Deusto
Facultad de Ingeniería
Universidad de Deusto
Ingeniaritza Fakultatea
Deustuko Unibertsitatea

Máster Universitario en Automatización, Electrónica y Control Industrial
Industria Automatizazio, Elektronika eta Kontroleko Unibertsitate Masterra

Proyecto fin de máster
Master amaierako proiektua

Big Data Pipeline for an ETL Process and Real Time Visualization of Data Generated by PLCs in an Industrial Environment

David Zamora Arranz

Director: Ignacio Angulo Martinez

Bilbao, 31 de Agosto de 2020

Summary

This project consists of the development of a big data pipeline that allows the extraction, transformation and loading of the data generated by a large number of PLCs at the same time in a database and the ability to visualize the data in real time.

The development of this project demonstrates how the pipelines developed with big data technologies such as Apache ZooKeeper, Apache Kafka and Apache Spark, in combination with the set of libraries that Apache PLC4X provides, are a very powerful tool to treat and manage the data of a large number of PLCs in real time.

Descriptors

Apache PLC4X, PLC, ETL, Apache Kafka, Apache Spark.

General Index

.

List of Figures

List of Tables

1. INTRODUCTION

Industrial communications are an essential part of industrial processes and have enabled the evolution of industrial automation and the improvement of the characteristics of PLCs.

Over the years, new protocols and more advanced forms of communication have emerged, which has resulted in advances in the field of automation.

For all these reasons, the world of PLCs has experienced great growth and industrial communications have helped these PLCs to perform more advanced and precise tasks and to be able to communicate from further distances and at greater speed.

1.1 PROBLEM DESCRIPTION

Although industrial communications have experienced a boom in recent decades, the techniques to access data from PLCs have not advanced adequately, which has caused many industrial companies to have great difficulties to extract, process and store in a simple way the data generated by the PLCs.

The fact that different PLC manufacturers use different communication protocols has made the access and handling of this data even more difficult.

On the other hand, factories have seen growth in industrial automation, which has led to a large increase in the number of PLCs used. Due to this, another big problem has arisen which consists of how to access the data of a very large number of PLCs and how to handle the massive amount of data generated by the PLCs in real time.

Therefore, the problem can be summarized in how to access the data of a large number of PLCs, which can use different protocols and handle this massive data in real time in an easy way, being able to visualize and store it.

1.2 STATE OF THE ART

1.2.1 Basic Concepts of Industrial Communications

To explain the state of the art it is necessary to briefly introduce the basic concepts of industrial communications.

Industrial communications play a very important role in the field of industry and more specifically in the automation sector.

It is common to break down industrial communication needs in the form of a pyramid made up of several levels. At the upper levels we find the operational and management levels and it is at this level where we can locate the SCADA systems. The intermediate level corresponds to the control level, and it is at this level where the PLCs are. And finally, the lowest level corresponds to the field level in which the physical devices such as sensors, actuators ... etc. are located.

The following image shows a simplified representation of the pyramid of industrial communications.

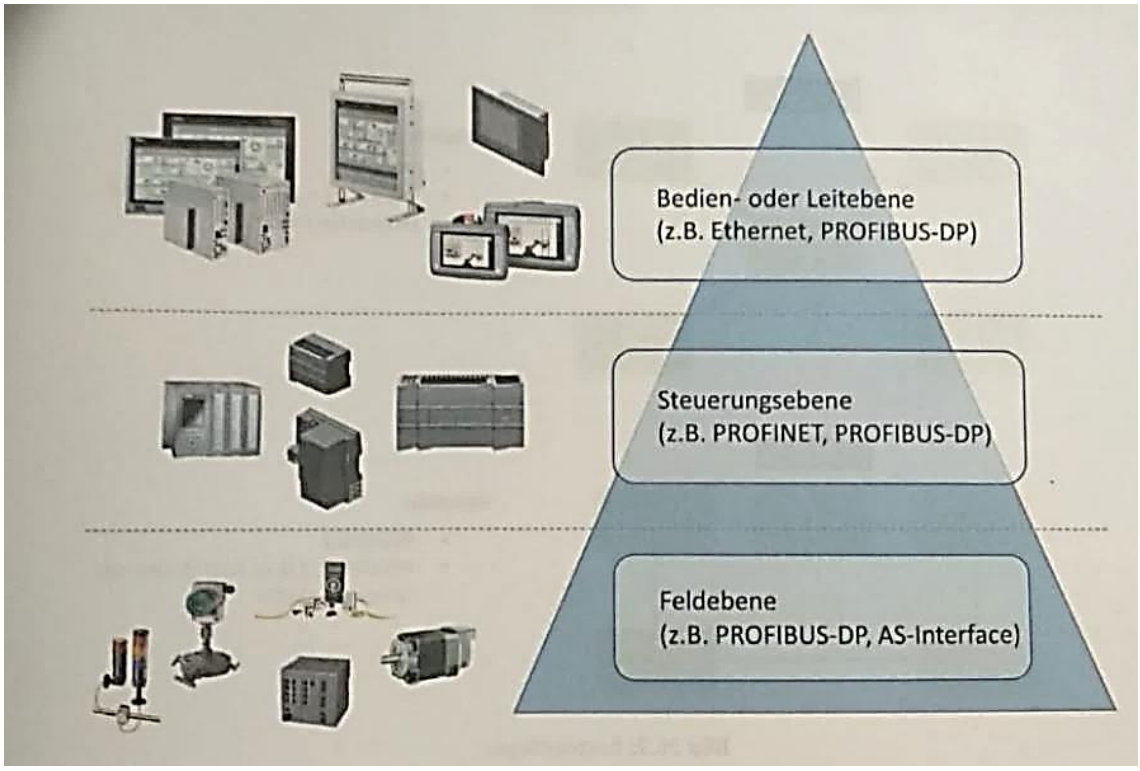


Figure 1. Communications Pyramid

For many years, industrial networks have been mainly monopolized by two different types of networks. The so-called fieldbuses and networks based on Ethernet (Industrial Ethernet).

At first fieldbuses were used in a majority way at all levels of the pyramid and today Profibus DP is the most used fieldbus in the industry. Over the years, Industrial Ethernet-based networks, like for example Profinet, have been gaining ground at the highest and middle levels of the pyramid, reaching 64% of the market by 2020, while fieldbuses are in a time of clear decline.

The experts foresee that the industrial networks based on Ethernet will continue to increase their implantation in the industry and that it will also conquer the lowest levels of the pyramid of industrial communications.

The following pie chart represents the current percentage of implementation of the most used communication protocols in the industry. The percentage of fieldbus networks is 30% (35% the previous year) and the percentage of Industrial Ethernet networks is 64% (59% the previous year). **[11]**

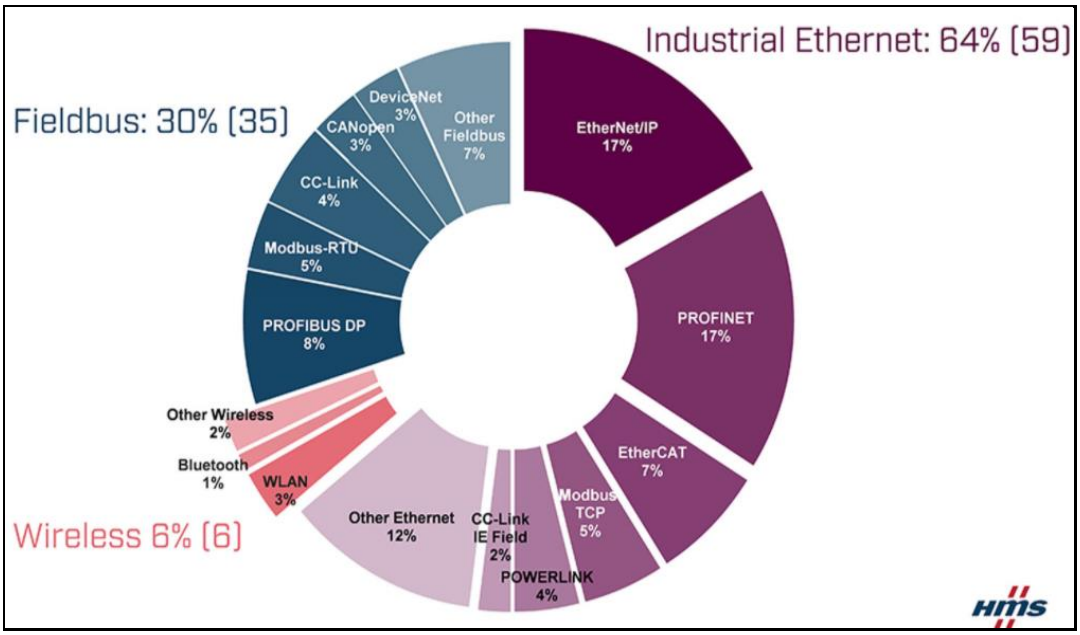


Figure 2. Protocols in the Industry

1.2.2 Current Status of PLC Data Handling Techniques

As can be seen in the previous pie chart, there is currently a great variety of different protocols. Therefore, there is some difficulty in being able to communicate devices from different manufacturers (which tend to use different protocols) or create applications that communicate (either by reading or writing data) with PLCs that use different protocols.

It should be noted that this difficulty is not insurmountable and that there are ways to deal with it successfully, although usually at the cost of investing at least a lot of additional time.

At this point, the current state of the techniques used by companies to communicate with PLCs from different manufacturers and manage their data are complex and disparate. Among them are: open communications, open source libraries, closed source software or even communications through OPC-UA.

There is currently no method that stands out especially or has had significant market penetration, except for the increasing implementation of the OPC-UA protocol (often with the server embedded in PLCs) by many manufacturers that would allow easy intercommunication between devices from different manufacturers.

While being able to solve the problem of communication with PLCs from different manufacturers and different protocols, it is also necessary to deal with the problem of handling the data from a large number of PLCs, often in real time, and this is where the difficulties increase considerably for companies.

Once it has been decided how the communication will be carried out, each company must decide how to treat the flow of data and its use, either to use it immediately or to store it.

There is also no particularly remarkable solution for this purpose. However, it is common to choose between two alternatives: the use of non-free software that performs these functions in

a highly automated way, avoiding the hardest part of programming for the company, or using open source libraries and programming one or more applications that make use of these libraries to extract the information from the PLCs.

1.3 ALTERNATIVES

1.3.1 Open Communications

Some manufacturers, apart from having their own communication protocol, also provide a way to establish open communications. Open communications allow to communicate with devices of different manufacturers.

For example, Siemens has its S7 protocol, which is used to communicate Siemens equipment. However, if you want to carry out communication between Siemens equipment and equipment from another manufacturer, you can use open communications to enable this communication. Siemens has the TSEND_C and TRCV_C instructions that serve this purpose.

These open communications also make communication between a PC and a PLC possible, since we can create a program on the PC side that communicates with the PLC by reading or writing data in it, making use of these open communications.

However, the configuration of these blocks must be done individually for each PLC, which causes a great waste of time if it is necessary to configure many PLCs.

Also, if we wanted to develop a PC application that communicates with devices that use different protocols, such as a Siemens S300 and a Modicon with Modbus TCP protocol, we would have to program the logic in our PC application to be able to connect with the Siemens PLC and the logic to be able to send and receive the frames required for the Modbus TCP protocol.

Through this example it can be easily understood that the programming of an application that accesses many PLCs at the same time and with different protocols, becomes very expensive to develop and has very limited scalability possibilities.

1.3.2 OPC-UA

OPC-UA, standardized under the IEC 62541, is an evolution of the old OPC standard to create a service-oriented architecture that allows increasing the interoperability of assets deployed in an industrial environment, regardless of the platform used.

Among the features of OPC-UA are authentication and confidentiality security mechanisms, node discovery or an object-based information model for simplified access to data.

More and more PLC manufacturers are incorporating this architecture into their systems. For example, Siemens already incorporates OPC-UA servers in its SIMATIC S7–1500 controllers. Omron also has PLCs with an embedded OPC-UA server and Beckhoff also allows them to be deployed within its TwinCAT software.

Although this standard is rapidly gaining popularity and is significantly increasing its acceptance and implementation in the market, it has not yet been adopted by most manufacturers of industrial products.

In addition, the fact of needing a server, although it may be embedded in the PLCs, reduces the transmission speed and the performance, something widely discussed in the industrial field.

1.3.3 Closed Source Software

There are companies that have developed non-free software to access PLC data and handle this data, being able to send it to a wide variety of databases.

This type of software has two main disadvantages: the need to pay for licenses and the impossibility to access or modify the source code, which can lead to mistrust about data security or the integrity of the code.

1.3.4 Open Source Communication Libraries

In order to communicate with the PLCs there are some famous open source libraries such as Libnodave, Snap7, and many others. These libraries allow communication with PLCs.

1.3.4.1 Libnodave

Libnodave is a library of functions to communicate with Siemens PLCs using MPI / PPI or Ethernet adapters. It can be used on Linux or Windows, with a large number of programming languages and supports S7-200, S7-300, S7-400 PLCs. **[22]**

1.3.4.2 Snap7

Snap7 is an open source, 32/64-bit, multi-platform Ethernet communication suite for interfacing natively with Siemens S7 PLCs. It fully supports S7-300 and S7-400 PLCs and partially supports S7-1200 and S7-1500 PLCs. **[25]**

1.3.4.3 Apache PLC4X

According to its official definition, "*PLC4X is a set of libraries for communicating with industrial programmable logic controllers (PLCs) using a variety of protocols but with a shared API"* **[4]**

This is a particularly striking project for several reasons: It is open source, it allows communication using a very large variety of protocols, it has a very broad integration with other technologies, including Apache Kafka, and it is a project of the Apache foundation, something which is synonymous with safety and quality.

1.4 JUSTIFICATION OF THE CHOSEN ALTERNATIVE

The solution to the problem of "*how to access the data of a large number of PLCs, which can use different protocols and handle this massive data in real time in an easy way, being able to*

visualize and store it?" is complex and requires the analysis and choice of multiple parts as explains below.

