to lift the Zookeeper service, to do so:

```
→ zookeeper sudo bin/zkServer.sh start
[sudo] password for utad:
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper-3.4.9/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
```

Figure 34 Starting Zookeeper service.

## 4.4. Apache Yarn Configuration

Apache Yarn is a part of the bundle contained on the Hadoop download on Section 3.3, in order to configure Yarn we need to go to the *"/opt/hadoop/etc/hadoop"* directory and configure the *"yarn-site.xml"* according to our cluster topology. Below we can observe an example of *"yarn-site.xml"*:

```
<?xml version="1.0" encoding="UTF-8"?>

<!--Autogenerated by Cloudera Manager-->
<configuration>
  <property>
    <name>yarn.acl.enable</name>
    <value>true</value>
  </property>
  <property>
    <name>yarn.admin.acl</name>
    <value>*</value>
  </property>
  <property>
    <name>yarn.resourcemanager.address</name>
    <value>urraca.arcos.inf.uc3m.es:8032</value>
  </property>
  <property>
    <name>yarn.resourcemanager.admin.address</name>
    <value>urraca.arcos.inf.uc3m.es:8033</value>
  </property>
```

Figure 35 Yarn-site.xml configuration file example.

## 4.5. Alternatives

There are many alternatives to manually installing and configuring each of the technologies one by one, there are bundles offered by different companies that contain most of the Big Data related technologies, below we offer 2 different alternatives.

### 4.5.1. Cloudera

Cloudera [23] it's a company that offers different software bundles related to Apache Hadoop, (its what has been used on our system to get every component working) its main products are the following:

1. CDH: it's the Cloudera distribution of Apache Hadoop, it includes all the core Hadoop elements as well as other related projects (Spark, Impala, Kafka, etc). CDH offers a series of advantages (and its open source), flexibility (storing any type of data and various manners of processing), integration (easiness of get running a complete Hadoop platform), security (process and control of all the data on your environment), high availability, scalability and compatibility with other IT infrastructures. Below we can observe an image representing the CDH architecture.

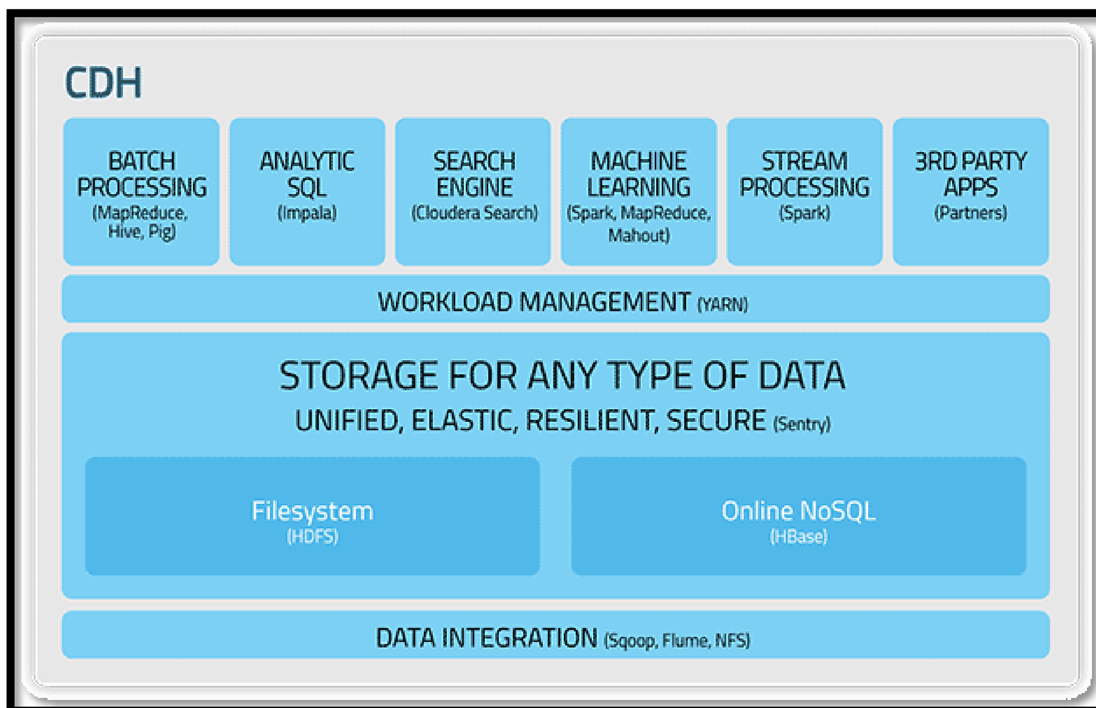


Figure 36 CDH architecture[24].

2. Apache Impala: its a parallel processing SQL engine for business intelligence and interactive analytics.

3. Cloudera Search: service that provides near real-time Access to data on Hadoop and HBase.

4. Cloudera Manager: another service offered by cloudera used to deploy, manage, monitor and diagnose CDH deployments.

5. Cloudera Navigator: its an end-to-end data management and security tool for the CDH platform.

Cloudera offers and installation guide that can be found in in [25]

### 4.5.2. Hortonworks

Hortonworks [26] is another software company similar to Cloudera that also offers software bundles related to the Apache Hadoop platform, its main products are:

1. Hortonworks Data Platform: HDP is similar to Cloudera's CDH, it offers the core components of Hadoop, its purpose os the processing, storing and analysis of data-driven applications. The image below shows its architecture (very similar to CDH):

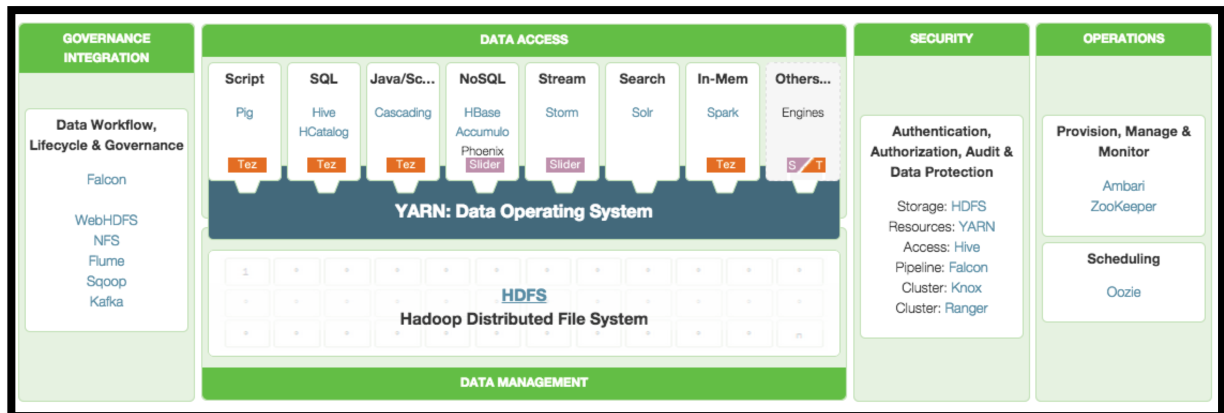


Figure 37 HDP architecture[27].

2. Hortonworks Data Flow: HDF is and application used for the analysis of the flow of data on our applications, it is intended for real-time applications and is integrated with HDP.

# 5. Development environment characteristics

In this section we will be explaining and analysing the environment in which the application was developed.

This application was carried on the Universidad Carlos III department of Computer Science and Engineering cluster *Tucan*. This cluster is a high performance computing purpose cluster, this means it is intended to perform data-parallel, Big Data and high-performance applications.

## 5.1. Topology of the cluster

Typical cluster topologies are rack-based; racks are standardized frame or enclosure for mounting multiple electronic equipment's [28]. In the Universities cluster we are in disposal of 3 racks organised according to Figure 38.

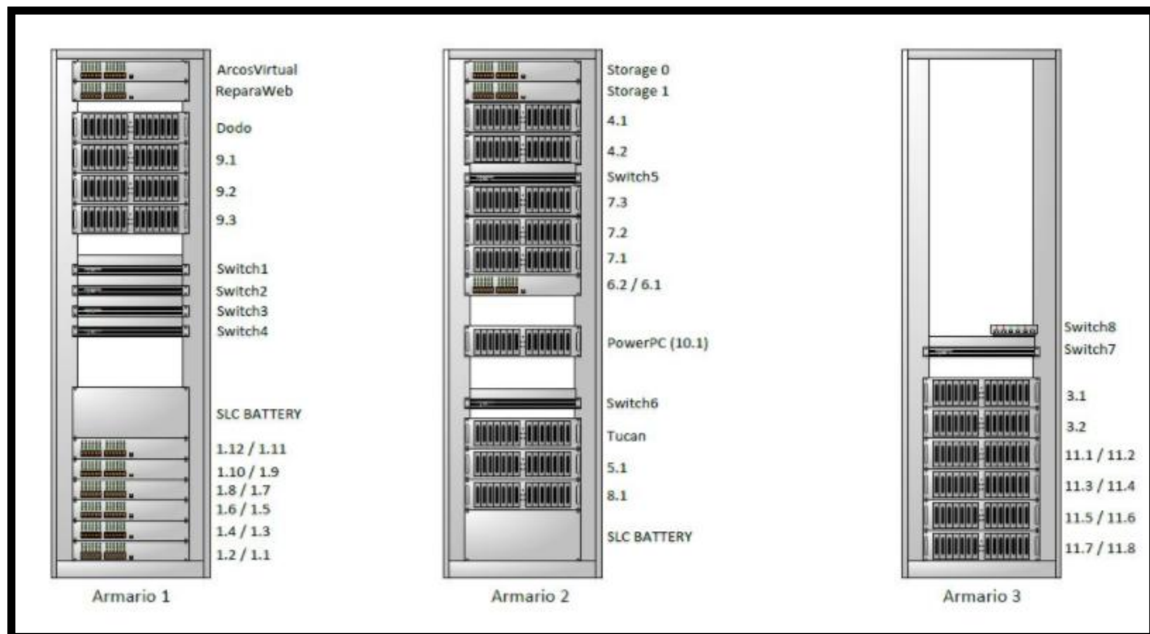


Figure 38 Cluster topology.

The clusters communication network is switch-based; all switches are physically connected via *u-link cables* to maximize speed among them. Maximum connection between nodes at the time is of 10Gbps (Switches 1, 5 and 7), which is a pretty high communication speed among nodes on a cluster.

## 5.2. Single machine characteristics

In this section we will be showing the different physical capabilities of the different machines composing the cluster, for it we will use tables with the following schema.

| | |
|---|---|
| **Architecture** | |
| **CPU** | |
| **RAM Memory** | |
| **Storage Memory** | |

Table 31 Machine characteristics schema.

In order not to have tables that have the same characteristics we will aggregate machines having the same characteristics (* means all machines with that prefix).

### 5.2.1. Compute 1-2, 1-3 and 1-7 machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5405 @ 2.00GHz- 8 cores |
| **RAM Memory** | 8GiB |
| **Storage Memory** | 1000.0 GB (1 disk) |

Table 32 Compute 1-2, 1-3 and 1-7 characteristics.