First, it is necessary to consider the type of architecture to be used to manage the data flow and to keep the system in a safe and stable operating mode. Secondly, it is necessary to define the technologies that will be part of the architecture and the way in which they will intercommunicate. And finally, it is necessary that all these elections meet the requirements specified in the project.

1.4.1 Architecture

Regarding architecture, a Big Data architecture has been chosen in the form of a pipeline. This type of architecture allows the handling of large data flows in an effective way and facilitates the integration of the various technologies that will be necessary to use.

The following diagram shows the architecture of the pipeline.

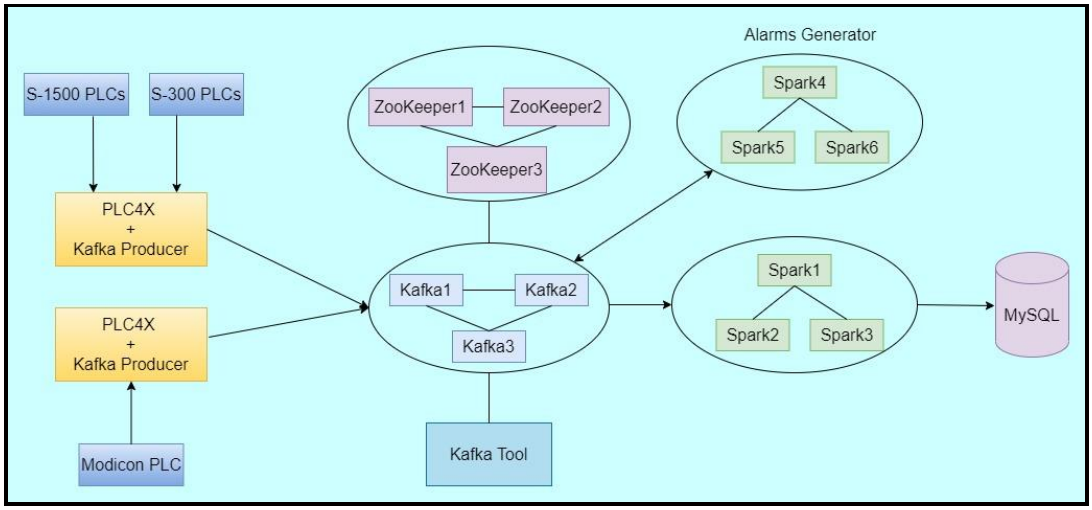


Figure 3. Pipeline Architecture

All parts of the pipeline have been containerized using Docker containers since Docker brings a wide variety of advantages to this pipeline, which are explained below:

In the first place, the encapsulation of the applications or technologies and their nodes in containers provides a very high degree of modularity that will give us very precise control of each of the parts that make up the pipeline. This control allows nodes or applications to be deployed or shut down without affecting the operation of the others.

Secondly, Docker, together with the use of Kafka (which will be explained later) greatly facilitates the decoupling of data producers from consumers, which provides great facility to continue integrating more systems in the future and the security that if a system stops working for some reason, it will not affect the rest of the systems.

Furthermore, this decoupling makes it possible to stop one or more data producers / consumers to be able to make updates without having to stop the entire system as it usually happens in traditional and strongly coupled applications. In this way, for example, if the system that generates the alarms stops working or we need to stop it to deploy another with new

functionalities, the rest of the system continues extracting the data from the PLCs and the flow is not interrupted, finally being stored in the database.

Third, containerization together with the type of technologies that have been chosen and which will be explained later, allows for practically unlimited horizontal scalability.

Fourth, all parts of the pipeline belong to one or two communication networks (depending on the needs of each application or cluster) created within Docker, which isolates these technologies from being accessed by applications that do not belong to these networks. In this way we also achieve an extra degree of security for the system.

And finally, the fact of containerizing the entire system allows a very fast deployment of the entire pipeline in any type of scenario, either physically or in cloud providers. Once the pipeline is deployed, a future migration from physical computers to a cloud service would be relatively quick and cheap.

1.4.2 Technologies

Regarding the chosen technologies, each one of them is detailed below, grouped into categories, with a brief explanation.

1.4.2.1 Data Extraction

For the reading of data from the PLCs, PLC4X has been chosen for several reasons, among which are that it provides access to a very high number of protocols using the same API, something that greatly facilitates not only access to data from PLCs that use different protocols, but also application programming. Furthermore, PLC4X has direct integration with very important technologies, including Apache Kafka and it also has tools for systematic data extraction based on triggers and the management of Pooled Connections that greatly facilitates the management of connections when accessing many PLCs at the same time. These three points, make it the perfect choice for this system.

1.4.2.2 Streaming Data Flow

As a central part of the pipeline, Apache Kafka has been chosen, which is an event streaming platform that allows us, among other things, to decouple the data producers from the data consumers, ensuring sufficient capacity for any volume of streaming data because its cluster is horizontally scalable. It also provides great security to the system against failures since if any part of the pipeline stops working due to a failure, when it recovers from the error, it will be able to continue reading the Kafka data right where it left off. So, information loss is avoided.

Furthermore, as Kafka is a distributed system and specially designed to avoid failures, if any of Kafka's brokers (nodes) fail, the rest will continue working normally since each broker has replicas of the data of the other brokers (if this type of configuration is chosen).

And also, Kafka is a horizontally scalable system, so if more computing power is needed, it will simply be necessary to add more brokers. This ensures that if the amount of data flow change in the future, the system can be adapted very quickly and easily.

1.4.2.3 Data Transformation and Data Load

For the transformation and loading of data, it has been decided to use Apache Spark which is, according to its official definition, “a unified analytics engine for large-scale data processing”. **[2].**

Spark is a very powerful processing engine, capable of processing massive amounts of data in a distributed way that provides this pipeline with the necessary power to be able to work with a very large amount of data while allowing, in the same way as Kafka, horizontal scalability that ensures easy and cheap adaptation if requirements change in the future.

1.4.2.4 Data Storage

There is a wide variety of possibilities for storing the data. This pipeline, through the use of Spark, gives us the ability to easily store data in any type of data storage. In this way we could store the data in SQL databases, in NoSQL databases, in distributed file systems such as Hadoop's HDFS or in other types of data lakes in a very easy way. In this project, it was decided to use a MySQL database for several reasons.

The first reason is that SQL databases are widely known by programmers of industrial companies and avoids the problem of having to look for qualified personnel or train existing personnel in NoSQL databases or Big Data techniques to be able to use the Hadoop's HDFS or other distributed file systems or data lake.

Second, the type of this database, that is to say, relational, stores the information in rows and is something that adapts well to the type of information that the PLCs generate so it can be saved and can be consulted later, in an efficient way.

However, the choice of the storage system for the data depends directly on the needs of each company. That said, there may be use cases in which it is more appropriate to use SQL databases, others in which it is more appropriate to use NoSQL databases, and other cases in which it is better to use another type of storage.

1.4.2.5 Cluster Nodes Coordination

According to its official definition, “*ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services*” **[6]**

Kafka uses ZooKeeper which helps with the coordination between the Kafka brokers. ZooKeeper stores a list of brokers that belong currently to a cluster. This way if a broker, due to any kind of problem (like for example broker stopping or network connectivity problem) loses connectivity to ZooKeeper, Kafka components will be notified.

In this project a ZooKeeper cluster with 3 brokers has been implemented. In this way, if one of the nodes fails, the others will continue working normally, thus achieving a higher degree of security than working with a single node.

In the same way as Kafka and Spark, this technology is horizontally scalable, so if it were necessary to increase the capability of this cluster, it would be enough to add more nodes.

Although in this project the Spark cluster does not use ZooKeeper due its kind of implementation, in the future if it would be necessary to work with Spark in a multi master

configuration, this ZooKeeper cluster could be used for that purpose, helping to coordinate the masters, without the need to make any changes.

1.4.2.6 Data Visualization

For this purpose, it has been decided to use the software called Kafka Tool. Kafka Tool gives us the ability to consult the information stored in the Kafka Topics.

It is important to be able to look up the data in real time, that is, as soon as it has been generated, so it is necessary to carry out the query directly in the Kafka topics, as this technology does, and not in the final database because that would lead to a certain delay.

Although it does not have a very modern appearance nor does it offer a large number of options like other more visual data visualization tools, it covers the most important needs that both a programmer and the end user of the pipeline may have, which are the following:

First, it allows access to the information stored in the topics in real time.

Second, it provides detailed information about the data stored in the topics, including timestamp, key, value, partitions, ... etc. and it allows displaying the data of the topics in different formats such as text, json, xml and hexadecimal.

Third, it displays the list of consumers for a given Kafka cluster.

And finally, it allows a search to be carried out on the data displayed on the screen, filtering a text entered by the user by key, value, regular expression ... etc.

For all these reasons, this technology has been chosen.

1.4.3 Fulfilment of Requirements

Both the chosen architecture and the technologies used meet the requirements set out in Chapter 5 (entitled “Project Design”).

For this reason, and considering all the requirements, it should be noted how this pipeline has been developed using mostly free or open source technologies, with a design based on modularity and decoupling of its components, as well as with a very high-performance capability and horizontal scalability.

2. PROJECT DESCRIPTION

2.1 GENERAL AND SPECIFIC OBJECTIVES

The main objectives of this project are the following:

1. Develop a system, mostly with open source technologies, with the capability to extract data from a large number of PLCs that can use different protocols while allowing these data to be visualized in real time, handling them in such a way that it can be transformed if necessary, and stored in a final storage system.
2. Integrate the following capabilities into the system: Horizontal scalability, high performance, possibility of integrating new applications to the system and easy deployment both in physical systems and in cloud providers.
3. Design the architecture in a modular way that allows an expansion of the architecture in the future or replacement of the technologies used.
4. Carry out the configuration of the necessary parameters to ensure a good interconnection between the different technologies and optimal performance.
5. Carry out the configuration of the necessary parameters to ensure a good interconnection between the different technologies and optimal performance.
6. Test the system to verify that it works correctly and that it provides the desired performance.

2.2 PROJECT SCOPE

The scope of the project is described below, considering the project objectives and the requirements specified by the client.

This project develops, and includes within its scope, a system that accesses the data of a large number of PLCs, which can use different protocols and handle this massive data in real time in an easy way, being able to visualize and store it.

The scope of this project includes the creation of the architecture that will allow the interconnection of the different technologies and applications that are necessary for the correct operation of the system, offering an effective way of intercommunication between them and facilitating the movement of the data flow.

The scope of this project also includes the necessary configuration of the technologies used for its correct and efficient operation and the programming of four applications that will be in charge of extracting the data from the PLCs, generating the alarms based on that information and sending all the data to the final data storage system.

All the technologies or applications to be developed in a cluster will be made up of 3 nodes.

Any possible expansion of the clusters, that is, addition of the number of nodes, is outside the scope of this project.

Any possible expansion of the architecture, that is, expanding the number of clusters or creating new applications, is outside the scope of this project.

2.3 BLOCKS DIAGRAM

Figure number 4 shows the diagram of the system architecture made up of the technologies and applications that comprise it.

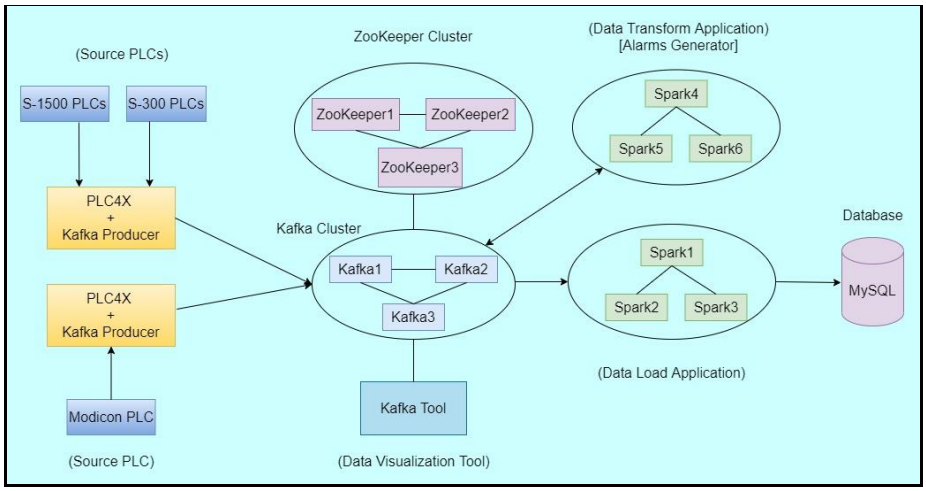


Figure 4. System Architecture

As can be seen in the diagram above, each component of the system has an accompanying text that briefly summarizes its role. Next, we will begin to explain the components from left to right.

On the far left we can see there are three different kind of PLC'S (S-1500, S-300 and Modicon). The data that they generate are extracted by the data extraction applications which consists of two PLC4X applications capable to extract that data and send it to Kafka through the use of a Kafka producer. Kafka, which is composed of a cluster with three brokers, stores temporally the data. Kafka brokers coordination is made up with the help of the ZooKeeper cluster which is located just right above and which it is also composed of three nodes.

Just below the Kafka cluster is located Kafka Tool which is a data visualization tool of the Kafka topics. The alarm generator is a Spark data transform application that is in charge of reading Kafka topics data and generating alarms, when appropriate, based on the information read from

Kafka. The alarms generated will be sent to a Kafka topic called "alarms". The data load application is a Spark application that is responsible for reading all the information available in the Kafka topics (data from the PLCs and alarms), parsing them in SQL statements and sending this data to the database for persistent storage.

2.4 FUNCTIONAL SPECIFICATION

The functional specification of this project has been tested on an MSI GL 75 95EK computer with original factory components, with an SSD disk and 16 Gb DDR4 RAM. However, given the system architecture, it is guaranteed that it is possible to distribute the clusters or their nodes between several computers, so if you do not have very powerful computers, the use of several computers to distribute the nodes would be enough to increase the performance of the system to the desired level. Below is a table that exposes the functional specification of this project.

Maximum data extraction speed from PLCs (through the data extraction application)	Theoretical speed tested with 38 PLCs from 1 ms (in the application side) with stable behaviour. (Real acquisition speed may be limited by the capability of the PLC to respond)
Maximum data transmission speed from the data extraction application to Kafka.	Every 5/20 ms or 8/32 Kb of data (limited by software by the configuration of the high-throughput producer in the 2 Spark applications).[1]
Maximum data transmission speed from the data load application to the database.	Every second (limited by software by Spark high throughput trigger setting). (Ingest speed could be affected by the ability of the database to process all queries.)
Approximate average delay from when the data is generated until it is stored in the data base.	Less than two seconds. [2]
Approximate average delay from when the data is generated until it is stored in Kafka.	Less than 200 milliseconds. [3]

Table 1. Functional Specification

2.5 PROJECT OUTPUTS

[1] 5ms/8Kb corresponds to the alarmGenerator app., 20ms/32Kb corresponds to the sparkTransformer app

[2] Speed has been approximately measured.

[3] Speed has been approximately estimated.

The result of this project is the development of a system capable of accessing the data of a large number of PLCs, which use different protocols and handle this massive data in real time in an easy way, being able to visualize and store it.

The data generated by the PLCs will be extracted by the data extraction applications built with PLC4X and equipped with a Kafka Producer each one. The data will flow to Kafka cluster that consists of a 3 brokers cluster. The Kafka's broker coordination is made up with the help of the ZooKeeper cluster which has 3 nodes.

The data transform application (called Alarm Generator) is a Spark application, which has a cluster of 3 nodes and is responsible for reading the Kafka topics and generating certain alarms based on that information. The alarms generated are sent to a Kafka topic called "alarms" to be available for any other service that may need to read them.

In this project, the inputs corresponding to the range I1.0 to I1.7 simulate the operation of valves and every time one or more are activated, an alarm is generated indicating that the valve or valves corresponding to that input are damaged.

The so-called Data Load Application is a Spark application, which also has a cluster of 3 nodes and which is responsible for reading all Kafka topics (both those that contain the data generated by the PLCs and the alarms generated by the Alarm Generator application) to parse them to an SQL query and send them to the database for storage.

The Kafka Tool is a tool that allows us the reading of the info contained in the Kafka topics and it is executed from outside of the Docker environment.

The operation of the system will be automatic and the system will start when the docker-compose is deployed.

The only part that must be started manually will be the two Spark applications to be able to choose if we want to work in distributed mode or local mode.

Once the system is started, it will work automatically and it will be possible to access the logs generated within each of the containers by using the command "docker logs containerName" where containerName has to be changed by the name of the container. The data generated by the containers will be stored in Docker volumes.

3. SOLICITATION DOCUMENT

3.1 MATERIALS AND CONSTRUCTION ELEMENTS

The system to be developed will consist of a single software part. This section explains the elements that compose it and its most relevant characteristics to consider.

Although the system consists only of a software part, it must be implemented in a hardware element (PC), so to know under what conditions and hardware requirements it must be implemented, please refer to chapter 3.2 (Requirements for Deployment).

3.1.1 Software

For the development of the pipeline, the following technologies have been used: Docker, Apache PLC4X, Apache ZooKeeper, Apache Kafka, Apache Spark, MySQL and Kafka Tool (Kafka Tool being a tool external to the pipeline). And more specifically for the development of the 3 applications that this pipeline has, Apache PLC4X, Apache Kafka and Apache Spark have been used.

This section explains the components of each of these two parts (technologies and applications) and their most relevant characteristics to consider. The role of each of these technologies and applications is explained below. In addition, the functionalities and benefits that they bring to the pipeline are also explained.

For a comprehensive and detailed explanation of each of the technologies that make up this pipeline, please refer to chapter 4.

3.1.1.1 Docker

According to its official definition "*Docker is the de facto standard to build and share containerized apps - from desktop, to the cloud*" **[**15**]**

Docker allows containerization of applications, to allow easy deployment both in physical environments and in the cloud.

In this project, it is used to containerize each application and each node of the clusters for several reasons, among which the following stand out: It facilitates rapid deployment, provides the desired modularity, facilitates the scalability of the pipeline, and facilitates individualized management of each of the parts of the pipeline (allowing to turn on, stop and turn off parts of the pipeline or nodes if needed).

All parts of the pipeline are containerized with Docker containers. The deployment is done using docker compose. Docker compose allows us to write a yaml file in which all the instructions to build the services and their parameters will be defined, helping to generate and deploy all the containers with a single command.

For a comprehensive and detailed explanation of Docker, please refer to chapter 4 (Docker section).

3.1.1.2 Apache PLC4X

According to its official definition "*PLC4X is a set of libraries for communicating with industrial programmable logic controllers (PLCs) using a variety of protocols but with a shared API*". **[5]**

In this project, two applications capable of extracting (reading) the data from the PLCs and sending that information to Kafka has been programmed.

These applications have been programmed using PLC4X that provides the ability to extract data from the PLCs and with a Kafka producer that provides the ability to send the data to the Kafka cluster.

The programming has been done with two Java applications in a Maven-type project with all the necessary PLC4X and Kafka dependencies included.

Although PLC4X has its own Kafka-Connect connector, it has been chosen to program a Kafka producer manually in order to better control all the aspects related to the programming of sending the data to Kafka. The producer has been created with settings adjusted for high throughput.

These settings include buffering the data for 5/20 milliseconds (depending of the Spark application) or a certain minimum amount of Kb (8/32) before being sent to increase the throughput at the expense of increasing a little bit the latency and these settings also include compression of the sent messages (with snappy compression method) which translates into an increase in performance of the network avoiding collapse it at the expense of a little increase in the CPU working time.

The existence of a schema in the data that is transmitted is of vital importance in the world of Big Data. Two important reasons that support the need for a schema can be to avoid the difficulty of debug data corruption issues and to be able to adapt to future changes in the data through the schema evolution.

Therefore, the information extracted through PLC4X is parsed and converted to JSON format to create a basic schema that will facilitate the treatment of the data for the programs that will read it from Kafka later (such as Spark applications).

For a comprehensive and detailed explanation of the schemas, please refer to chapter 4 (Apache Kafka section).

For a comprehensive and detailed explanation of Apache PLC4X, please refer to chapter 4 (Apache PLC4X section).

3.1.1.3 Apache ZooKeeper

According to its official definition, "*ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services*" **[7]**

As explained earlier, Kafka uses ZooKeeper which helps with the coordination between the Kafka brokers. ZooKeeper stores a list of brokers that belong currently to a cluster. This way if a broker, due to any kind of problem (like for example broker stopping or network connectivity problem) loses connectivity to ZooKeeper, Kafka components will be notified.

In this project a ZooKeeper cluster with 3 brokers has been implemented. In this way, if one of the nodes fails, the others will continue working normally, thus achieving a higher degree of security than working with a single node.

For a comprehensive and detailed explanation of Apache ZooKeeper, please refer to chapter 4 (Apache ZooKeeper section).

3.1.1.4 Apache Kafka

According to its official definition, "Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications." **[2]**

In this project, a Kafka cluster with 3 brokers has been created. This cluster is the central core of the pipeline and is responsible for receiving the data from the producers (in this case there is only a single application that extracts the data from the PLCs), stores it temporarily, and allows any consumer or group of consumers (in this case the two Spark applications) access this data.

In addition, Kafka has also been used, as explained above, to create the producer in the data extraction application.

Kafka is widely used worldwide because it provides impressive features since it allows the management of streams in real time with a very high performance, it allows to serve a large number of consumers and producers at the same time, it helps the decoupling of systems (as in this project) and above all because it guarantees extremely high security against failures or data loss.

This security that it provides against data loss or failures is due to several factors, among which are: it is a distributed system, so its brokers can be distributed in different servers so that if network failures occur, they will not affect all the brokers at the same time. In addition, the brokers can be configured to save replicas of the data of the other brokers so that if any broker stops working, the others can continue serving the data to consumers.

And as a final point, Kafka has an offsets system that can be used by consumers (as in this example in the two Spark applications with their checkpoint mechanism) to know where they are reading, so that if a failure occurs and the consumer stops, when it is running again, it will be able to resume reading Kafka data exactly from where it left off.

For a comprehensive and detailed explanation of Apache Kafka, please refer to chapter 4 (Apache Kafka section).

3.1.1.5 Apache Spark

According to its official definition, “Apache Spark is a unified analytics engine for large-scale data processing.” **[6]**

In this project, two Spark clusters with 3 nodes each have been created. A Spark application has been programmed for each cluster.

One of them is the so-called Alarm Generator, which is responsible for reading the Kafka topics data, which is the data that was previously extracted from the PLCs, and based on that, generating the pertinent alarms.

The other Spark application is in charge of reading the Kafka topics (both those containing the information of the PLCs and the “alarm” topic), parsing everything in SQL query format and inserting the information into the MySQL database.

Spark applications have been programmed with the Scala programming language since it is Spark's native language and is the one that provides the highest performance. **[23] [14] [18]**

In addition, both applications have been programmed with a checkpoint system that allows recovering from an error and being able to continue reading Kafka data right where they left it.

For a comprehensive and detailed explanation of Apache Spark, please refer to chapter 4 (Apache Spark section).

3.1.1.6 MySQL

MySQL is an open-source relational database management system (RDBMS). The data in MySQL is stored in tables and can be queried using SQL language.

In this project a MySQL database has been created to store all the data that is generated in the pipeline in a persistent way.

For simplicity's sake it has been developed with a standalone topology, which means there is a MySQL instance designed to run as a single node with only one master process.

However, this database has the capacity for scaling, so if it were necessary in the future, the topology could be changed to work with more nodes (either by adding slaves or masters), requiring a reconfiguration.

For a comprehensive and detailed explanation of MySQL, please refer to chapter 4 (MySQL section).

3.1.1.7 Kafka Tool

According to its official definition, “Kafka Tool is a GUI application for managing and using Apache Kafka clusters. It provides an intuitive UI that allows one to quickly view objects within a Kafka cluster as well as the messages stored in the topics of the cluster. It contains features geared towards both developers and administrators.” **[20]**

In this project, the Kafka Tool has been used both to debug the applications during programming and to display data once the application is running.

3.2 REQUIREMENTS FOR DEPLOYMENT

The following are the minimum requirements for both software and hardware that must be met for the correct operation of the system.

3.2.1 Software Requirements

Since the pipeline is delivered with the architecture built and configured, no software version changes are expected to be made to it.

It is expected that if it is necessary to expand the number of nodes to scale the system, it will be done with the same versions of the existing technologies in the pipeline at the time of delivery.

If for any reason the version of some of the technologies or applications in this pipeline were to be upgraded or downgraded, it is possible that it will stop working correctly because the changes to versions of the technologies usually lead to structural changes and changes in the versions of the applications may affect their programming.

3.2.2 Hardware

As explained in chapter 2.4 (Functional Specification) this pipeline has been tested on an MSI GL 75 95EK computer with original factory components, with an SSD disk and 16 Gb DDR4 RAM. However, given the system architecture, it is guaranteed that it is possible to distribute the clusters or their nodes between several computers, so if you do not have very powerful computers, the use of several computers to distribute the nodes would be enough to increase the performance of the system to the desired level.

3.3 SYSTEM VALIDATION

In this section, a process will be defined to check if the pipeline meets the desired requirements.

3.3.1 Validation Requirements

Therefore, the characteristics and tests that the device must pass in order to be validated will be defined below:

- The pipeline must be able to operate without failure and store the data of the PLCs and the alarms generated if all its parts are in operation.
- The pipeline must be able to operate without failure and store the data of the PLCs if all its parts except the alarm generator are operating.
- The pipeline must be able to operate without failure and generate alarms if all its parts except the data load application are running.

- The pipeline must remain in a stable state if there is no producers connected, keeping it waiting without the other systems presenting an error.
- Consumers should be able to pick up reading Kafka data from where they left off before stopping

3.3.2 Validation Tests

To validate the correct operation of the system, the following tests must be passed successfully:

- Turn on all parts of the pipeline and check that the pipeline is able to operate without failure and store the data of the PLCs and the alarms generated if all its parts are in operation.
- Turn on the pipeline except the alarm generator and check that it is able to operate without failure and store the data of the PLCs
- Turn on the pipeline except the data load application and check that it is able to operate without failure and generate the alarms
- Turn on the pipeline and stop the producers to verify that the pipeline remains in a stable state, keeping it waiting without the other systems presenting an error.
- Turn on the pipeline, launch consumers, and verify that consumers are able to pick up reading Kafka data from where they left off before stopping

3.4 WARRANTY TERMS

3.4.1 Technical Support

Start-up and bug-fix support is offered free of charge for the next 24 months after project delivery and final start-up.

Within this guarantee, the support offered in the system during the guarantee months after the final commissioning of the pipeline, will include a maintenance service as described in the section 3.4.2.

3.4.2 Maintenance Service

During the warranty period, one maintenance day per month is offered to perform the following maintenance tasks:

- Software updates
- Reconfiguration of parameters to improve throughput
- System performance check

4. GENERAL CONCEPTS AND TECHNOLOGIES

Although in chapter 3 the use of all the chosen technologies has been justified, this chapter 4 is dedicated to explaining and accurately describing each of the technologies used in this project in a general way. Since many of the technologies used are relatively modern and complex, it is considered absolutely necessary to write this chapter as it can serve to avoid doubts and misunderstandings that may arise about the characteristics of the technologies used and justified in chapter 3.

The description of most of the technologies that are going to be described in this chapter will be referenced directly from my Bachelor Thesis **[30]** but keeping in any case the references to the original texts of the authors that are cited in it.

4.1 APACHE KAFKA

4.1.1 The Concept of Publish/Subscribe Messaging

It is very important to understand how this messaging pattern works to be able to understand how Apache Kafka works.

In this messaging pattern the senders are called publishers and the receivers are called subscribers. The piece of data to be sent is called message. When a message is going to be sent by the publisher, it is not directed directly to the subscriber. Instead, the publishers (senders) categorize the published messages into classes and the subscriber (receiver) subscribes to receive the classes in which it is interested.

In this kind of messaging systems, the publishers and the subscribers are not directly connected and it is usual that the publish/subscribe systems have a broker which is an intermediate point where messages will be published.