### 5.2.2. Rest of Compute-1-* machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5405 @ 2.00GHz- 8 cores |
| **RAM Memory** | 8GiB |
| **Storage Memory** | 2000.0 GB (2 disks) |

Table 33 Rest of Compute 1-* characteristics

### 5.2.3. Compute-3-* machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz – 255 cores |
| **RAM Memory** | 143GiB |
| **Storage Memory** | 240.0GB (1 disk) |

Table 34 Compute 3-* characteristics.

### 5.2.4. Compute-5-* machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5640 @ 2.67GHz – 16 cores |
| **RAM Memory** | 64GiB |
| **Storage Memory** | 1000.0GB (1 disk) |

Table 35 Compute 5-* characteristics.

### 5.2.5. Compute-6-* machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5645 @ 2.40GHz – 24 cores |
| **RAM Memory** | 24GiB |
| **Storage Memory** | 1000.0GB (1 disk) |

Table 36 Compute 6-* characteristics.

### 5.2.6. Compute-7-* machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E7- 4807 @ 1.87GHz – 48 cores |
| **RAM Memory** | 128GiB |
| **Storage Memory** | 1000.0GB (1 disk) |

Table 37 Compute 7-* characteristics.

### 5.2.7. Compute-8-* machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz – 24 cores |
| **RAM Memory** | 64GiB |
| **Storage Memory** | 1000.0GB (1 disk) |

Table 38 Compute 8-* characteristics.

### 5.2.8. Compute-9-1 and 9-3 machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz – 32 cores |
| **RAM Memory** | 256GiB |
| **Storage Memory** | 2000.0GB (2 disks) |

Table 39 Compute 9-1 and 9-3 characteristics.

### 5.2.9. Compute-9-2 machine

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz – 32 cores |
| **RAM Memory** | 384GiB |
| **Storage Memory** | 2998.0GB (1 disk) |

Table 40 Compute 9-2 characteristics.

### 5.2.10. Compute-10-* machines

| | |
|---|---|
| **Architecture** | PowerPC Architecture ppc64el |
| **CPU** | POWER8E (raw), altivec supported – 160 cores |
| **RAM Memory** | 63GiB |
| **Storage Memory** | 283.0GB (1 disk) |

Table 41 Compute 10-* characteristics.

### 5.2.11. Compute-11-* machines

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | Intel(R) Xeon(R) CPU E5-2603 v4 @ 1.70GHz – 12 cores |
| **RAM Memory** | 127GiB |
| **Storage Memory** | 256.0GB (1 disk) |

Table 42 Compute 11-* characteristics.

### 5.2.11. Storage-1 machine

| | |
|---|---|
| **Architecture** | Amd64 |
| **CPU** | 8 cores |
| **RAM Memory** | 32GiB |
| **Storage Memory** | 10235.0GB (5 disks) |

Table 43 Compute 11-* characteristics.

## 5.3. Aggregated capabilities

In this section we will provide the aggregated physical limitations of the cluster (WITHOUT taking into account all the resources used for operating system, programs, etc).

| | |
|---|---|
| **CPU’s** | 1198 cores |
| **RAM Memory** | 3981 GiB |
| **Storage Memory** | 48,044 TB |

Table 44 Aggregated capabilities of the cluster.

# 6. Implementation of the application

In this section we will be discussing the different steps we followed in order to create the security application. We will describe the communication between our server, client and visualization entities.

We will divide the section in each of the previously mentioned entities in order to provide understanding of the application.

## 6.1. Server application

As stated on the introduction, the server application is the typical multithreaded C server created using *Unix Stream Sockets*. This server is intended to be gathering different information about the system in which it will be running, each thread reading and sending a different file to be analysed by the client side of the application.

### 6.1.1. Communication protocol

As stated on the introduction, the communication protocol will be contingent upon the election of the type of sockets we will be using to develop the server. In this section we will analyse why we chose the type of sockets we did and provide a brief explanation on how the communication process is carried out.

The sockets we will be working with are *Unix Stream Sockets*, these type of sockets are connection-oriented and provide in-order delivery of the flow of information, they work (usually) over TCP internet protocol. We chose these type of sockets as we are creating a Streaming application and thus the requirement for Stream sockets and connection-oriented communication.

The communication protocol is a simple 3 steps protocol in which:

1. Server creates a local socket descriptor that will be permanently listening for incoming connections from clients.
2. Client issues a connection request and the server accepts it creating a new socket that will be in charge of the communication between the server and a particular client.
3. The server is continuously (as it is a streaming application) sending information to the client for it to process the gathered data.

We can observe the different steps on the diagram of Figure 39 below.

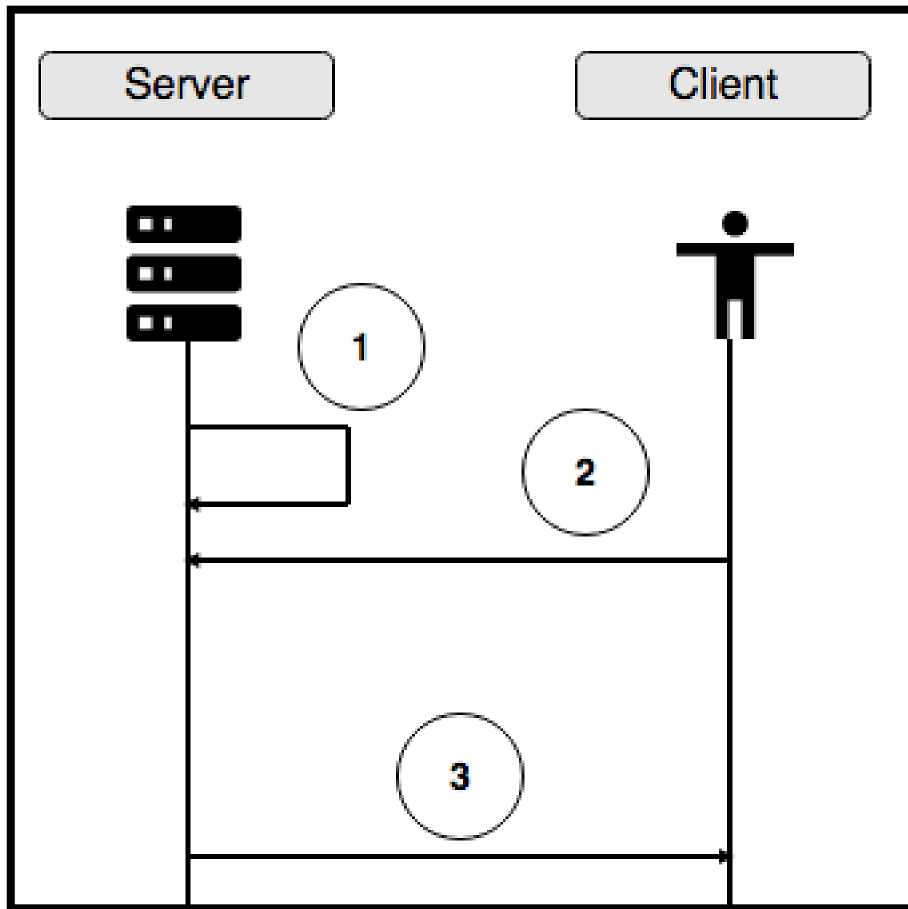


Figure 39 Communication protocol.

## 6.1.2. Implementation

In order to implement it, a series of steps where followed (to assure correctness) for the incremental development of the application. The first step was to have a simple C server, which was fully working (no multithreading was added in this first part). To do so we followed the main points to take into account when creating a C-based server.

1. Importation of the necessary libraries to have a functioning C server: the first thing we have to do is import the server-related libraries to our C application, these libraries include *<sys/socket.h>* (library used for the usage of computer sockets [29]) and *<arpa/inet.h>* (library used for the manipulation of internet operations [30]), apart from these libraries other C libraries where included for the application to correctly function *(<stdio.h>, <string.h>*, *<stdlib.h>…*). All these libraries add functionality to our C server for easiness.

2. Definition of the main variables to be used: the next step was to define the different variables that are required for a server to function (socket addresses for both the client and the server, socket descriptors, etc).

*a.* Addresses: we require 2 different addresses, which can be seen on Figure 40 the suffix _in is related to the type of socket indicating that it's a *AF_INET* socket working over TCP.

```
struct sockaddr_in server_addr;
struct sockaddr_in client_addr;
```

**Figure 40 Socket addresses.**

These addresses are unique and identify the connection points between the client and the server application; they are composed of three different parts (32 bits to identify the IP address, 16 bits to identify the port associated to that socket).

3. Socket creation: the next step is to create and set the different information regarding the server socket that will be accepting connections from the different clients. This process can be seen on Figure 41.

```
sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

val = 1;
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, (char *) &val, sizeof(int));

bzero((char *) &server_addr, sizeof(server_addr));
server_addr.sin_family                    = AF_INET;
server_addr.sin_addr.s_addr       = INADDR_ANY;
server_addr.sin_port                      = htons(9999);
```

**Figure 41 Socket creation + tweaking.**

In here we see the type of socket being created (*SOCK_STREAM*) and the protocol being used (*IPPROTO_TCP*) as well as the setting of the IP address and port (9999).

4. Binding of the socket: the process of binding refers to the assignment of the different arguments of a socket, this action assigns the address and port to the created socket.

```
bind(sd, &server_addr, sizeof(server_addr));
```

**Figure 42 Binding of the socket.**

5. Listen + Accept: the next step would be for the server to listen for incoming connections and for it to be accepting connections in an infinite loop (*while(1)*). This is achieved by using the listen call (it prepares the socket to accept connections) and, inside the infinite loop, accept them (while there is connections to accept).Once a connection has been accepted the server obtains both the client's socket address and a new socket. These 2 steps can be seen on Figure 43.

```
listen(sd,5);

size = sizeof(client_addr);

while(1){
        //Infinite loop recieving requests.
        printf("Waiting for connection\n");
        printf("LLEGO\n");
        //sc = accept(sd, (struct sockaddr *) &client_addr,(socklen_t *) &size);


        while(sc = accept(sd, (struct sockaddr *) &client_addr,(socklen_t *) &size)){

                printf("Got a connection from (%s , %d)\n",inet_ntoa(client_addr.sin_addr),ntohs(client_addr.sin_port));
```

**Figure 43 Listen + Accept functions.**

6. Function to be carried out by the threads: once the C server was accepting connections and functioning in the desired way, the next step was to create the function to be carried out by the different threads. This function is composed of 3 different parts:
    a. Arguments to be passed to the thread: each thread requires 3 different arguments represented on Figure 44.

```
struct readProcParams{
        /* data */
        char *path;
        char *s;
        int socket;
};
```

**Figure 44 Arguments to the threads function.**

These arguments are the path to the file to be read (represented by a constant defined on the program), a pointer to a string that will later be dynamically allocated in which the contents of the file will be copied (with some information added to it) and the socket descriptor assigned to each thread.

    b. Adding to the beginning of the file the last part of the path to identify each file on the client side: The next step was to add identifiers to each file to make easier the job to be carried out on the client-side of the application.

```
//Used to insert identifier.
/*******************************************************************************************************/
/**/    char aux[25];
/**/    int j=0;
/**/    int count = 0;
/**/    int size_aux;
/**/    for(j = strlen(parameters->path), j<=0; j--;){
/**/
/**/            if(parameters->path[j] == '/' && count == 0){
/**/                    count++;
/**/                    int t = 0;
/**/                    int k = j;
/**/                    k++;
/**/
/**/                    printf("Size to be printed: %ld\n", (strlen(parameters->path) - k));
/**/                    size_aux = (strlen(parameters->path) - k);
/**/                    for(t=0; t<=(strlen(parameters->path) - (j+1)); t++){
/**/                            aux[t] = parameters->path[k];
/**/                            k++;
/**/                    }
/**/            }
/**/    }
/**/    aux[size_aux] = ':';
/**/    aux[size_aux+1] = ' ';
/**/    printf("Aux es: %s\n", aux);
/*******************************************************************************************************/
```

Figure 45: Identifier addition.

c. Dynamic allocation of the file to be copied and sent: the final ster is to dynamically allocate the memory required to copy the file to the argument referred to in the previous section and copying the file character by character (this was done as in the initial files to be treated in order to learn Scala, identifiers were required along the file). The implementation can be observed on Figure 46.

```
//Obtaining file from /proc.
FILE *fp;

fp = fopen(parameters->path, "r");

fseek(fp,0L,SEEK_END);

size_t size = ftell(fp);

parameters->s = (char *) malloc(size * sizeof(char));

if(fp == NULL){
        perror("Error opening the file: ");
}

//Seek begining of file.
fseek(fp, SEEK_SET, 0);

int i = 0;
while( ( ch = fgetc(fp) ) != EOF ){

        parameters->s[i] = ch;
        i++;
}
printf("%s",parameters->s);

//printf("asdf\n");
//strcat(aux, parameters->s);
//printf("FALLO1\n");

write(parameters->socket, parameters->s, size * sizeof(char));
free(parameters->s);
fclose(fp);
sleep(3);
```

Figure 46 Dynamic allocation, copying and sending of the desired file.

d. Files to be sent by the server application: the final server application once it obtains a connection from a client creates 2 threads each of them sending a different file to the client application. These files are the following:

   i. auth.log: this is the first file we were working with, it contains all logging information of the system. The use of this log-file is to get information regarding both IP's trying to use invalid users as well as those trying to connect to existing users with a wrong password.
   ii. fail2ban.log: *Fail2ban* [31] is an application that scans system log files and bans IP's that show signs of malicious activity towards the system. The use of this type of log is very powerful as we can obtain IP's trying to do DNS Lookups and all the banned IP's for brute force attacks.
   iii. diskstats: this file shows the I/O statistics for block devices, specific information to what each of the fields on this file represent can be found in [32].
   iv. meminfo: this file represents information regarding RAM memory of the system. It will be used to obtain information regarding: total memory of the system, available memory of the system, free memory of the system, buffered memory of the system and cached memory of the system. All values on this file are given in kB.

## 6.2. Client application

The client application is entirely based on an *Apache Spark Streaming* self-contained application, for it to work properly we need the use of *sbt* (which was explained on Section 4.1. Apache Spark Installation) to build the applications.

As with the server application, the client application was developed on an incremental way, first we would be working on the spark-shell (can be observed on Figure 21 Spark shell.) with simple files to get used to Scala code and Spark's requirements, and once that was successfully working we created the self-contained application that will be run using *Apache-Yarn* as resource manager of the cluster.

The main purpose of the client application is to develop a streaming process that cleans the log files that are being sent by the server so that only the desired information is stored; this information will then be mapped to *Scala's case classes* and stored into *HDFS*.

### 6.2.1. Keyconcepts

In order to fully comprehend the different aspects of the implementation of the client process, a few concepts must be introduced (apart from those on Section 2.3. Apache Spark).

- Spark Configuration: the Spark Configuration (*SparkConf()*) sets up different properties of the application (master of the application, name of the application, allowing multiple contexts…). Once the configuration of the application has been created matching the requirements needed, this configuration will be used to create the Spark Context.

- Spark Context: the Spark Context (*SparkContext()*) is essentially the execution environment of a spark application; it enables us to use all the spark functionality (working with RDD, parallel operations…). In Figure 47 we can observe the different components of the Spark context.

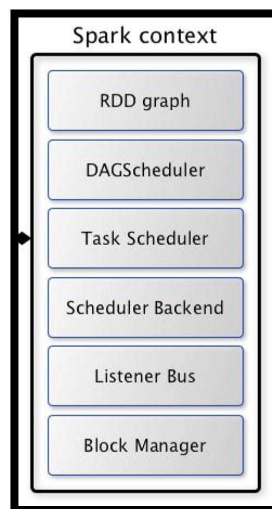


**Figure 47 Spark Context [33].**

Until the spark context is created we cannot start to implement the spark application, as it is the element that initializes the whole Spark environment.

- Spark Streaming Context: the Spark Streaming Context (*StreamingContext()*) is similar to the Spark Context but initializes streaming functionality (data captured with streaming context will be composed of DStreams instead of RDDs).

- Spark SQL Context: the Spark SQL Context (*org.apache.spark.sql.SQLContext*) is the entry point to be able to use SQL functionality in the Spark application.

### 6.2.2. Implementation of the Spark Streaming application

In this section we will be explaining how the Spark Streaming application was made. In every spark application we have two main parts in the code.

#### 6.2.2.1. Case classes

Outside our main function we have to define our *Case classes*, *Case classes* are regular classes that are extended by the compiler to automatically support the following features:

1. Public getters for constructor parameters.
2. Pattern matching on constructor parameters.
3. Copy constructor.
4. Automatic toString/hash/equals implementation.

Case classes (as we are working with Spark RDDs) should not be mutable.

In our code we use case classes to match the information we want to extract from the files sent by the server application to later be stored in HDFS from which the visualization application will read the information.

We created the following *Case classes*:

- Connection: this *Case class* matches information from the auth.log file, it stores username, day, hour, minute, second, IP and port, of every connection that was not successful (both with existing users on the system as those who do not).

- Ban: this *Case class* matches information from the fail2ban.log file; it is storing IP's that were banned. Its parameters are the timestamp the IP was banned (month, day, hour, minute and second) its IP.

- DNS: this *Case class* also matches information from the fail2ban.log file, in this case it's the IP's that were detected performing DNS Lookup's. It the date of the infringement (year, month and day), its time (hour, minute and second) and the IP that did it.

- DiskSts: this *Case class* matches information from the diskstats file; we will be storing the device name, number of reads completed, time spent reading, number of writes completed, time writing and overall I/O time.

- MemoryInfor: this *Case class* matches information from the meminfo file, we will be storing the ID of the information being stored (total, free available, cached or buffered memory) and the value in kB's.

In Figure 43 we can observe the creation of the case classes outside the main function.

```
object TFG{

    case class MemoryInfo(ID: String, Value: Long)
    case class DiskSts(Name: String, Reads: Long, ReadT: Long, Writes: Long, WriteT: Long, IOT: Long)
    case class Conection(User: String, Day: Int, Hour: Int, Minute: Int, Second: Int, Ip: String, Port: String)
    case class Ban(Month: Int, Day: Int, Hour: Int, Minute: Int, Second: Int, Ip:String)
    case class DNS(Date: String, Hour: Int, Minute: Int, Second: Int, Ip:String)

    def main(args: Array[String]) : Unit = {
```

**Figure 48 Case class creation.**

#### 6.2.2.2. Main function

In this section we will be explaining the actual implementation of the Spark application, we will divide the explanation into three main parts.

1. Initialization of the application: the first step is to prepare the application to be able to run as expected, this process includes creating the configuration of the application, its necessary contexts and other minor details (setting Spark so that it does not show all the logging information of the application). We can observe this process on Figure 49.

```
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)

// Define the Spark configuration. In this case we are using the local mode
val sparkConf : SparkConf = new SparkConf().setAppName("TFGfinha")
                            .set("spark.driver.allowMultipleContexts", "true");

val sc : SparkContext = new SparkContext(sparkConf)

val sqlContext= new org.apache.spark.sql.SQLContext(sc)
import sqlContext.implicits._


val ssc = new StreamingContext(sparkConf, Seconds(10))
```

**Figure 49 Initialization of the application.**

The first 2 lines of code are the ones suppressing part of the Spark output, afterwards we create the spark configuration, giving the application a name and specifying that more than one context can coexist. Then we create the Spark Context, which we will be used to create the sqlContext (necessary for later on transforming *RDD's* to *Dataframes*). Finally we create the Streaming Context indicating the duration of every batch (as we said previously Spark Streaming is not real streaming but rather micro-batching) to 10 seconds, this will make that every *DStream* will be a collection of *RDD's* that were sent during that batch of time.

2. Filtering of the *DStreams* and mapping to our *Case classes*: the next step was to, first, capture the input data. The line of code shown in Figure 50 achieves this.

```
//Obtain Input Data
val lines = ssc.socketTextStream("10.0.4.2", 9999)
```

Figure 50 Server connection call.

Then we filter the incoming data on our *DStream* (which contains all the information from all the files sent by the server), so that we can map the necessary information to our *Case classes*. We first started with the *fail2ban.log* related data, to do so we look for different keywords that are only available (and our source of information) on that file. On Figure 51 we can observe how the filtering and mapping to the desired *Case classes* was made.

```
//Procesamos el archivo de logs FAIL2BAN
val fail2ban_ban = lines.filter(_.contains("Ban"))
val fail2ban_dns = lines.filter(_.contains("WARNING"))
val fail2ban_ban_splitted = fail2ban_ban.map(x => x.split("[- :,]"))
val fail2ban_dns_splitted = fail2ban_dns.map(x => x.split("[ :,]"))

//Mapeamos a case class
val my_BAN = fail2ban_ban_splitted.map(x => Ban( Month = x(1).toInt, Day = x(2).toInt, Hour =  x(3).toInt,
val my_DNS = fail2ban_dns_splitted.map(x => DNS( Date = x(0).toString, Hour =  x(1).toInt, Minute = x(2).to
```

Figure 51 Filtering and mapping of fail2ban.log data.

Each of the *Case classes* matches a certain keyword, lines containing the keyword “Ban” are IP’s which have been banned due to miss authentication when trying to Access the system. Lines containing the keyword “WARNING” are IP’s, which have tried to do a DNS Lookup. Then we apply to each element of the *DStream* a map function to split the according to a certain regular expression, which provides a map in which we can access its elements to be then mapped to our *Case classes*.