These kinds of systems that use a message broker to send messages are also known as messages queue. A message queue can be understood as a kind database which is specially designed to work and handle efficiently and very quickly message streams.

One of the advantages of this system is it can manage more efficiently the connections, disconnections and the crashes due to its centralized broker nature. The broker can be configured to determine how much time it must maintain the data. The retention can be performed by writing it to disk (in which case if occurs a broker crash the data will not be lost) or by keeping the messages in memory, depending on the specific system and its configuration.

In a publish/subscribe messaging system the consumers are normally asynchronous which means when the publisher sends a message, it does not care about if the message has been consumed by the subscriber. The only thing it does is to wait until it receives the broker

acknowledgement indicating the message has been received correctly and it has been buffered. And later, the message will be delivered to the consumers very fast if possible (often milliseconds) or with more delay if there is a queue backlog.

The benefits of this kind of systems are they allow good network scalability in addition to a very dynamic network topology.

4.1.2 The Concept of Point to Point Connections and Individual Queue Systems

When there is only one application that needs to send information somewhere is usual to think in the use of a point-to-point (direct messaging) connection between the producer and the consumer. However, as the number of applications increases and they need to connect to shared services to obtain data, the direct messaging system becomes quickly complicated and difficult to maintain.

It is easy to appreciate this situation thinking in a specific use case, adapted from the book "*Kafka The Definitive Guide*" [54]. For example, if a company have multiple servers and multiple applications that are using those servers to obtain individual metrics to use it for different goals, the architecture could be something like it is shown in the figure number 5.

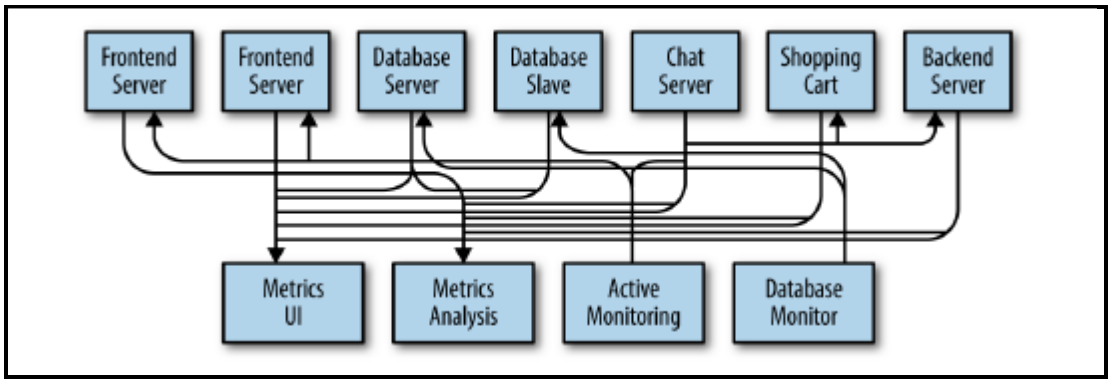


Figure 5. Multiple Servers and Applications Architecture

Looking at this kind of architecture it is easy to appreciate that is not a good architecture to use not only because it is difficult to maintain but also because it makes scalability very difficult if we have the need to continue adding more servers or applications in the future.

The solution for this problem could be to build a publish-subscribe system with only one application that receives all the metrics and to provide these metrics to all the systems that need them.

The figure 6 shows the architecture of this example implemented with a publish-subscribe system.

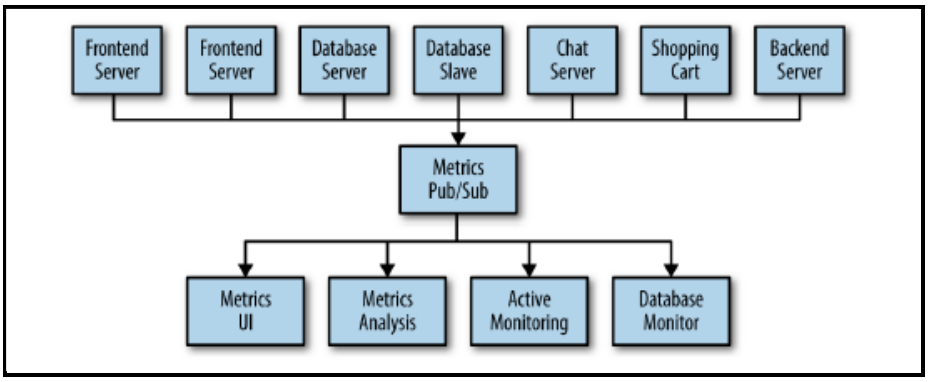


Figure 6. Publish-Subscribe System

With this approach it is easy to improve quickly the architecture reducing the complexity and facilitating the scalability. There are situations where different departments of a company or different workers could have been working by separate implementing their own individual queue (publish-subscribe) systems for their needs with the result of an architecture with the publishers and subscribers decoupled but with a lot of duplication (due to maintain several individual queue systems) as we can see in the figure 7.

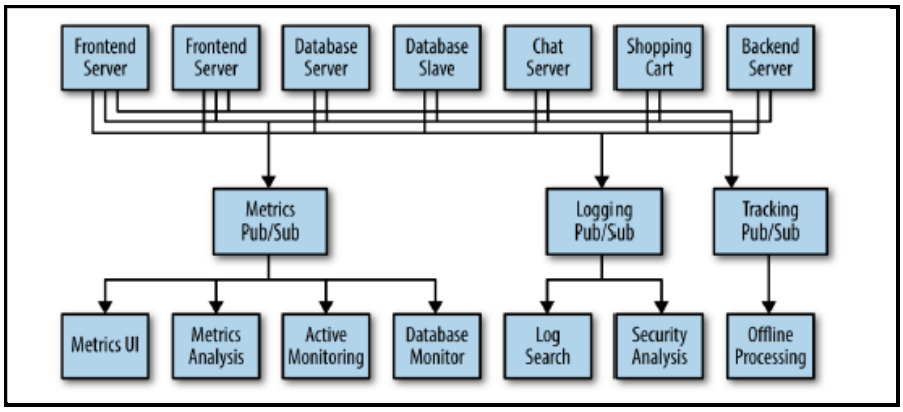


Figure 7. Publish-Subscribe Duplicity

To avoid this kind of duplicities, a good solution is to have a single centralized system that enables for publishing generic types of data. In this way the architecture complexity will be reduced drastically and the scalability will be increased extremely. And this is where Kafka comes into play.

4.1.3 What is Apache Kafka?

Apache Kafka is a fast, scalable, durable, and fault-tolerant publish/subscribe messaging system. Specifically, Kafka's official website defines it as a "*distributed streaming platform*", although it is not difficult to find other definitions where Kafka is defined as a distributed commit log.

The data stored in Kafka is stored in order and can be read deterministically. Also, the data can be distributed within the system which allows a high fault-tolerant degree at the same time that it provides a great capacity for scalability.

4.1.3.1 Commit Log

A commit log is an append-only record of all transaction. It is used by the producer to send a message by appending it to the end of the log and on the other hand, the consumer will read the log sequentially to obtain the stored messages.

It is important not to be confused with the application logs which are a different topic. In short, this data structure called commit log is simply a time-ordered sequence of data which in Kafka is represented by an array of bytes.

4.1.3.2 Messages and batches

A message is an array of bytes which have not a specific format or meaning for Kafka. Every message can be accompanied or not with a key. The key is optional metadata consisting of an array of bytes. Keys are very useful when the messages are written in different partitions because they allow an easy way of controlling them.

The way in which the messages are written in Kafka is in batch. This is the way Kafka becomes very efficient because it can achieve to manage many messages per unit of time achieving a high throughput.

The reason why Kafka does not send the messages one by one across the network is because it would result in a great overhead. It is true that sending the messages individually instead of in batch would achieve a lower latency (the time to propagate an individual message would be less) but it would reduce the throughput.

It is necessary to achieve a balance between latency and throughput and Kafka achieves this by sending the messages in batch for a good throughput ratio and compressing the batches for a better latency ratio at the cost of some processing power.

4.1.3.3 Schemas

As explained before, messages are an array of bytes which have not a meaning for Kafka. But nevertheless, it is possible to add a schema to facilitate the understanding of the content of the message.

It is possible to choose between some different options for message schema like for example XML or JSON. Nowadays the JSON data format is one of the most popular. It is present in almost all the languages and a lot of modern application uses it.

Between the advantages of JSON we can find that the data can take any form (nested elements, arrays…) and it is also a widely used format in all the languages and in the web. Looking at the disadvantages the most striking is the size of the objects which can be too big due to repeated keys.

A very interesting alternative is Apache Avro. Avro is a serialization framework which solves the size problem of JSON by providing a compact serialization format. The main advantages of

Avro apart from the compact serialization format are the Avro schemas are defined using JSON, the Avro serialization provides the schema and the data (it is not possible in Avro format to have only data without schema) and it is a good option to control the schema changes over time. Like all the serialization frameworks it has advantages and disadvantages. The main drawbacks of Apache Avro are the following: It takes more time the development using Avro objects than using JSON and it is not possible to read an Avro file directly with an editor because of its binary nature. Although it is possible in the future the Avro objects will be viewed from the IDEs, this is something that is not supported yet.

There are other serialization frameworks available such as Apache Thrift, Protocol Buffers (also called Protobuf), Parquet... etc. It is not possible to assure that one serialization framework is better than another in a general way. The choice will depend on the needs of each project according to the advantages and disadvantages of each serialization framework.

It is highly recommended to have a schema. A very important reason to have it is the difficulty to debug data corruption issues. In this context, it is common to find two problems that are difficult to solve if there is not an available schema. The first of them occur when an expected field does not exist and the second of them occur when the type of a specific field is not the expected (e.g. when the type of a field has changed).

4.1.3.4 Topics and Partitions

Messages in Kafka are classified into topics. A topic is a category where the records will be published. The topics can be broken down into a specific number of partitions. Partitions are an ordered and immutable sequence of records where the messages will be appended. These messages will be read later in order. All the records in the partitions have an offset which is a sequential identifier used to identify the records within the partitions.

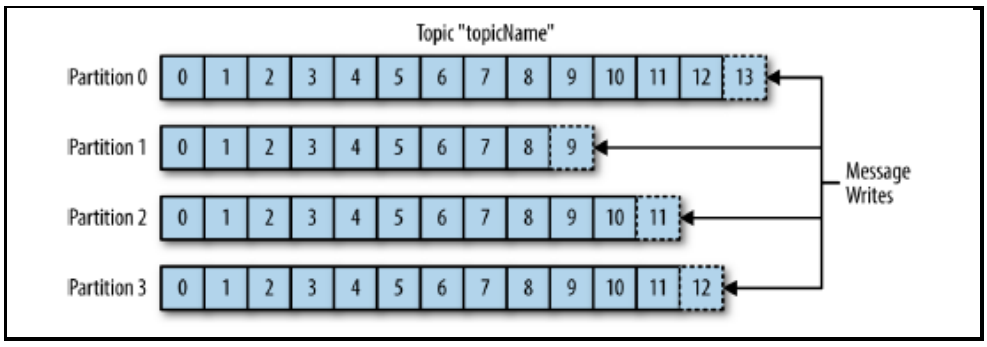


Figure 8. Kafka Topic Structure

Partitions provide multiple benefits such as redundancy and scalability. Each partition can be replicated to other servers to achieve fault-tolerance. For every partition, will be a server which will be the leader and the others (if there are) will be the followers. If each partition is hosted in a different server, it will be possible to scale up horizontally every topic across different servers, what provides a very high level of parallelism.

4.1.3.5 Brokers and Clusters

A Kafka cluster can have multiple brokers. A broker is a single Kafka server. Every broker has an identifier and partitions. To connect to a Kafka cluster, it is only necessary to connect to one broker and after that, the connection will be stablished for the entire cluster.

There is an important characteristic called replication factor and it allows specifying how many replicas of every topic will be in the cluster. In this way, the partitions can be replicated to other brokers providing redundancy of messages at the partition level. In this context, only one broker can be the leader for a specific partition. In other words, if there is one topic (Topic A) with two partitions (partition 0 and partition 1) and two brokers (Broker 1 and Broker 2), both partitions can exist in the two brokers due to the replication but only one of the two brokers can be the leader for the partition 0 and only one can be the leader for the partition 1. If a broker failure occurs, the other brokers can take over leadership of the affected partitions.

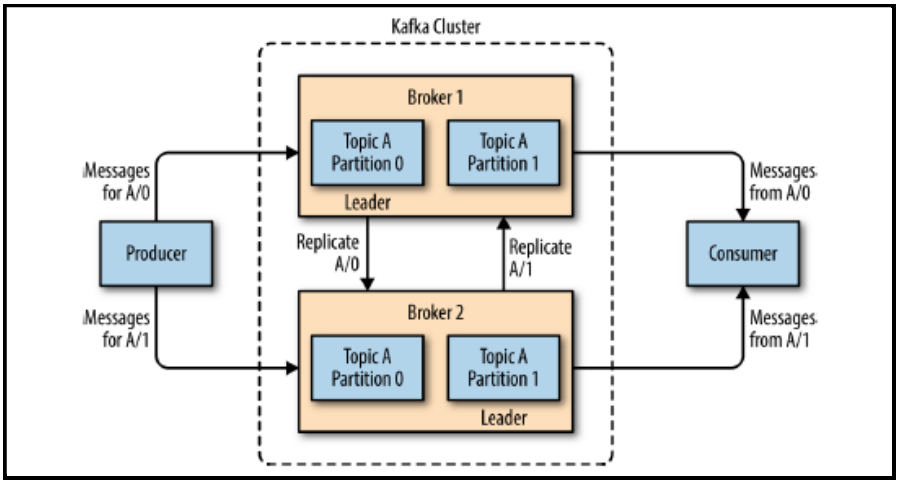


Figure 9. Kafka Cluster Replication

The figure 9 shows the replication of the partitions of one topic over a cluster with only one leader for every partition.
It is also necessary to explain that there is one broker called controller, which is elected automatically, and it will be the cluster controller. It will be responsible for state management of partitions and replicas. This means if a broker goes offline, the controller will assign a new leader to all the partitions that need it.

The retention of the messages is also a key factor of the brokers. The brokers can be configured with a retention setting based in time or in size. But not only the brokers can be configured with the retention setting. It is also possible to stablish the retention setting at the topic level that will provide more flexibility to adjust the real needs of the system.

4.1.4 Kafka APIs

Apache Kafka has four core APIs: Producer, Consumer, Streams, Connect and AdminClient. Below is the explanation of every one.

4.1.4.1 Producer API

The Producer API is used to publish the messages to the correspondent topics. Although is possible to send a message to a specific partition of a topic, normally the producer will distribute messages over all partitions uniformly.

If it is necessary to send the messages to specific partitions there are two ways to do it. One way is using a message key and a partitioner to generate a hash key to achieve all the messages with the same hash key will be written in the same partition. This is the most used method when it is necessary to direct the messages to specific partitions but sometimes it could be necessary to direct messages basing them in other logic. Then, the other way is to use a custom partitioner to stablish a different logic.

The figure below shows the process of sending data.

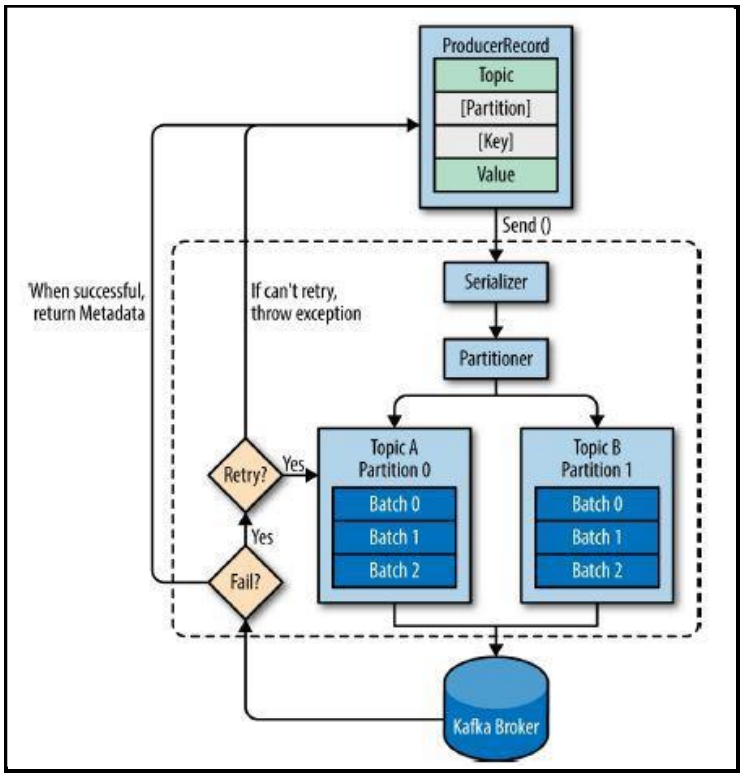


Figure 10. Process of Sending Data in Kafka

The process of sending messages to Kafka is done as follows:

1. First of all a 'ProducerRecord' is created. It includes the destination topic, a value, a key (optional) and a partition (optional).

2. The key and value objects are serialized by the 'ProducerRecord' to Byte Arrays and sent to the partitioner.

3. The partitioner will select the appropriate partition and it will adds the message (record) to a batch of records whose destinations are the same topic and partition and all of these batches will be sent to the Kafka brokers.

4. If the broker receives and write correctly the messages to Kafka, it will return a 'RecordMetadata' object with information to know where the record is located. This information will be the topic, partition and offset of the record. In other case it will return an error.

4.1.4.2 Consumer API

The Consumer API is used to read the messages from topics. Consumers are responsible for reading messages in the same order in which they were generated and to perform this task they need to keep track of the offset of the messages.

Kafka provides each message with an identifier called offset that consists of an incremental integer which identify a position in the partition. All the messages inside a partition have a unique offset.

The way Kafka scales topic reading is distributing partitions among a consumer group. A consumer group consists of one or more consumers that are coordinated to read a topic. The way the group works ensures that each partition will only be consumed by one consumer.

A consumer is able to read from one or more partitions at the same time but it is not allowed that two consumers read from the same partition. Attending at this, if a consumer group have more consumers than we have partitions then one or more consumers will be idle.

This way of working provides several advantages like scalability and fault tolerance. Regarding scalability, the consumer group allows horizontally scalability. This means it is possible to add more consumers to the group if more parallelism is needed. With regard to fault tolerance, the consumer group assures if a consumer fails, the others consumers of the group can continue consuming their partition after performing a partitions rebalance.

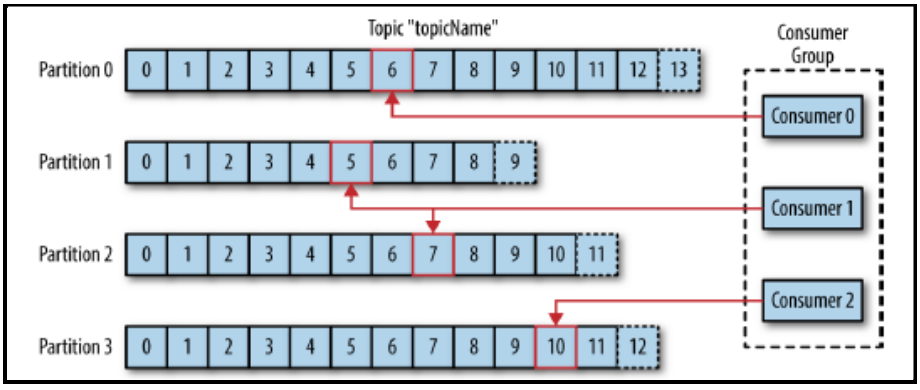


Figure 11. Consumers

4.1.4.3 Streams API

In its beginning Kafka was only intended as a publish/subscribe messaging system and mainly used to provide a reliable source of streams to the stream-processing frameworks.

However, since version 0.10, Kafka provides a stream-processing library that allows Kafka to be used like a stream processor. That means an application can consume data from topics, process it without the need of an external stream processor and produce events.

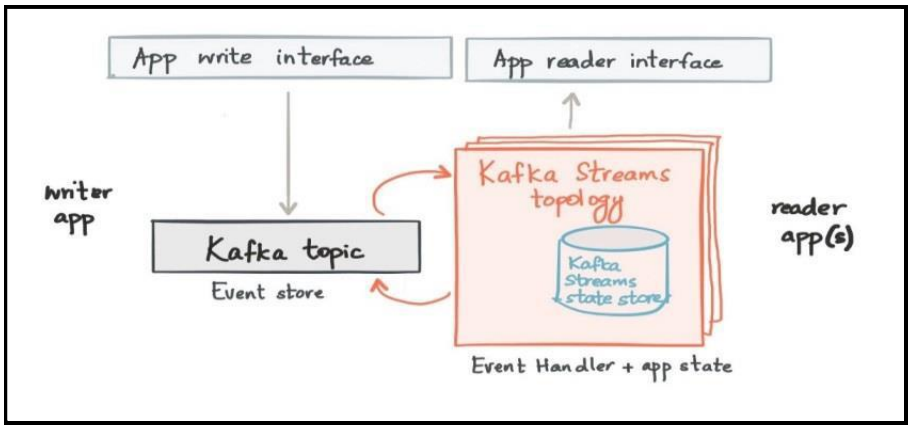


Figure 12. Kafka Streams

Figure 12 shows a diagram where it is possible to see an example of a simple use case of Kafka Streams. In this example an application send messages to the Kafka topics and Kafka Streams processes it and produces an output.

4.1.4.4 Connect API

Kafka Connect is a tool that allows creating connectors in a scalable and easy way. The connectors allow moving data into Kafka or out of Kafka. Kafka Connect can ingest large amounts of data from databases or collect metrics generated by the applications into Kafka topics, allowing immediate availability (with low latency) of the data to be used in a streaming context.

Convertors allow Kafka storing the data in different formats. Convertor for JSON format is provided natively by Kafka and the necessary convertor for Avro format is provided by Confluent which is a data streaming platform based on Apache Kafka.

4.1.4.5 AdminClient API

The AdminClient API allows managing and inspecting topics, brokers, and other Kafka objects. **[2]**

4.1.5 Delivery Guarantees

In distributed publish-subscribe systems like Kafka it is necessary to bear in mind that it can occur problems during the producer is sending data. It is important to be careful with these situations because they usually can produce data loss or duplicate the data and this is something that it is necessary to consider.

Depending how the producer handles the sending of data there will be different possible results in the case of a failure or in other words there will be different semantics. **[21] [23] [24]**

1. At least once: This approach assures the data will reach the target system at least once. This means that there can be no data loss if there is a system failure but there is a possibility the data reaches the target system more than one time producing data duplication. This situation could happen if the producer sends the data and just when the broker has written the data in Kafka it happens a failure and it has no time to send the acknowledgement (ack) to indicate to the producer it received the data so in that case when the system recovers from the failure the producer will send the data another time.

2. At most once: This approach assures the data will reach the target system at most once. This means there may be data loss if there is a system failure. This situation could arise if the producer does not resend the data when it has not received the acknowledgement.

3. Exactly once: This approach assures the data will reach the target system exactly once. This means there can be neither data loss nor duplicated data. Even if there is a failure and the producer resend the data, this approach assures the data will reach the target system only once. This is the best guarantee of all but it is difficult to achieve because it implies that the messaging system and the application which produce and consume the data must work in team.

4.2 APACHE SPARK

4.2.1 What is Apache Spark?

There are multiple definitions for Apache Spark. The official website definition is the following: “Apache Spark is a unified analytics engine for large-scale data processing”. **[6]** It is also possible to find a lot of different definitions. However, the definition provided by the book “*Spark the definitive guide*” is one of the most complete and defines it as follows: “Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters”. **[12]**

In this definition there are some key words which must be explained. Spark is a unified computing engine because it is possible to use its set of APIs for multiple purposes with a high performance like for example to do machine learning computations, or to use SQL queries over the data and always being possible to use their different API’s in conjunction if needed.

Spark is also defined as a computing engine because it is used to realize computations in a unified way being able to manage big quantities of data but it does not store the data. For that purpose, Spark is able to save the data to a lot of different storage systems: persistent, distributed or in the cloud.

Spark is written in Scala and it uses the Java Virtual Machine. For this reason, to use Spark is necessary, at least, to install Java. Spark support different languages like Scala, Java, Python, R and SQL.

4.2.2 Spark Architecture

Spark is used to coordinate the execution of tasks across clusters of computers. To carry out this task, Spark needs to use a cluster manager. In this way there are different cluster managers available like for example Spark Standalone cluster (which is a simple Spark's cluster manager), Apache Mesos or Apache Hadoop Yarn.

4.2.3 Spark Applications

A Spark application is made up of two parts, the driver and the executors.

The driver is located on a node in the cluster. It is responsible for the assignment and distribution of work to the executors and it will react to the user's program. The driver is the core of a Spark application and executes the main function.

The executors are in charge of carrying out the assigned task and sending back to the driver the result or state of the computation.

'SparkSession' is a process located in the driver and from Spark 2.0 it is a unified entry point of a Spark application which is used to interact with the park functionalities.

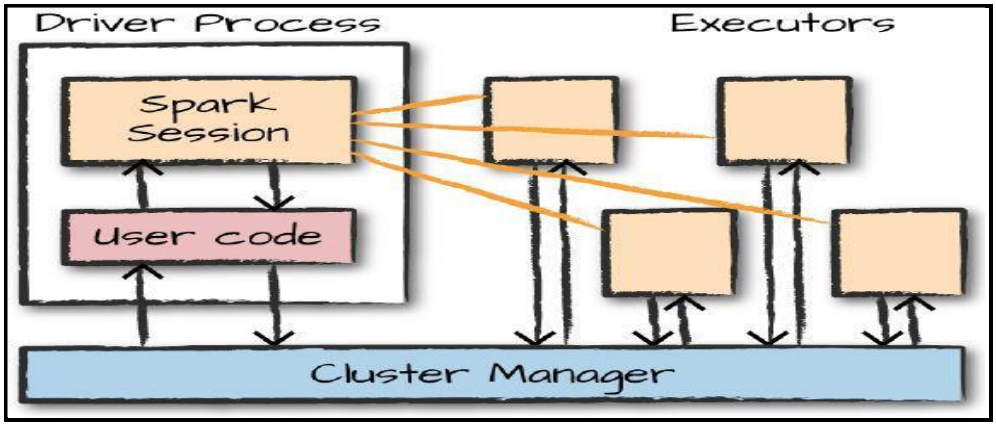


Figure 13. Spark Application

4.2.4 Spark Toolset

Spark has structured APIs, low-level APIs and a set of libraries that has been created to facilitate the use other functionalities.

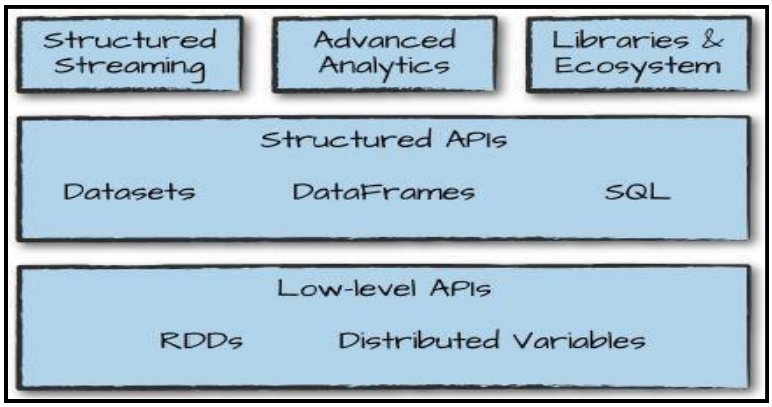


Figure 14. Spark Toolset

4.2.4.1 RDDs

The meaning of RDD is Resilient Distributed Datasets which means it is a collection of records that can be computed in a distributed way by using the low-level API. This kind of collections is immutable and will be split into several partitions across the cluster nodes so that they can be computed in parallel. At the beginning of Spark, RDD was the main API.