The next step was to filter the data related to the *auth.log* file, the process is very similar to the one followed when dealing with the *fail2ban.log* file, we encountered a problem when dealing with this file because when the file contained days of the month that had only 1 digit (1 – 9 of every month) instead of inserting it as 01 – 09 it will insert a space more causing the program to crush. To fix this we split the filtering between the cases matching this problem and the rest of them. The implementation of this section can be observed on Figures 52 and 53.

```
//Procesamos el archivo de logs AUTH
val auth = lines.filter(_.contains("Failed password"))
val auth_filtered = auth.filter(!_.contains("repeated"))

val auth_invalid_1 = auth_filtered.filter(_.contains("invalid"))
val auth_invalid = auth_invalid_1.filter(!_.contains("  "))
val auth_invalid_mm1 = auth_invalid_1.filter(_.contains("  "))

val auth_rest_1 = auth_filtered.filter(!_.contains("invalid"))
val auth_rest = auth_rest_1.filter(!_.contains("  "))
val auth_rest_mm1 = auth_rest_1.filter(_.contains("  "))

val auth_invalid_splitted = auth_invalid.map(x => x.split("[: ]"))
val auth_rest_splitted = auth_rest.map(x => x.split("[: ]"))
val auth_invalid_mm1_splitted = auth_invalid_mm1.map(x => x.split("[: ]"))
val auth_rest_mm1_splitted = auth_rest_mm1.map(x => x.split("[: ]"))
```

Figure 52 Filtering of auth.log data.

```
//Mapeamos a case class
val my_CONN_INV = auth_invalid_splitted.map(x => Conection( User = x(13).toString, Day = x(

val my_CONN_INV_mm1 = auth_invalid_mm1_splitted.map(x => Conection( User = x(14).toString,


val my_CONN_REST = auth_rest_splitted.map(x => Conection( User = x(11).toString, Day = x(1]

val my_CONN_REST_mm1 = auth_rest_mm1_splitted.map(x => Conection( User = x(12).toString, Da

val my_CONN_REST_FINAL = my_CONN_REST.union(my_CONN_REST_mm1)
val my_CONN_INV_FINAL = my_CONN_INV.union(my_CONN_INV_mm1)
```

Figure 53 Mapping of the data to Case classes.

The final step was processing the files corresponding to *diskstats* and *meminfo* and mapping them to our created case classes, this process can be seen on Figures 54 and 55.

```
//Procesamos el diskstats
val disk = lines.filter(_.contains("sd"))
val disk_filtered = disk.filter(!_.contains("urraca")
val disk_splitted = disk_filtered.map(x => x.split("\\s+"))

//Mapeado a Case class
val my_DISK = disk_splitted.map(x => DiskSts(Name = x(12).toString, Reads = x(13).toLong,
```

Figure 54 Filtering and Mapping of diskstats information.

```
//Procesamos meminfo
val mem_Total = lines.filter(_.startsWith("MemTotal"))
val mem_Available = lines.filter(_.startsWith("MemAvailable"))
val mem_Free = lines.filter(_.startsWith("MemFree"))
val mem_Buffered = lines.filter(_.startsWith("Buffers"))
val mem_Cached = lines.filter(_.startsWith("Cached"))


val mem_Total_splitted = mem_Total.map(x => x.split("\\s+"))
val mem_Available_splitted = mem_Available.map(x => x.split("\\s+"))
val mem_Free_splitted = mem_Free.map(x => x.split("\\s+"))
val mem_Buffered_splitted = mem_Buffered.map(x => x.split("\\s+"))
val mem_Cached_splitted = mem_Cached.map(x => x.split("\\s+"))

//Mapeado a case class
val my_TOTAL = mem_Total_splitted.map(x => MemoryInfo(ID = x(0).toString), V
val my_AVAI = mem_Available_splitted.map(x => MemoryInfo(ID = x(0).toString)
val my_FREE = mem_Free_splitted.map(x => MemoryInfo(ID = x(0).toString), Val
val my_BUFF = mem_BUFF_splitted.map(x => MemoryInfo(ID = x(0).toString), Val
val my_CACH = mem_CACH_splitted.map(x => MemoryInfo(ID = x(0).toString), Val
```

Figure 55 Filtering and Mapping of meminfo data.

3. Writing into HDFS and starting the application: The final step is to take our mapped *DStreams* into *Case classes* and write them to our HDFS directory and starting the application. To do so we have to apply the *foreachRDD()* method which applies a function to each *RDD* contained on the *DStream*, we will be transforming *RDD's* to *Dataframes* and latter using a Databricks library (*com.databricks.spark.csv*) [34] to write them in CSV format. This process is shown on Figure 56.

```
//Escribimos en Hdfs
my_DNS.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia/T

my_BAN.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia/T

my_CONN_INV_FINAL.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/use

my_CONN_REST_FINAL.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/us

my_DISK.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia/

my_TOTAL.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia

my_AVAI.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia/

my_FREE.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia/

my_BUFF.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia/

my_CACH.foreachRDD{ rdd => rdd.repartition(1).toDF().write.format("com.databricks.spark.csv").option("path", "/mnt/hdfs/user/rgarcia/
```

Figure 56 Writiing to HDFS + starting of the application.

Note that the application will not start running until we call the *ssc.start()* call, *ssc.awaitTermination()* waits for the termination signal from the server (meaning it wont stop until de server does).

## 6.3. Visualization application

The visualization section of the application was carried out using *Apache Zeppelin* (that was introduced on Section 2.8. Apache Zeppelin). Zeppelin works as a spark-shell based application; on Figure 57 we can observe Zeppelin being an application working on the Hadoop system by using Spark.

Show 20 entries Search:

| ID | User | Name | Application Type | Queue | StartTime | FinishTime | State | FinalStatus | Running Containers | Allocated CPU VCores | Allocated Memory MB | Progress | Tracking UI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| application_1497944767146_0037 | rgarcia | TFGfinha | SPARK | root.users.rgarcia | Tue Jun 20 13:57:58 +0200 2017 | Tue Jun 20 15:42:38 +0200 2017 | FINISHED | SUCCEEDED | N/A | N/A | N/A | | History |
| application_1497944767146_0036 | rgarcia | TFGfinha | SPARK | root.users.rgarcia | Tue Jun 20 13:57:48 +0200 2017 | Tue Jun 20 15:42:38 +0200 2017 | FINISHED | SUCCEEDED | N/A | N/A | N/A | | History |
| application_1497944767146_0035 | root | Zeppelin | SPARK | root.users.root | Tue Jun 20 13:06:09 +0200 2017 | N/A | RUNNING | UNDEFINED | 21 | 81 | 31744 | | ApplicationMaster |

Figure 57 Applications on the cluster.

Zeppelin is a web application based on *Notebooks* and it gives us the option to run the Notebook according to a timer so that the application is more real-time like. We will use 2 different interpreters in order to perform the visualization of the application.

- Scala interpreter: we will be using Scala-based code in order to load the data stored by the client application into *Dataframes* so that we can later represent this data using different types of graphs.

- SQL: in order to perform the different graphs we will be using SQL-based code to format the loaded data. Zeppelin provides interactive graphs on table-like *Dataframes*.

In the following sections we will provide an explanation on the different graphs created for the visualization application (code-wise, what they represent and the files which they are related to).

### 6.3.1. Loading of data

This is the first section of code of the Zeppelin application, we load the data stored by the client application into *Dataframes* and register them as tables. This process can be seen on Figure 58 below.

**Loading of data.**

```
val INV = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("hdfs:///user/rgarcia/INV.csv")
INV.registerTempTable("Invalid")

val error = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("hdfs:///user/rgarcia/REST.csv")
error.registerTempTable("Error")

val bans = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("hdfs:///user/rgarcia/BAN.csv")
bans.registerTempTable("Bans")

val dns = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").option("inferSchema", "true").load("hdfs:///user/rgarcia/DNS.csv")
dns.registerTempTable("DNS")

INV: org.apache.spark.sql.DataFrame = [User: string, Day: string, Hour: string, Minute: string, Second: string, Ip: string, Port: string]
error: org.apache.spark.sql.DataFrame = [User: string, Day: string, Hour: string, Minute: string, Second: string, Ip: string, Port: string]
bans: org.apache.spark.sql.DataFrame = [Month: string, Day: string, Hour: string, Minute: string, Second: string, Ip: string]
dns: org.apache.spark.sql.DataFrame = [Date: string, Hour: string, Minute: string, Second: string, Ip: string]
```

Took 5 sec. Last updated by anonymous at June 20 2017, 6:13:52 PM.

Figure 58 Data loading.

### 6.3.2. Invalid connections per day

In the following graph we show the number of connections that were made using invalid users on the system. This graph is related to the auth.log data source, we take the IP's and group them by day.

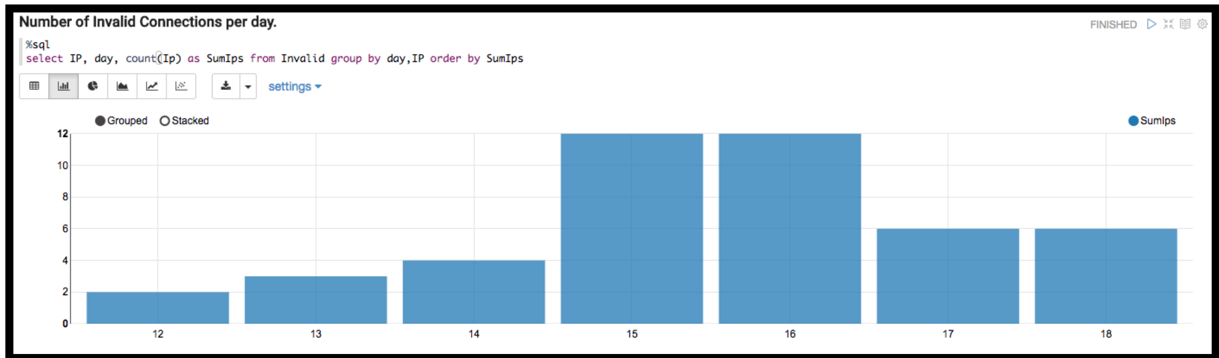


Figure 59 Invalid connections per day.

### 6.3.3. Invalid connections per IP

This graph represents IP's that are trying to access the system using invalid users in order to authenticate, it is also related to the auth.log file. We group data according to the IP that has been performing the malicious activity and order them in ascending order.

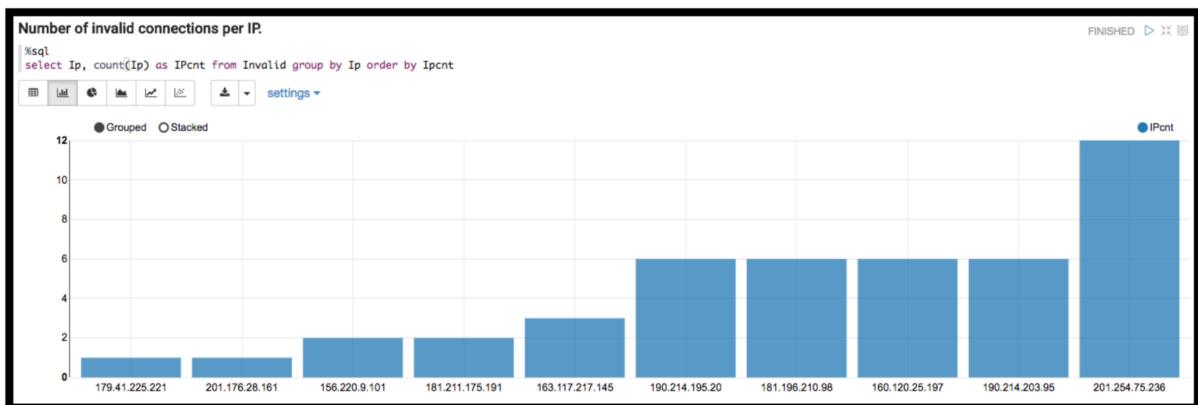


Figure 60 Invalid connections per IP.

### 6.3.4. Invalid users used to authenticate

The following graph represents the users IP's tried to authenticate as in order to Access the system, this information was also obtained from the auth.log file. As we can see on the graph the user that was used the most was "admin" (which is a pretty common user on every system).

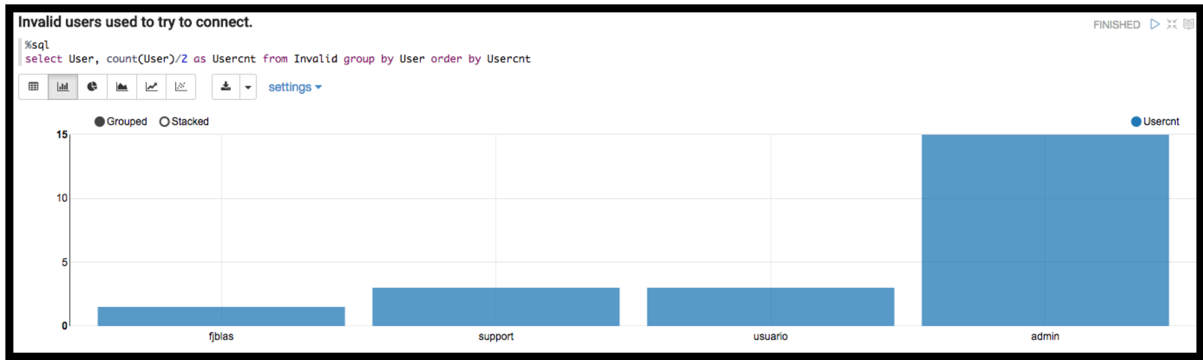


Figure 61 Invalid users.

## 6.3.5. Existing users failing to authenticate

This graph represents users that exist in our system that failed to authenticate, this information was taken from the auth.log file. As we can see on the graph the user that failed authentication the most was "root" (this is because is the most common user to attack as it is always present on every system).

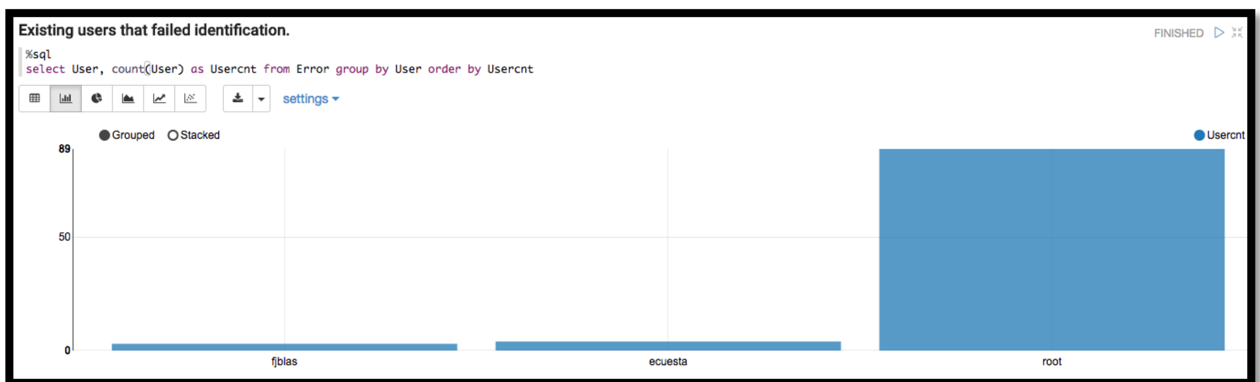


Figure 62 Existing users that failed authentication.

## 6.3.6. Authentication failures per day

This graph is similar to the one on Section 6.3.2. Invalid connections per day, it represents authentication failures per day by existing users.

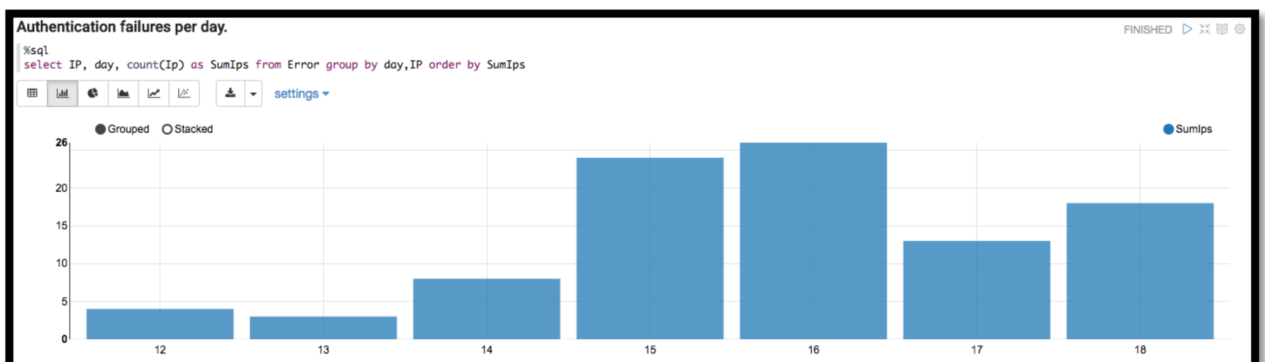


Figure 63 Authentication failures per day.

### 6.3.7. Top 10 IP's that tried to authenticate as root

In this graph we will be showing the 10 IP's that tried to authenticate as root the most. These are IP's trying to gain root Access to our system by brute force attacks. This information is taken from the auth.log file. We select those IP's whose username was "root" and order them in descending order of the number of connections tried and limit the output to 10.

Figure 64 Top 10 IP's attacking root user.

### 6.3.8. Number of IP's banned by the fail2ban application

This graph represents how many IP's were banned each day by the fail2ban service (which bans IP's according to the number of times it failed to authenticate). This graph is related to the fail2ban.log file.

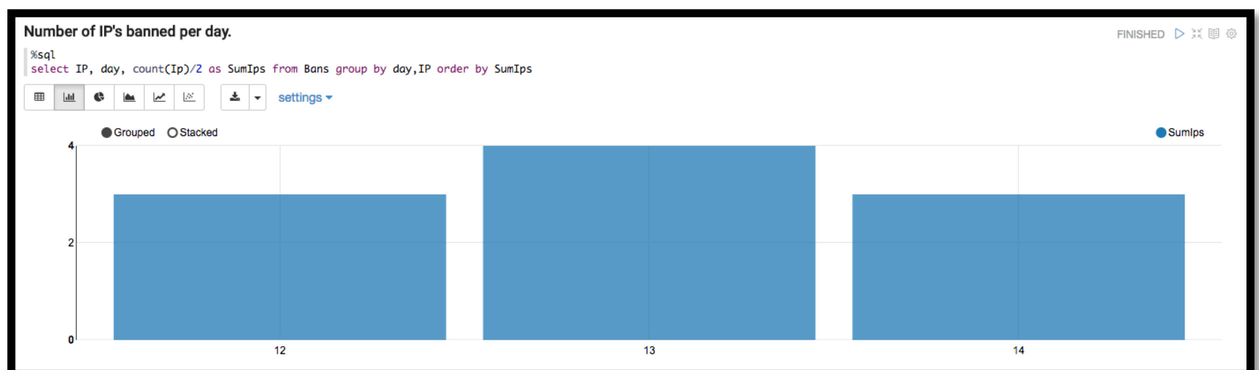


Figure 65 Banned IP's per day.

### 6.3.9. Actual IP's that were banned

This graph represents the actual IP's that were banned by the fail2ban service.

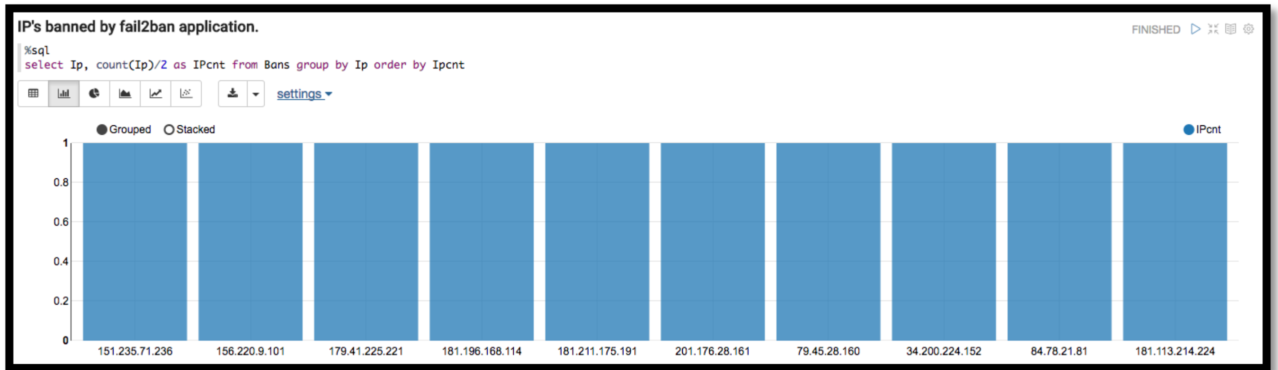


Figure 66 Banned IP's.

### 6.3.10. DNS Lookup's per date

This graph represents the number of DNS Lookup's that were tried to be performed on the system grouped by date. This information is also obtained from the fail2ban.log file. In this case the graph is not very representative as there was only 1 day in which there was this type of threat, but if there were more entries on the log file it would be much more helpful.