Although nowadays is recommended to use the Dataset API instead of RDDs, it can be useful the use of RDDs in some cases like for example if there is unstructured data, or if it is necessary to work with complex datatypes that cannot be serialized with Encoders.

4.2.4.2 DataFrames

DataFrames are an immutable collection of data but unlike the RDD is organized into columns and they are also distributed. A DataFrame is also a Dataset but organized into columns what would be the equivalent of a table in a relational database. It is considered the most used Structured API. The DataFrames has a schema consisting of a list where the columns and its types are declared.

It is useful to work with DataFrames in the cases we have structured or semi-structured data and when it is necessary the best possible performance, automatically optimized. Spark will split the data into little pieces called partitions. These partitions will be distributed across the cluster to allow the executors to work in parallel.

4.2.4.3 Spark SQL

Spark SQL is a module included in the Structured API that allows executing SQL queries and reading data from Hive. Therefore, in addition to being able to use DataFrames to describe the business logic, it is also possible to use pure SQL language without losing any performance with respect to the use of DataFrames. To make use of Spark SQL it is necessary to register a DataFrame as a temporary table and then will be ready to receive SQL queries.

4.2.4.4 Datasets

Datasets can be described as typed distributed collections of data that allows writing statically typed code.

It is not possible to use the Dataset API with some programming languages like Python and R due to those languages are dynamically typed so that the Dataset API is only available in Scala and Java.

Datasets are type-safe and they are parameterized with the type of object contained inside. This characteristic avoids the possibility of accidentally use the wrong object inside a Dataset.

That means if we have in Scala a 'Dataset[House]' it is guaranteed to contain only objects of type House. Dataset API unifies the DataFrame and RDD APIs.

The main benefits of Datasets are they provide more information than DataFrames and it provides more optimizations than RDDs.

It is useful working with Datasets in the cases we have structured or semi-structured data and when it is necessary type safety.

It is also beneficial working with Datasets when it is necessary to work with functional APIs or if it is desirable a good performance but it does not have to be the best.

4.2.4.5 Distributed Shared Variables

In some situations, like for example when using a map, filter or reduce functions these operations are being executed on a cluster and they work with a copy of all the necessary variables for the execution of that function.

The result of this is the remote machines are doing computations with their own copy of the variables and the updates of the copy are not being propagated back to the driver.

In the low-level API there are RDDs and distributed shared variables. Distributed shared variables are composed of accumulators and broadcast variables and they are useful in this situation.

Accumulators are shared variables that all the workers nodes that are doing computations on the cluster can use to aggregate values and propagate those values to the driver.

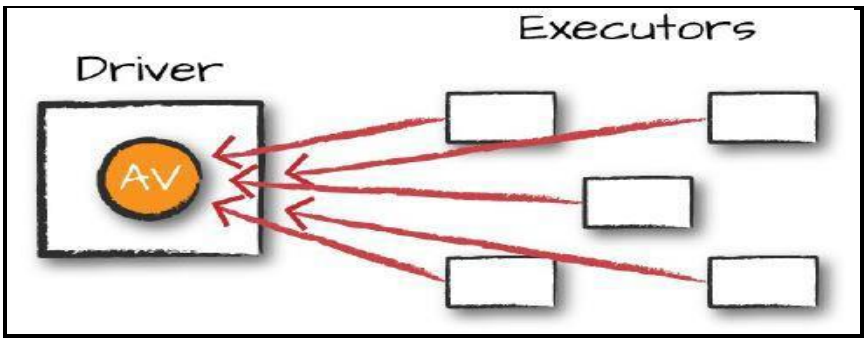


Figure 15. Accumulators

Broadcast variables are immutable (read-only) variables which are used to share its value with all the worker nodes efficiently. This kind of variable is very useful when the worker nodes need to work multiple times with the same data avoiding sending many times the same content. It is very important to note that although these variables can be very large, they must to fit in memory of the executors**. [12]**

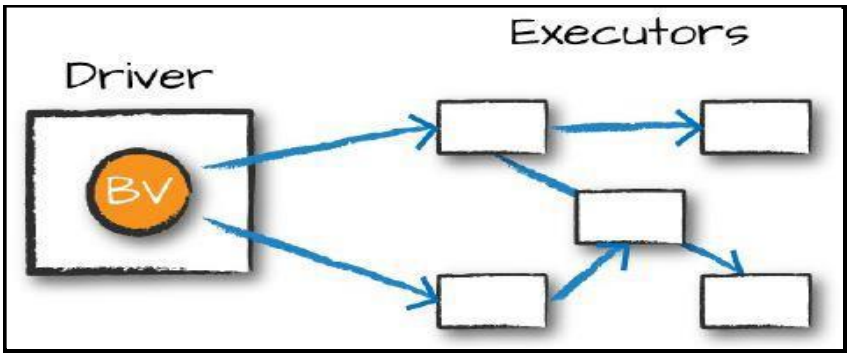


Figure 16. Broadcast Variables

4.2.4.6 Structured Streaming

Before the appearance of Spark Structured Streaming, introduced in Spark 2.0, the way to work with streaming was by using Spark Streaming and its DStream API.

However, the DStream API had many limitations, for example, that it was based on Java / Python objects as opposed to the richness that the use of DataFrames and Datasets would bring, losing the ability to obtain certain automatic optimizations or, for example, that DStreams, which are a sequence of RDDs, could only operate in a micro-batch way.

When Spark Structured Streaming arrived, it brought many advantages. One of them is the support of event-time data. Another one is that Structured Streaming can be used with Dataframes, Datasets and SQL which is something that greatly facilitates the work of the developers which can program in a similar way batch or stream processing.

Although Spark Structured Streaming is more oriented to a real steaming than Spark Streaming (DStreams) in Spark 2.0 the Structured Streaming model was still working with micro-batch. In Spark 2.3 it appeared the Continuous Processing that has a processing mode called Continuous Mode. This new mode is a way to work with real streaming and greatly decreases the latency but in Spark 2.4 is still in experimental phase.

4.2.4.7 Machine Learning and Advanced Analytics

Spark has a library called MLlib which allows working with machine learning. This library is very attractive because combine the scalability of Spark with the power of the machine learning algorithms. In this way it is possible to train models in MLlib and use it later, for example, in a streaming application by using the Structured Streaming API.

In a similar way that happened with the evolution of Spark Streaming to Spark Structured Streaming, in its beginnings the main API of MLlib was RDD-based but it started to switch towards a DataFrame-based API that is now the primary API. **[8]**

With this library it is possible to use a large number of algorithms and utilities like classification, regression, decision trees, clustering, pipelines, statistics, distributed lineal algebra, etc.

4.2.5 How Spark Structured Streaming Works

As discussed previously, with Structured Streaming the developers can code a streaming application in a similar way to code a batch application. The only necessary step is to use the correct syntax to specify that we are working with streaming. For example, to read a file like a DataFrame in Scala we have to use “spark.read” but if we need to read it as a streaming DataFrame we have to use “spark.readStream”.

This similar syntax facilitates the development. When we are working with streaming, Spark will execute the queries continuously to be able to process indefinitely the incoming data.

As we mentioned previously in the DataFrame definition, a DataFrame can be understood as a table. Therefore, a stream, when using Structured Streaming, can be also understood as an unbounded table which is continuously receiving new data.

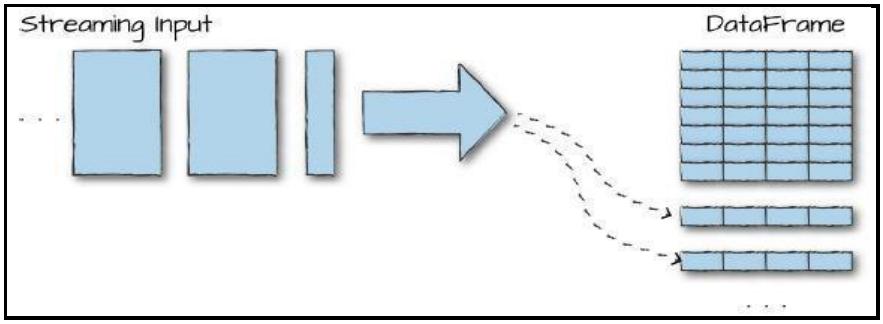


Figure 17. Structured Streaming

4.2.5.1 Input Sources

As of Spark 2.2 the supported input sources are Apache Kafka, file sources with a large number of supported formats like text, csv, json, orc or parquet, and socket sources. As of Spark 2.3 it was added the possibility to read from rate sources for testing purposes. **[9][10]**

4.2.5.2 Output Sinks

Sinks specifies the destination for the result of the processed stream. As of Spark 2.2 the list of the available sinks is: Apache Kafka, file sink, foreach sink, console sink, and memory sink**. [12]**

4.2.5.3 Output Modes

Spark not only allows the developer to define where the data will be stored (output sink) but also provides the ability to choose how the data will be written to that Sink. In this way there are three output modes:

1. **Append**: Only the new rows added to the result table since the last trigger will be sent to the output sink.
2. **Complete**: It updates the entire result table and sends it to the output sink.
3. **Update**: It only modifies in the output sink the records of the result table which were updated since the last trigger

4.2.5.4 Triggers

By setting a trigger it is possible to control when the data is sent to the output sink. Spark 2.2 has two types of trigger:

1. **Processing time trigger**: To use this trigger it is necessary to set the duration. The trigger will be fired every time the duration time is ended.
2. **Once trigger**: This trigger is useful when it is necessary to run a job only once.

4.2.5.5 Event Time and Processing Time

When working with Spark in the context of stream processing there are two well-known special concepts which are processing time and event time and it is very important to distinguish them.

1. **Event time**: It is the time at an event was generated. In Spark, event time is a column embedded in the own data. Event time helps to process the data in the order it was produced.
2. **Processing time**: It is the time at an event is processed by the system.

Spark Structured Streaming support event time. Event time is very important because provides the power to process the data in order of generation even if the data arrives disordered.

4.2.5.6 Watermarks

Watermarks are a useful method to control late data. With the use of watermarks, it is possible to establish a threshold which indicates how long the system has to keep track of late events and process them. Watermarks are a very useful feature when working with event time processing and Spark Structured Streaming supports it.

4.2.5.7 Windows

In Spark is possible to work with the windows by using processing time or event time. By default the windows works with event time, however is possible to emulate processing time by adding in the time column a call to the current_timestamp() method.

To make use of the windows it is necessary to uses them as part of a group by operation. Windows takes two mandatory parameters and a third optional one. The explanation is as follows:

1. Time Column: The name of the time column used to order the events.
2. Window Time: Used to indicate the duration of the window.
3. Slide Time: This optional parameter is used to establish the sliding time.

In Spark Structured Streaming it is possible to make use of three types of window: tumbling windows, sliding windows and session windows.

1. Tumbling window: This kind of window has a fixed size and it cannot be overlapped with other windows which mean every event will be only present in one window.

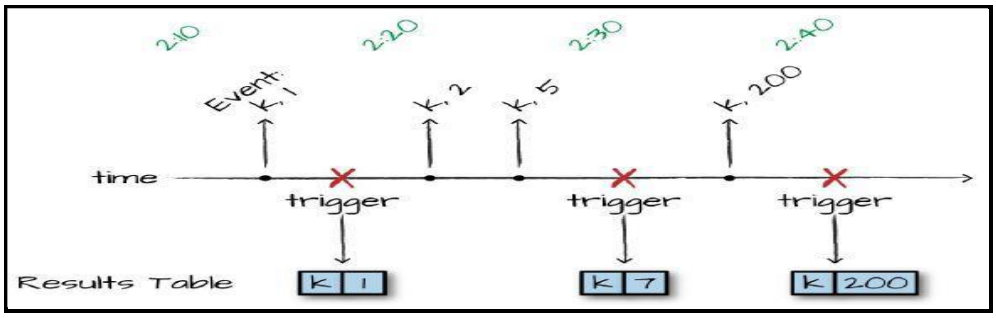


Figure 18. Tumbling Window

2. Sliding window: This kind of window has two important parts. A window size and a sliding interval. The window will process the data during the duration of the window and it will slide every given interval. A very representative use case example could be the following: “Count every 2 minutes (sliding interval) the number of cars that have been on the highway in the last 4 minutes (windows size)”. In this kind of windows there is an overlapping between windows, that means, some events can be present in more than one window (see the red point in the figure 19). **[27]**

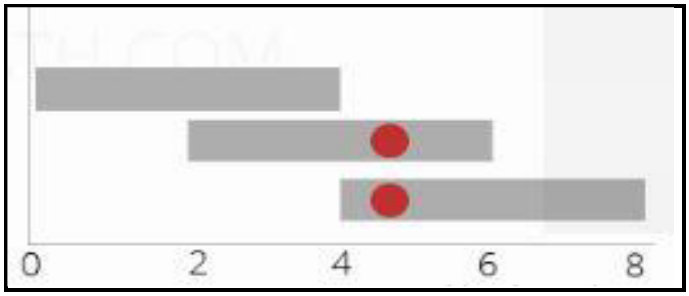


Figure 19. Sliding Window

3. Session window: Unlike sliding windows and tumbling windows, a session window provides a mechanism to group together all the events for a specific session that occur closely in time.

Although Structured Streaming only has a direct API to make use of time-based windows, it is possible to use non-time-based windows (like the session windows) by using the feature mapGroupsWithState.

4.2.5.8 Checkpoints

It is very important for a streaming application to be able to recover from a failure. To tackle with this, Spark provides a mechanism to recover an application by maintaining the same state that it had before the failure. This mechanism is known as checkpointing.

To make use of the checkpoint mechanism, it is necessary to configure a query indicating where the path to the checkpoint location is so that all the relevant progress information is saved in that place.

To mention an example, if a Spark application is reading from a Kafka source, the actual position in the offsets will be saved in the checkpoint. That means if the application is configured to start reading from the 'earliest' offset and it crashes, when the application restart it will continue reading from the exact point where it left off and not from the beginning of the Kafka offsets which is something that would send duplicate events.