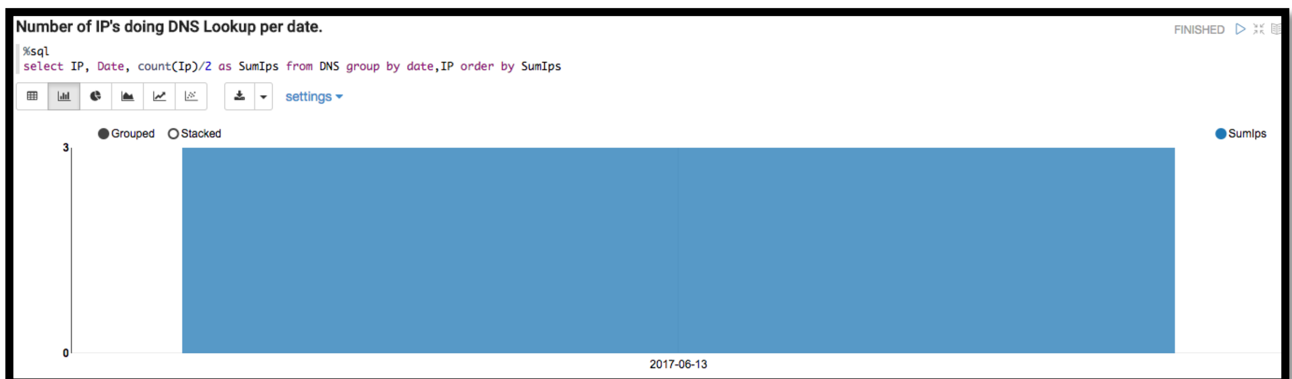


Figure 67 DNS Lookup per day.

### 6.3.11. IP's performing DNS Lookup

This graph shows the IP's that tried to perform a DNS Lookup on the system. The information shown on the graph is also obtained from the fail2ban.log file. We can also observe the relation to the previous graph on Figure 64 that was showing 3 IP's performing this type of attack (1 IP performs it once and the other one 2 times).

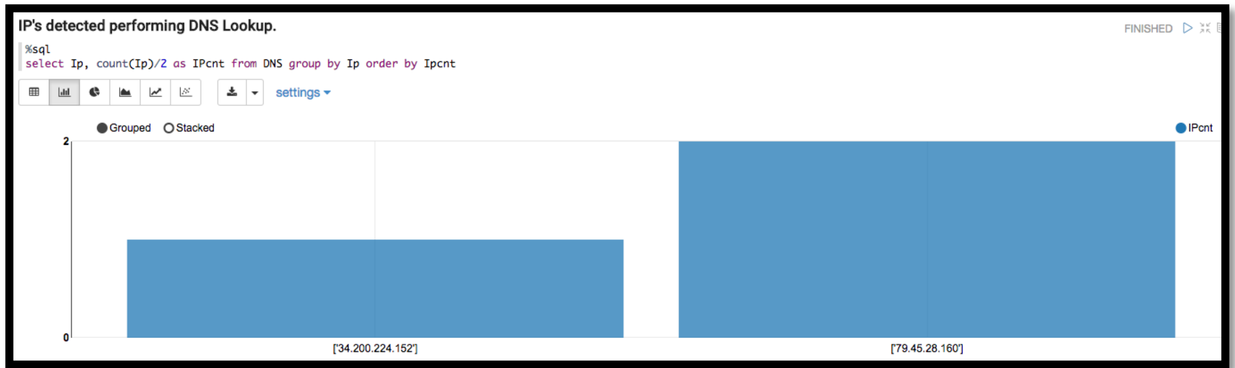


Figure 68 IP's performing DNS Lookup.

## 6.3.12. Disk writes per device and time

The following graph aims to illustrate the devices most used for writing so that afterwards we can perform time-series analysis over them. This information is obtained from the diskstats file. We can observe that the devices performing most of the writes are sdf and sdf1 and those will be the ones we will be performing time series over. We also provided 2 more graphs that are very similar to Figure 68 (analysing Reads and overall I/O time).

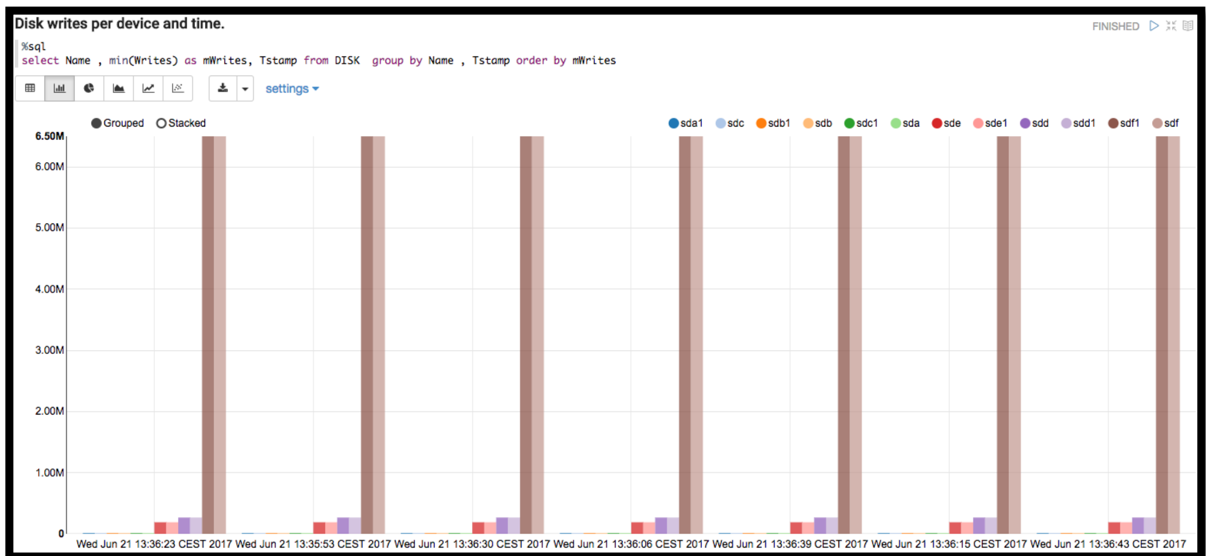


Figure 69 Disk writes per device and time.

### 6.3.13. Time-series graph for write operations on devices sdf and sdf1

This graph represents how the amount of writes completed by the stated devices behaves. As we can observe these graphs devices must be related as they behave in the same way.

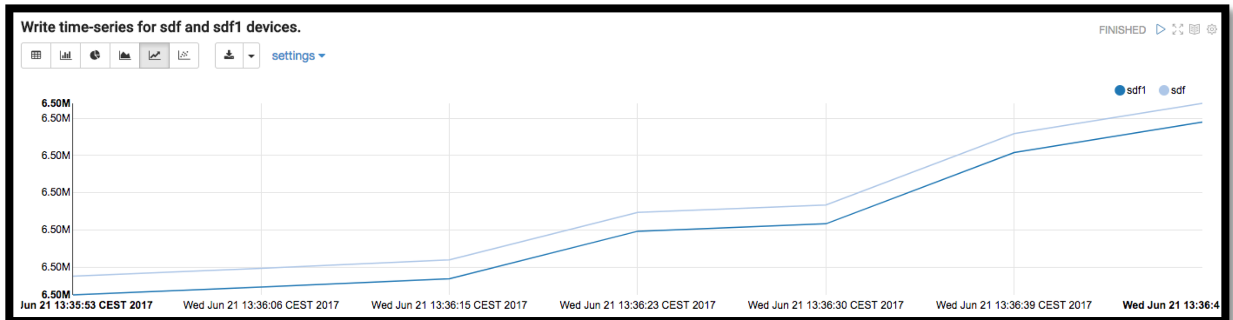


Figure 70 Write time-series graph.

### 6.3.13. Time-series graph for read operations on devices sdf and sdf1

This graph represents how the amount of reads completed by the stated devices behaves. As we can observe on Figure 70 during the execution of the application there were no reads on these devices.

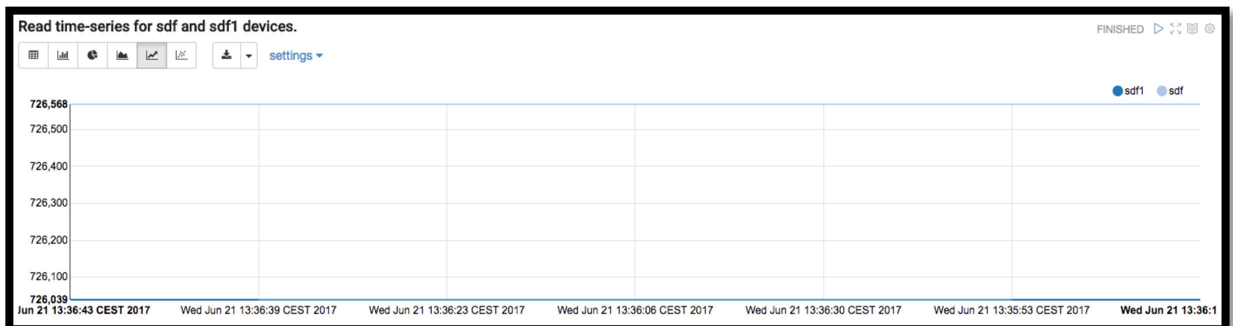


Figure 71 Read time-series graph.

### 6.3.14. Time-series graph for overall I/O time on devices sdf and sdf1

This graph represents the overall I/O time spent by the devices and how it behaves along time. As we can observe ist behaviour is very similar to the one on Figure 69 (as only write operations were performed on them).

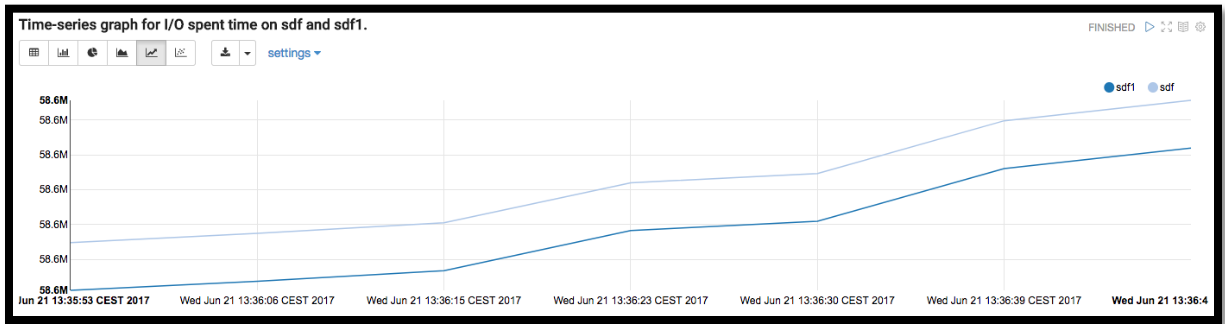


Figure 72 I/O time time-series graph.

## 6.3.14. Time-series graph for available RAM memory

This graph represents the available RAM memory (in GB) over time of the system and how it behaves. This information is obtained from the meminfo file.

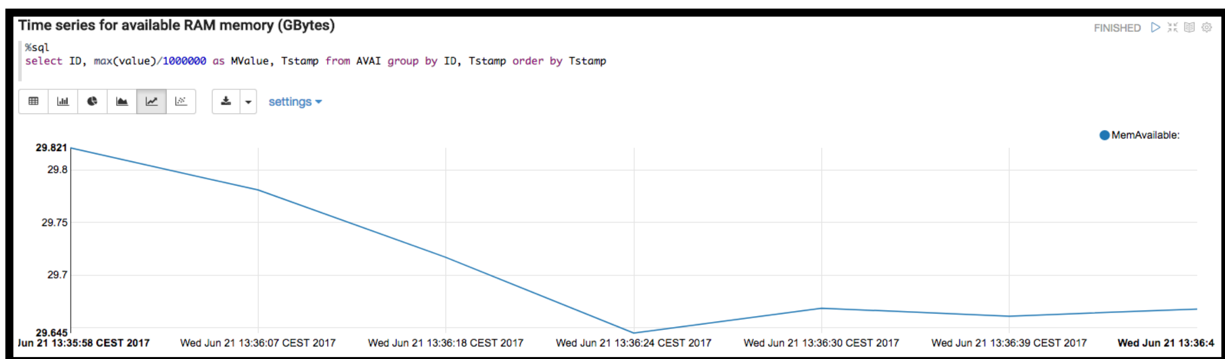


Figure 73 Available RAM time-series graph.

## 6.3.14. Time-series graph for free RAM memory

This graph represents the free RAM memory (in GB) over time of the system and how it behaves. This information is obtained from the meminfo file.

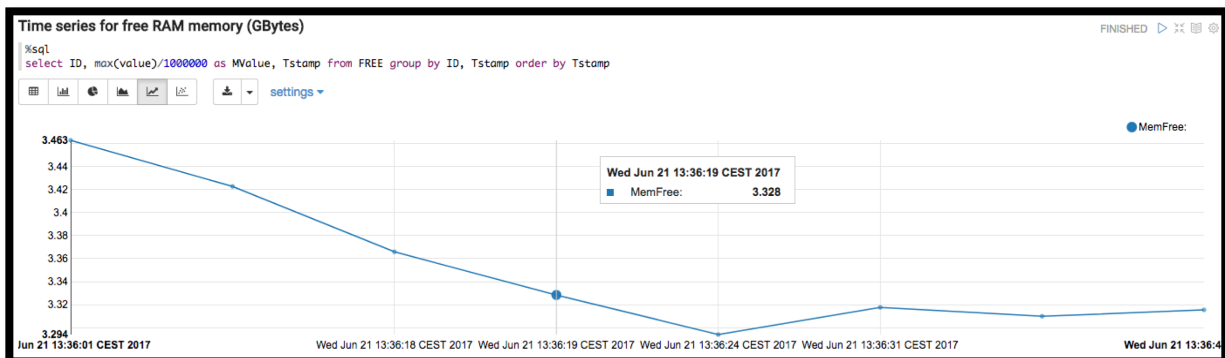


Figure 74 Free RAM time-series graph.

## 6.3.14. Time-series graph for buffered RAM memory

This graph represents the buffered RAM memory (in GB) over time of the system and how it behaves. This information is obtained from the meminfo file.

Figure 75 Buffered RAM time-series graph.

## 6.3.14. Time-series graph for cached RAM memory

This graph represents the cached RAM memory (in GB) over time of the system and how it behaves. This information is obtained from the meminfo file.

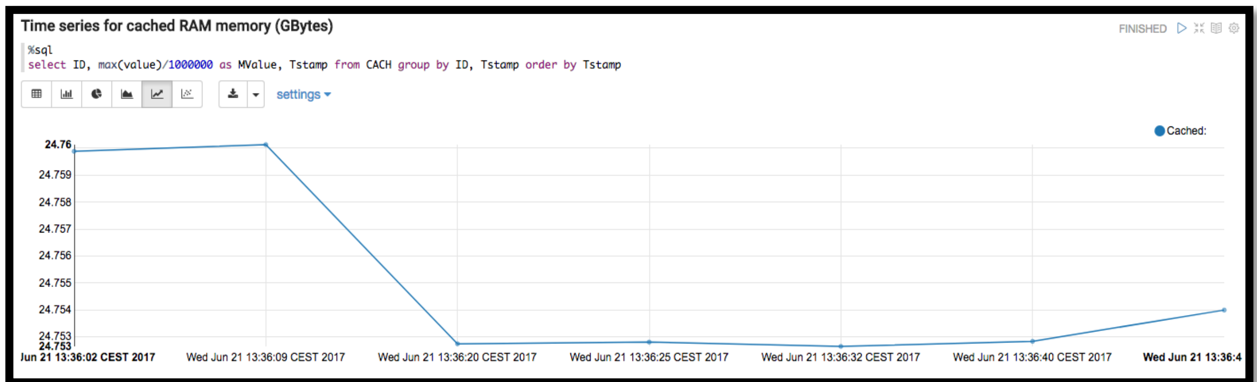


Figure 76 Cached RAM time-series graph.

# 7. Evaluation

This section aims to provide an evaluation plan for the application. As we are talking of a streaming application and it is not very processing-intensive, we cannot provide the usual benchmarking evaluation plans. We cannot provide an evaluation by covering the different use cases of the application because as we stated on Section 3.2. Use cases, there is only one use case that the application is fulfilling.

So in order to provide a evaluation method we will be evaluating depending on the window we set when we run the client application. In order to better understand this evaluation plan we will give a brief explaining on what that window is.

## 7.1. Window duration

When we create the Spark Streaming Application we bind the *streaming context* to a *Spark Configuration* and the window duration (see Figure 49 Initialization of the application.) this window marks the amount of data represented by each *DStream*, that is every *N* seconds a new DStream will be generated, and so the smaller the window the more real-time like the application will be. So in order to provide an evaluation plan we will be decreasing the window duration and observe the results.

In order to evaluate the application, the following parameters were taken into account:

- Application running for 1:30 minutes and then stopped.
- Application running in 1 node.
- Decreasing values of the window duration: 10s, 5s, 2s, 1s, 0.5s, 0.1s.
- Correctness: OK if the visualization works as expected FAIL if not.

| WINDOW DURATION (seconds) | CORRECTNESS |
|---|---|
| 10 | OK |
| 5 | OK |
| 2 | FAIL (auth.log related data fails, fail2ban.log data does not) |
| 1 | FAIL |
| 0.5 | FAIL |
| 0.1 | FAIL |

Figure 77 Evaluation of the application.

## 7.2. Conclusions of the evaluation

The application works as expected with window durations of 10 and 5 seconds, when we reduce the window duration to 2 seconds we start obtaining error son the graphs saying that data is not available, this is because the window is so small that we are overwriting the file with data at a speed that makes access to the data fail.

The section of graphs regarding time-line behaviour graphs never fails as we create new files each time a batch is processed in order to perform the time-analysis. The problem relies in the fact that if we reduce the window duration below 5 seconds most of the files generated are data-empty causing a waste of space on the storage environment.

As a conclusion we could say that the optimal time for the window duration of the application would be between 10 and 5 seconds so that it works as expected.

# 8. Project planning

In this section we will illustrate the planning done for the development of the project. We will show the initial planning and the real planning, finally a section with the budget for the development of the application will be shown.

## 8.1. Initial planning

This section contains the initial planning made for the development of the project, which was started on February 1st of 2016 and finished on June 14th. The planning can be seen on Table 41 below.

| Activity | Start date | Finish date | Days spent | Hours spent |
|---|---|---|---|---|
| Planning | 01/02/2017 | 03/02/2017 | 3 | 10 |
| Analysis | 04/02/2017 | 20/02/2017 | 16 | 40 |
| Design | 01/03/2017 | 09/03/2017 | 9 | 30 |
| Implementation | 10/03/2017 | 10/04/2017 | 31 | 65 |
| Testing | 12/04/2017 | 20/04/2017 | 8 | 15 |
| Evaluation | 21/04/2017 | 01/05/2017 | 11 | 20 |
| Documentation | 06/02/2017 | 14/06/2017 | 128 | 90 |

Table 45 Initial Planning.

To provide a more visual representation of the initial Project planning we offer a Gantt chart on Figure 78 below.

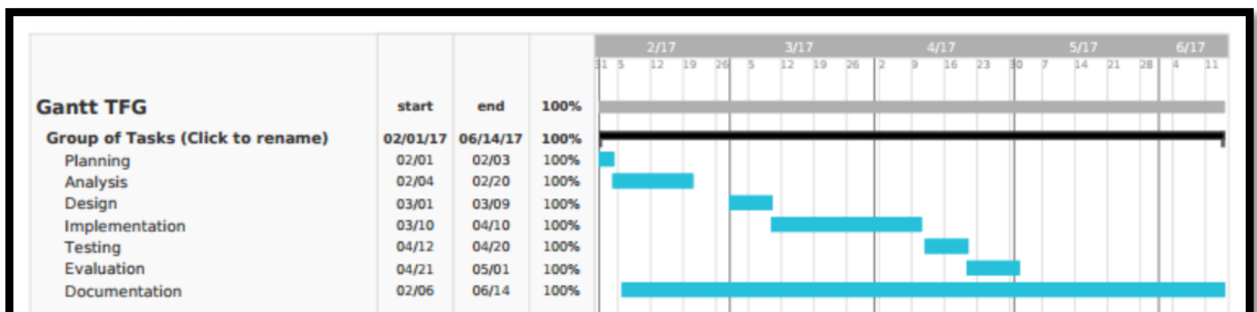


Figure 78 Gantt diagram for the initial planning.

## 8.2. Real planning

This section covers how the Project was really carried out, to show this information we will use the same elements as in the previous section. The main difference was in the time spent on the documentation of the Project having a difference of 7 days to the original planning.

| Activity | Start date | Finish date | Days spent | Hours spent |
|---|---|---|---|---|
| Planning | 01/02/2017 | 03/02/2017 | 3 | 10 |
| Analysis | 04/02/2017 | 20/02/2017 | 16 | 40 |
| Design | 01/03/2017 | 09/03/2017 | 9 | 30 |
| Implementation | 10/03/2017 | 10/04/2017 | 31 | 65 |
| Testing | 12/04/2017 | 20/04/2017 | 8 | 15 |
| Evaluation | 21/04/2017 | 01/05/2017 | 11 | 20 |
| Documentation | 06/02/2017 | 21/06/2017 | 125 | 120 |

Table 46 Real planning.

On Figure 79 below we can see the real Gantt chart of the development of the project.

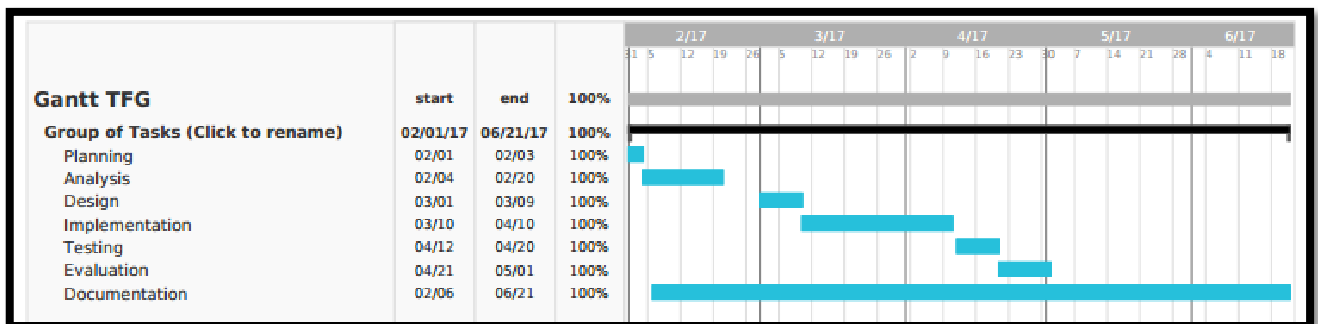


Figure 79 Gantt chart for the real planning of the project.