4.3 MYSQL

4.3.1 What is RDBMS?

In 1970, Edgar Frank Codd published a paper titled "A relational model of data for large shared data banks". In this paper he postulated the bases of the relational databases (RDB). **[13]**

A relational database management system (RDBMS) is a software system, based on relational model, used to store data. There are a lot of database systems based on this model like for example MySQL, SQL Server, DB2, Oracle, Access… etc.

4.3.2 What is MySQL?

MySQL is a RDBMS which is available as open-source (GNU General Public License) and also under different proprietary licenses. In 1995 was its initial release and it is written in C and C++.

4.3.3 Data Organization

The data in MySQL is stored in tables. A table is compound of columns and rows and it is used to store data records. Every table has one or more fields which are the header of the columns of the table and they are used to indicate what kind of information is stored inside every column.

The columns store specific information linked to a field. The rows store the record entries in a table and they are also called tuples.

4.3.4 Data Integrity

There are different types of integrity in MySQL, just like in the other relational databases. This kind of integrities assures the following rules:

It is not possible to have duplicated rows in a table. It is not possible to entry invalid data in a column according to its type, range or format. It is not possible to delete a row if this is being used by other records. It allows the user to define some integrity rules which are different than the previous ones.

4.3.5 The Binary Log

Binary Logs are an important feature present in MySQL consisting of a sequence of events describing the database changes such as the schema changes, table creations or Insert, Update and Delete operations. But nevertheless, it is important to say the binary log is only used for statements that modify data, which means that for example a Select statement will not be recorded.

The binary log also records how long it took the system to update the data after executing a query. The binary log is mainly used for replication purposes and for recovery operations.

As far as replication is concerned, it is useful because in a system with a master and one or more slaves it allows to record the data changes occurred on a master and to send that information to the slaves which will use it to produce the same data changes.

As regards of recovery operations, it is useful because when an incremental recovery has made it is necessary to re-execute the binary log events that happened after the last backup.

The disadvantage of working with the binary log enabled is that the performance of a server will be a little slower but on the other hand it brings several advantages like for example the possibility of replication and restore operations.

Another very important advantage of using the binary log is to allow external programs to access it, as Debezium does, facilitating the process of change data capture.

4.3.6 Topology

MySQL database supports many different topologies. Below are briefly explained some of the most relevant:

4.3.6.1 Standalone

In this kind of topology there is a MySQL instance designed to run as a single node with only one master process.

4.3.6.2 Master and Slave

In this kind of topology, it can be a cluster with multiple MySQL instances. There are some different configurations available. One of them is master-slave, which works with a single master server and one or more slaves. Another one is multiple masters with multiple slaves.

4.3.6.3 Multi Master

Multi Master is an evolution of the master-slave topology. In this type of topology there are two or more master nodes.

The difference between master-slave and multi master topology is that in the multi master topology all the nodes are master and replica at the same time and there will be circular replication between them.

4.4 APACHE ZOOKEEPER

4.4.1 What is Zookeeper?

ZooKeeper is a centralized service which provides distributed configuration service, distributed synchronization service, and naming registry. It has been written in Java but it supports other programming languages.

4.4.2 What is a Partial Failure?

One of the greatest difficulties in developing distributed applications is partial failures. A partial failure occurs when a message is sent from one node to another and during the operation the network fails resulting in an uncertainty of whether the operation was completed or not.

All distributed systems are prone to partial failures but that does not mean that it cannot be controlled. Therefore, Zookeeper cannot prevent these partial failures from happening, but it provides a series of tools that help build distributed applications that can safely handle partial failures.

4.4.3 Zookeeper Characteristics

- **Simplicity**: In short, Zookeeper is a file system that provides some operations and abstractions.
- **Usability**: It provides, through the use of its building blocks, the ability to build coordination data structures and protocols.

- **High Availability**: Zookeeper runs in different machines and it will continue working if any of the nodes goes down, so it achieves a high availability and the capacity of building reliable applications.
- **Interactibility**: Zookeeper act like a coordination system that allows different processes that does not know of each other can discover and interact between them.
- **Performance**: Zookeeper has a great performance achieving a great throughput both reading and writing operations. **[19]**
- **Scalability**: Zookeeper is a distributed service and for that reason it can be scaled horizontally by adding more nodes.

4.4.4 Data Model

4.4.4.1 Znodes

"ZooKeeper maintains a hierarchical tree of nodes called znodes. A znode stores data and has an associated ACL. ZooKeeper is designed for coordination (which typically uses small data files), not high-volume data storage, so there is a limit of 1 MB on the amount of data that may be stored in any znode" **[29]**

The atomicity in the data access is guaranteed so if there is a request of data stored in a znode, the response will be to return all the requested data or nothing. The atomicity is also guaranteed in the write operations.

4.4.4.2 Namespace

Zookeeper provides a namespace that is similar to a standard file system but with some differences like for example that is possible for the ZooKeeper nodes to have data associated with it and children. In other words, this kind of file system "allows a file to also be a directory". **[77]** ZooKeeper works with paths which are represented as slash-delimited strings. A name is made up of a sequence of paths.

The figure below shows a representation of the ZooKeeper namespace structure obtained from the ZooKeeper documentation.

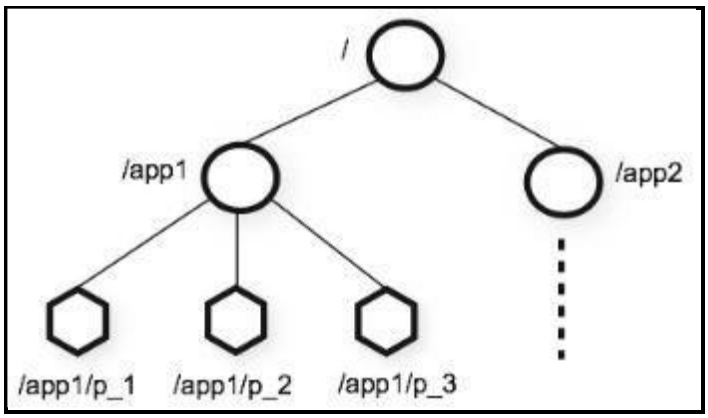


Figure 20. ZooKeeper Namespace Structure

4.4.4.3 Access Control List (ACL)

To maintain the control access to the Znodes, ZooKeeper makes use of ACLs. The ACL specify the permissions to realize operations in the nodes. The list of those permissions is: create, read, write, delete and admin.

ACLs depend on authentication and therefore the client that is going to use ZooKeeper must be authenticated. There are three ways to authenticate: The first one is digest, which allows the client to authenticate by providing a username and password. The second one is sasl which allows authenticating the client by using Kerberos. And the last way is to authenticate the client through an IP address.

4.4.5 Watches

The concept of watches is used by ZooKeeper providing the clients the ability to know when there is a change in a Znode. When a client has set a watch on a Znode and there is a change on it, it will receive a notification indicating that a change occurred in that Znode and the watch will be removed.

4.4.6 Operations

ZooKeeper provides a simple API that facilitates the programming. The next table obtained from the book "*Hadoop The Definitive Guide"* shows the list of the basic operations. **[29]**

Operation	Description
`create`	Creates a znode (the parent znode must already exist)
`delete`	Deletes a znode (the znode must not have any children)
`exists`	Tests whether a znode exists and retrieves its metadata
`getACL, setACL`	Gets/sets the ACL for a znode
`getChildren`	Gets a list of the children of a znode
`getData, setData`	Gets/sets the data associated with a znode
`sync`	Synchronizes a client's view of a znode with ZooKeeper

Figure 21. ZooKeeper Basic Operations

4.4.7 APIs

As we can read in the official documentation, Zookeeper has client libraries in two different languages: Java and C. However, there are also contributions in other programming languages

like for example Perl or Python among others, created by the community which offer unofficial APIs.

For both the Java and C bindings, there are two options which consist of working by performing operations in a synchronous way or to choose to do it in an asynchronous way.

4.4.8 ZooKeeper Modes

ZooKeeper has two modes: standalone and replicated mode.

When working in standalone mode there is only a ZooKeeper server. That means all the clients will connect to the same single ZooKeeper server. This mode is very simple and it can be used for testing purposes. The problem of this mode is that all the benefits that the replication and high availability provides are not present.

When working in replicated mode there are several servers. It will be one leader and the others will be followers. If the leader fails, one of the followers will become the leader. The way ZooKeeper works in this mode is as follows: First the choice of leader occurs. After that the leader receives write requests and it is in charge of send the updates to the followers. When a majority (called quorum) of followers has updated its state with the new change, the leader commits the update and the client will be notified about the success of the operation. Therefore, it can be easily deduced ZooKeeper will continue working properly while there are a majority of machines in the cluster (called ensemble) working without failures. For example, in an eight nodes ensemble, the system can tolerate at most three nodes failing.

4.4.9 Sessions

Clients initialize a session when they connect to ZooKeeper. When a client stablishes a connection, it provides a timeout value which indicates to the cluster when the session will expire. If client doesn't reply with a heartbeat to the server before the timeout value the server will expire the session.

If the session is idle longer than a certain period that would reach the timeout session, the client has to send a heartbeat (ping) to maintain session open. This heartbeat is managed automatically by the client library.

The heartbeat period is lower than the time-out value of the session and conservative enough to be able to control if there is server failure before the time-out value of the session expires. If ZooKeeper detects a server failure it will reconnect to another server within the session time-out period.

This mechanism is very important because if a session expires, the ephemeral nodes tied to that session will be deleted. However, if a failure has occurred on a server, only a reconnection to another server is required to continue working, in which case the session and its associated ephemeral nodes will remain valid.

4.4.10 States

The ZooKeeper connection is identified by states. When a client establishes a session, it uses a handle and starts of in ‘Connecting’ state. When the client connects with one server the state of the handle change to ‘’Connected’. In the case the application closes the handle, or the session expires or there is an authentication failure, the state of the handle will change to ‘Closed’.

The figure bellow, obtained from the book '*Hadoop The Definitive Guide*' shows the ZooKeeper state transitions.

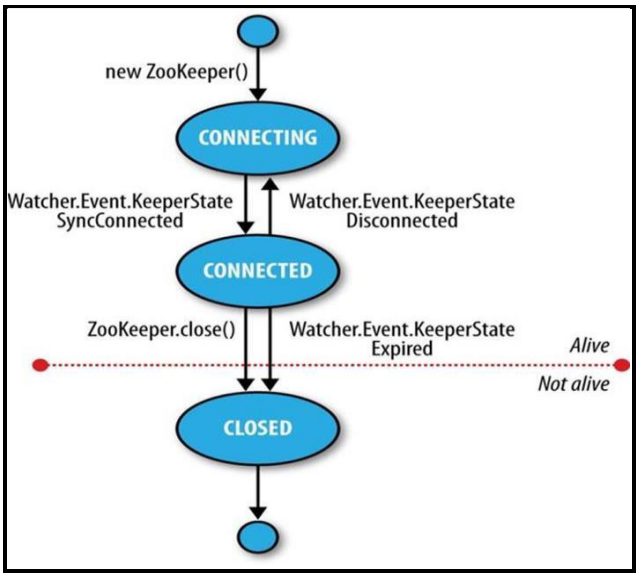


Figure 22. ZooKeeper State Transitions

4.5 APACHE PLC4X

4.5.1 What is PLC4X?

According to its official definition, “*PLC4X is a set of libraries for communicating with industrial programmable logic controllers (PLCs) using a variety of protocols but with a shared API”* **[4]**

With PLC4X is possible to read and write data to PLCs as well as subscribe or perform other operations depending on the protocol. It also provides ease of use and facilitates programming since it has a single API with which access to all the protocols it supports can be programmed.

The ease of programming and the variety of protocols it accepts, in addition to being open-source, makes it especially attractive for programming applications that need to communicate with PLCs that use different protocols.

Furthermore, the connectivity it provides with Apache Kafka makes it a very interesting option for the massive handling of data coming from PLCs.

4.5.2 Connections

This section covers the types of connections that PLC4X provides broken down into two parts: transport and protocols, and the way to connect using the connection string.

4.5.2.1 Transports

PLC4X provides the following kind of transports:

- **TCP**: Based on the normal TCP stack of the operating system
- **UDP**: Based on the normal UDP stack of the operating system
- **Raw Socket**: Allows the user to access to Ethernet frames
- **Serial Port**: Allows the user to read and write data using a serial port.
- **PCAP replay**: Designed to be able to record and replay network traffic.
- **Test**: For testing purposes related to the data managed by the drivers.

4.5.2.2 Protocols

PLC4X currently supports the following protocols:

- AB-ETH
- ADS/AMS
- BACnet/IP
- DeltaV
- DF1
- Ethernet/IP
- Firmata
- KNXnet/IP
- Modbus (TCP/Serial)
- OPC UA
- S7 (Step 7)

4.5.2.3 Connection Strings

To establish a connection, a connection string must be provided that must comply with the structure shown in the following figure:

```
{driver code}:{transport code}://{transport config}?{options}
```

Figure 23. Connection String

Where the following words corresponds to the following parts:

- **Driver code**: This part is used to stablish the protocol to be used.
- **Transport code**: This part is used to indicate the type of transport to be used.
- **Transport config**: Data needed by the transport like for example IP, hostname, port…etc
- **Options**: Can be used to change transport or protocol options to non-default values.

4.5.3 Integrations

PLC4X provides many integrations with another technologies that facilitates to process the data. The current integrations are the following:

- Apache Calcite
- Apache Camel
- Apache Edgent
- Apache IoTDB
- Apache Kafka
- Apache NiFi
- Apache StreamPipes
- Eclipse Ditto
- Elastic Logtash

Although all these are useful, the one that generates special interest since it has been used in this pipeline is Apache Kafa whose integration provides a connector to send the data extracted by PLC4X to Kafka in a very simple way, after configuring the connector.

However, there are other ways to send the data to Kafka such as using Producers. As indicated in another chapter, in this project I have chosen to manually program two producers in order to have more control over the code to be treated.

4.5.4 Tools

PLC4X provides a series of tools that are very useful depending on the needs of the programmer. The tools available are listed below.