# 9. Socioeconomic Environment

This section covers the impact the application will have on both the social and economic environment as well as a section detailing the budget for the development of the application.

## 9.1. Socioeconomic impact

Our application will not have any impact on the economic or social fields, as there are a lot of similar technologies to the one being developed by us who offer the same type of functionality and provide free plans.

Also regarding the economic field, the project is intended to be open-source and will be uplodaded to a public repository on GitHub as a resource to the programming community, right now there are no intentions on commercialising the application.

## 9.2. Budget

In this section we will cover the budget to carry out the development of the project. We will divide it into 4 different sections: human resources, hardware, software and consumables.

### 9.2.1. Human resources budget

This section covers the money spent on people working to develop the project, for it we will base the data on Table 46 Real planning., adding the salary for a system administrator throughout the project For the development of the project we will need to hire a coder-analyst whose estimated salary will be of 40€/hour. We can observe the budget related to human resources on Table 44 below.

| Activity | Duration | Employee | €/hour | Total(€) |
|---|---|---|---|---|
| **Planning** | 10 | Coder-analyst | 40 | 400 |
| **Analysis** | 40 | Coder-analyst | 40 | 1.600 |
| **Design** | 30 | Coder-analyst | 40 | 1.200 |
| **Implementation** | 65 | Coder-analyst | 40 | 2.600 |
| **Testing** | 15 | Coder-analyst | 40 | 600 |
| **Evaluation** | 20 | Coder-analyst | 40 | 800 |
| **Documentation** | 120 | Coder-analyst | 40 | 4.800 |
| **Administration** | 300 | System administrator | 20 | 6.000 |
| **TOTAL** | | | | 16.000 |

Table 47 Human resources budget.

### 9.2.2. Hardware budget

This section details money spent on hardware resources; note that the application was carried out using the infrastructure of the universities cluster so all that hardware is not taken into account.

| Description | Value (€) |
| --- | --- |
| Mac Book Air 1.8 GHz (256GB) | 1.349,00 |

Table 48 Hardware budget.

### 9.2.3. Software budget

All the software used to carry out the application was open-source software so the total cost for software resources is of **0€** as we can observe on Table 46.

| Description | | Value (€) |
| --- | --- | --- |
| Cloudera distribution (version CDH-5.11.1-1.cdh5.11.1.p0.4) | | 0 |
| Includes: | Apache Spark | 0 |
| | Apache Hadoop | 0 |
| | Apache Yarn | 0 |
| | Apache Zookeeper | 0 |
| | HDFS | 0 |
| | Hive | 0 |
| Apache Zeppelin | | 0 |
| TOTAL | | 0 |

Table 49 Software budget.

### 9.2.4. Consumables budget

The approximated cost for all consumable items (printer toner, white pages, pens…) is of **200,00 €**.

### 9.2.5. Overall budget

This section shows a table gathering all the money spent during the development of the project, it can be seen on Table 47 below.

| Description | Ammount (€) |
| --- | --- |
| Human resources | 16.000,00 |
| Hardware | 1.349,00 |
| Software | 0,00 |
| Consumables | 200,00 |
| TOTAL | 17.549,00 |

**Table 50 Overall budget.**

# 10. Regulatory framework

In this section we will be analysing the different legal aspects of the application being developed, the different technical standards the application has to follow and issues regarding intellectual property.

## 10.1. Legal aspects

This section aims to prove that the application being developed is not incurring in violation of any existing law that may be applicable to de application being developed.

The main law that might be applicable to this application is the Spanish law concerning personal data treatment *"Ley Orgánica de Protección de Datos de Carácter Personal"* [35], and we will state the different points to observe that this law does not apply to our application.

1. According to article number 3 Section a), *"personal data is considered that of physical people that are identified or identifiable on our system"*. Data being gathered on our application is not identifying physical people but rather IP's, which are doing (or seem to) malicious activity on the system in which the application has been implanted.

2. According to article number 3 Section e) states that, *"the affected person is that how owns the data being used"*. All the data being used on our application is generated by our system and never obtaining it directly from physical people but rather auto-generated by analysis software.

3. Regarding article number 6 *"consent of the subject owning the data"*, we have demonstrated previously that there are no personal data in our system (as there is no data that can identify a physical person), and so this article is not applicable to our application.

4. Articles 7 and 8 (regarding *"specially protected data"* and *"health-related data"* respectively) are also not applicable as there are no data with that nature being captured.

5. Article 9 stating *"data security"* and that *"the person responsible for the file containing the data is also responsible for the security of the data"*, this article is also fulfilled, as only people that are able to identify themselves on the cluster system are able to access its resources.

So we have first of all demonstrated that data being captured on our application is not considered as personal data by the Spanish law of personal data protection, as they do not relate to a physical person at any time. Moreover being the application an intrusion analysis system, all the data being used is only used to analyse and defend the system from external attacks.

## 10.2. Technical standards

In the development of this application we have used three different programming languages, each of these has its own programming standards. The programming languages are the following (with references to its programming standards):

- C: the development of the server of the application has been programmed as a multithreaded C, its programming standards can be found at [36].

- Scala: the client application has been developed using Apache Spark engine with Scala language; its programming style guide can be found at [37].

- SQL: in the visualization platform (Zeppelin) we use SQL to obtain the final data to be visualized; the programming style guide can be found at [38].

## 10.3. Intellectual property

In this section we will discuss the matters regarding intellectual property of the application.

- Patentability: according to European law, software programs are not susceptible to be patented (Article 52 of the European Patent Convention Section c) *"schemes, rules and methods for performing mental acts, playing games or doing business, and programs for computers"* [39]

- Intellectual property: intellectual property is directly related to business and financial matters, all the work done during this project was carried out by the author of this document and is (currently) an investigation project to get familiar with a series of tools that will be beneficial for the future and maybe continue improving project.

# 11. Conclusions and further work

This section aims to give a glimpse what was learned throughout the development of the project and the future improvements that can be performed on the application.

## 11.1. Conclusions

Taken into account the objectives set for the application on Section 1.2. Objectives, we can say that these objectives have been correctly fulfilled. We also believe that the development of the project has helped to strengthen a lot of the content learned throughout the degree (distributed systems, visualization tools…).

Taking into account the knowledge acquired on all the Big Data technologies (which are very popular nowadays) used to carry out the application we believe that we have further added a lot of information on a filed that was not covered by the degrees syllabus that will help me differentiate myself from fellow colleagues

## 11.2. Further work

The application currently is at an early state of what it can achieve, this section will cover some of the future improvements that can be performed.

- The first improvement that can be made is aggregating more data to the application; we can process more files that contain useful data for analysing intrusions on the system. As it is a Big Data fashion application currently the amount of data being used is not that big so this would be one of the future lines of work to improve the application.

- Zeppelin provides an easy to use interface and is very useful for developing applications, but when it comes to representing the data it is very limited. One of the future lines of work will be to use other type of application (like Tableau) to provide better and more graphs to visualize the information.

# 11. References and bibliography

The table below gathers all the references throughout the document.


| REFERENCE NUMBER | LINK |
|---|---|
| [1] | https://vowi.fsinf.at/images/b/bc/TU_Wien-Verteilte_Systeme_VO_(Göschka)_-_Tannenbaum-distributed_systems_principles_and_paradigms_2nd_edition.pdf |
| [2] | https://en.wikipedia.org/wiki/Amdahl%27s_law |
| [3] | https://spark.apache.org/ |
| [4] | https://0x0fff.com/spark-architecture/ |
| [5] | http://www.pilar-tools.com/doc/v62/ISO27005.pdf |
| [6] | https://en.wikipedia.org/wiki/DREAD_(risk_assessment_model) |
| [7] | http://revieweasyhomemadecookies.com/what-is-hadoop-ecosystem/ |
| [8] | https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html |
| [9] | https://zookeeper.apache.org/doc/trunk/zookeeperOver.html |
| [10] | https://zeppelin.apache.org/ |
| [11] | https://www.loggly.com/ |
| [12] | https://www.loggly.com/blog/loggly-aws-cloudtrail-simple-way-operate-smarter/ |
| [13] | https://www.splunk.com/ |
| [14] | https://www.splunk.com/es_es |
| [15] | https://goaccess.io/ |
| [16] | https://github.com/ |
| [17] | https://goaccess.io/ |
| [18] | https://logz.io/ |
| [19] | https://www.elastic.co/ |
| [20] | https://github.com/logzio/logzio-es-health |
| [21] | https://en.wikipedia.org/wiki/Use_case_diagram |
| [22] | https://spark.apache.org/ |
| [23] | https://www.cloudera.com/ |
| [24] | https://www.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_intro.html |
| [25] | https://www.cloudera.com/documentation/enterprise/5-9-x/topics/installation.html |

| | |
|---|---|
| [26] | https://es.hortonworks.com/ |
| [27] | https://www.adictosaltrabajo.com/tutoriales/hdp-sandbox/ |
| [28] | https://en.wikipedia.org/wiki/19-inch_rack |
| [29] | http://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html |
| [30] | http://pubs.opengroup.org/onlinepubs/009696899/basedefs/arpa/inet.h.html |
| [31] | https://www.fail2ban.org/wiki/index.php/Main_Page |
| [32] | https://www.kernel.org/doc/Documentation/ABI/testing/procfs-diskstats |
| [33] | https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-sparkcontext.html |
| [34[ | https://github.com/databricks/spark-csv |
| [35] | http://www.agpd.es/portalwebAGPD/canaldocumentacion/legislacion/estatal/common/pdfs/Constitucion_es.pdf |
| [36] | http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf |
| [37] | http://docs.scala-lang.org/style/ |
| [38] | http://www.sqlstyle.guide/ |
| [39] | https://www.epo.org/law-practice/legal-texts/html/epc/2016/e/ar52.html |


Below there is a list with the Bibliography used to develop the application and document.


- Distributed Systems Principles and Paradigms; Second edition; Authors: Andrew S. Tanenbaum and Maarten van Steen.
- https://en.wikipedia.org/wiki/Distributed_computing
- https://en.wikipedia.org/wiki/Apache_Spark
- https://en.wikipedia.org/wiki/Functional_programming
- https://en.wikipedia.org/wiki/Big_data
- https://en.wikipedia.org/wiki/CAP_theorem
- https://www.tutorialspoint.com/apache_spark/apache_spark_installation.htm