4.5.4.1 Capture Replay

It allows to replay recorded network traffic and to directly intercept this traffic in any passive mode drivers. **[3].**

4.5.4.2 Connection Pool

It allows to make use of pooled connections. So if a connection is required, it is searched if one already exists, and if not, a new one is created.

PLCs often only accept a limited number of connections so using this method makes programming easier and eliminates the stress that would be generated on the PLC if we continually open and close the connections.

The following figure shows how to use a connection pool through the use of the class PooledPlcManager.

```
PlcDriverManager driverManager = new PooledPlcDriverManager();

// This just simulates a scenario where a lot of connections would be created and immediately destroyed again.
for(int i = 0; i < 100; i++) {
    try(PlcConnection connection = driverManager.getConnection("...")) {

        ... do something ... (please refer to the PLC4J getting started for details)

    }
}
```

Figure 24. Pooled Plc Manager

4.5.4.3 Object PLC Mapping

This tool allows mapping PLC addresses to words more understandable by the user or programmer. This facilitates the understanding of the data being used and avoids the difficulty of dealing with values of the type “I1.0” or “Q2.5” instead of values of the type “valve 1” or “motor 5”.

4.5.4.4 Scraper

This tool is very useful when working with data monitoring continuously. It manages the scheduling of queries, the connection state and the refreshing interval during the data extraction.

Specifically, it has been used in this project to facilitate the extraction of data from a large number of PLCs using its trigger to establish how often the data needs to be read from the PLCs.

4.6 DOCKER

4.6.1 What is Docker?

Docker is a tool that allows to deploy applications in containers, fast and in a portable way. This containerization provides a powerful capability to package up an application, with all the needed components as one package.

4.6.2 Architecture

Docker architecture is made up of different components such as Docker host, Docker client, Docker registry and Docker objects.

The figure below shows a graphical overview of the Docker's architecture.

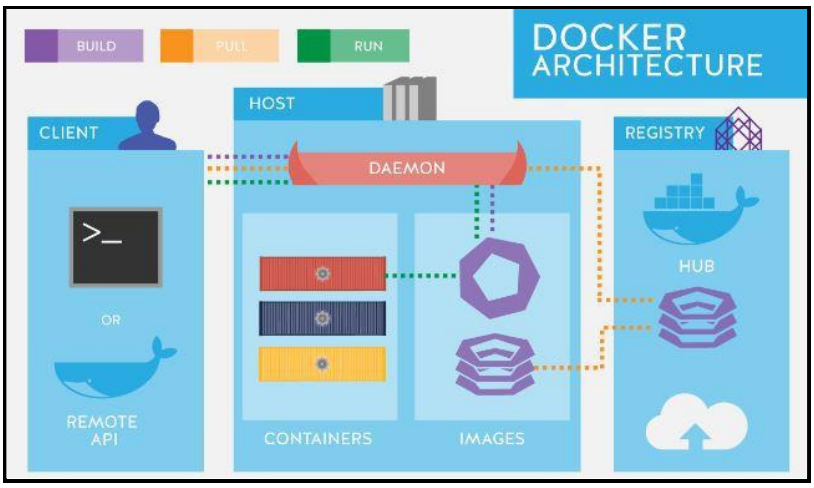


Figure 25. Docker Architecture

4.6.2.1 Docker host

It is the machine where Docker resides. It can be a local computer or a server in the cloud. Docker host consists of the following parts: Docker Server (Daemon), RESTful API and Docker Client.

It is necessary to clarify that the Docker client and the daemon can reside on the same system or not. In the case they don't reside in the same system is possible to connect a Docker client to a remote Docker daemon.

The following figure shows a representation of the Docker Host which includes, in this case, the Docker Client inside.

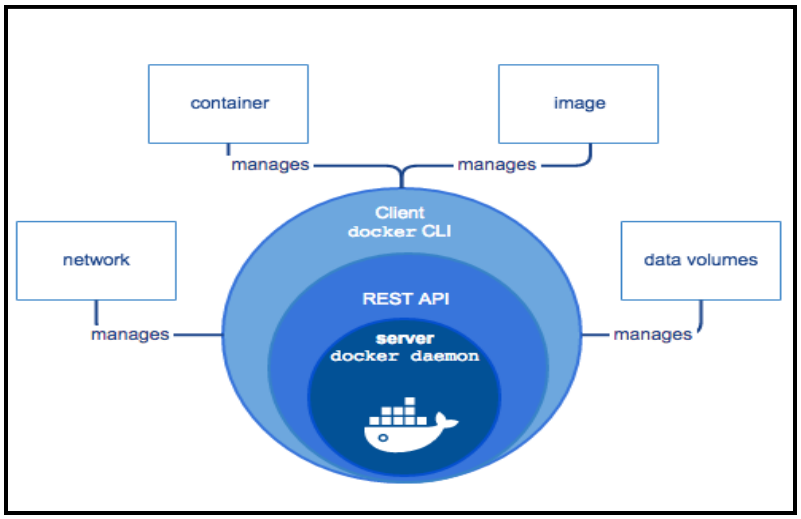


Figure 26. Docker Host

1. Docker Client: It is used to interact through the use of commands with the Docker server. This action is performed by using the Command Line Interface. The commands do not go directly to the Docker server but they go firstly to the RESTful API which will send to the Docker server the request to perform the pertinent tasks. It can connect to a daemon in the same Docker Host or to a remote daemon and it has the capability to communicate with multiple daemons.

2. Docker Server (Daemon): It is a server that interacts with the operating system and it takes care of creating and handling containers. It receives requests from the REST API and to execute the required actions. It is possible to run multiple daemons in the same host although the reference manual indicates this is an experimental feature and it is still under development [34]

3. RESTful API: It is the responsible to receive the commands send by the Docker client and send a request to the Docker server to perform the requested actions.

4.6.2.2 Docker Client

As explained before, it provides the user the capability to interact with Docker and it can be hosted in the same host as the daemon or not. For further information, see the Docker Host section.

4.6.2.3 Docker Registry

This is a service which provides repositories to the user to store and download Docker images. The user can create their own images and store them there for later use or make use of the images that other users created and stored it in a public registry like Docker Hub or Docker Cloud. By default, Docker searches the required images in Docker Hub.

4.6.3 Docker Objects

To be able to assemble an application Docker needs to made use of different entities called Docker objects. The following lines describes the main Docker objects.

4.6.3.1 Images

It is important to know the concept of Dockerfile in order to understand what an image is. A Dockerfile is a file which is used to build an image. It is a list of instructions that tells Docker how to build the image.

In simple words, images are read-only templates that are necessary to build containers. For a more detailed description we find the definition provided by James Turnbull in the book '*The Docker Book*' that defines it as: "At the base is a boot filesystem, *bootfs*, which resembles the typical Linux/Unix boot filesystem. […] Indeed, when a container has booted, it is moved into memory, and the boot filesystem is unmounted to free up the RAM used by the *initrd* disk image." **[26]**

Images have instructions inside to create a container. An image can be based on another image to add extra configuration details or services.

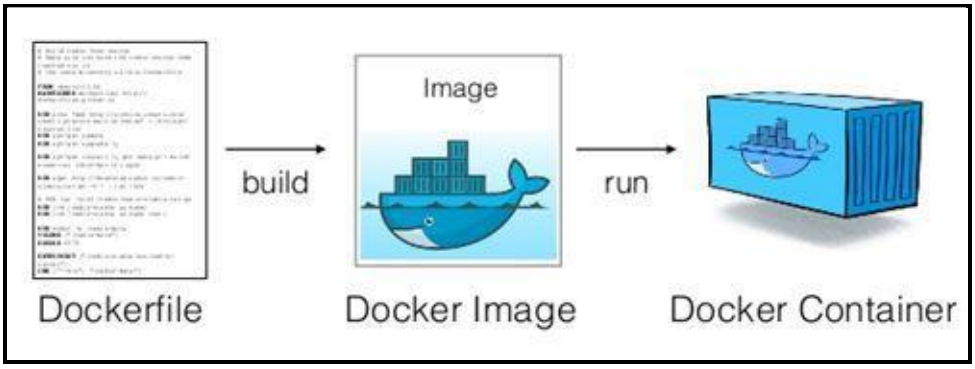


Figure 27. Docker Image and Container

The user has the possibility to create their own images or take advantage of the images that other users have previously created and uploaded to a public repository to use them directly without having to create them again.

4.6.3.2 Containers

"A container is a runnable instance of an image". **[16]** The user can perform, by using the Docker CLI, multiple operations related to the container like for example: create, delete, start, stop... etc.

Containers are isolated execution environments that have everything necessary to operate autonomously. Although the containers are isolated environments, they can communicate with other containers or with the outside world using the networks. They also can save the generated information by using volumes.

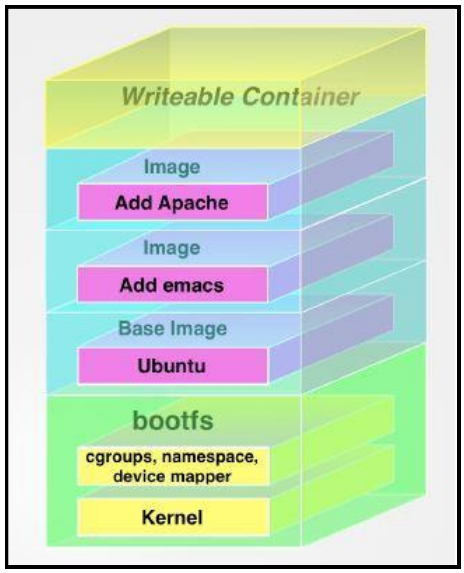


Figure 28. Container Architecture

The figure above, obtained from the book '*The Docker Book*' **[26],** shows multiple images stacked on top of each other. These images are of the type read-only and "when a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below. [...] When a container is created, Docker builds from the stack of images and then adds the read-write layer on top. That layer, combined with the knowledge of the image layers below it and some configuration data, form the container". **[26]**

4.6.3.3 Networking

The Docker network infrastructure works by using different drivers which provides the main networking functionality. Docker has the following default drivers:

1. Bridge: This driver is used by default. It creates an isolation that allows only the containers from the own network can communicate between them. This driver can be only used for communicate in the same Docker daemon host.

2. Host: This driver eliminates the isolation allowing the containers on the same Docker host can communicate each other directly.

3. Overlay: This driver allows communication between containers residing in different hosts. It can connect multiple Docker daemons between them.

4. Macvlan: It eliminates the bridge between the containers and the host by using MAC addresses. It is useful when an application has the need to be directly connected to the network.

5. None: It is used when it is necessary to completely disable the networking capability on a container.

4.6.3.4 Storage

Docker allows to persist container data. When a container is running it is managing data and if the data is not persisted, when the container will be removed, all their data will be lost. Docker provides different ways to persist the data.

1. Bind mounts: It consists in binding a local host directory inside the container. They are considered limited in functionality compared to volumes. To use it is necessary to provide an absolute path.

2. Volumes: They are managed by Docker and they are created in the host. A volume can be mounted into multiple containers. There are two types of volumes: named, if it is provided with an explicit name, or anonymous, if it is not provided with an explicit name, case in which Docker will assign to it a random unique name.

3. tmpfs mounts: This kind of mount does not persist data on disk. It persists data only in the host memory. That means when the container is stopped, all the data will be lost.

4.6.4 Docker Compose

Docker Compose is a tool that facilitates the job when working with multiple containers. It allows realizing operations in all the containers at once, like for example, start them or stop them with only one command. The main parts of Docker are described below.

4.6.4.1 YAML File

To work with Docker Compose is necessary to write a YAML file describing all the configurations of every service. The configurations allowed by this YAML file are very extensive and provides a rich grammar to configure things like services, networks, volumes, environment variables, the use of CPU and RAM … etc.

Once the YAML file is created, the necessary command to use it to initialize the services included inside it is:

docker-compose up

At the beginning of the YAML file it is necessary to specify the version of the Compose format and from there, start describing the services one by one. As follows you can find a description of the core parts of a YAML file.

4.6.4.2 Services

The *services* word is used to establish a section for the containers' configuration. In this section is where is necessary to name every service and describe its configuration. The figure below shows an example where is possible to appreciate the file starts with the *version* word followed by the *services* word.

```
version: '2'
services:
  zoo1:
    image: zookeeper
    restart: always
    container_name: myzookeeper1
    hostname: zoo1
    ports:
      - 2181:2181
    environment:
      ZOO_MY_ID: 1
      ZOO_SERVERS: server.1=0.0.0.0:2888:3888;2181
      ZOO_INIT_LIMIT: 20
    volumes:
          - "zookeeperData1:/data"
          - "zookeeperDataLogs1:/datalog"
          - "zookeeperLogs1:/logs"
    networks:
      control_net:
        ipv4_address: 10.0.1.12
  zoo2:
    image: zookeeper
```

Figure 29. Services in Docker Compose

In this example, zoo1 corresponds to the name of the service, which in this case is a zookeeper service. The other settings will be explained in the following sections.

4.6.4.3 Volumes

To establish the volumes in the YAML file we can write directly the host volumes inside the service (under the volumes word). For the named volumes it is necessary to configure it in the outer level of the configuration. In this way the visibility of this volume is global (all the containers can see it) and then specify inside the service (under the volumes word) the volume to use.

The way to configure a volume inside a service is as follows:

"nameOfTheVolume:/cotainerPath"

In this example the 'nameOfTheVolume' corresponds to the volume name given in the outer level of configuration and the '/containerPath' corresponds to the containers' path that will be persisted.

The figure below shows three named volumes, zooKeeperData1, zooKeeperDataLogs1 and zooKeeperLogs1, defined in the outer level of configuration like explained before and used for the ZooKeeper service, in the volumes section of the ZooKeeper service, consisting in three text strings of the form: "zookeeperData1:/data", "zookeeperDataLogs1:/datalog" and "zookeeperLogs1:/logs"

```
services:
    zool:
        image: zookeeper
        restart: always
        container_name: myzookeeper1
        hostname: zool
        ports:
          - 2181:2181
        environment:
          ZOO_MY_ID: 1
          ZOO_SERVERS: server.1=0.0.0.0:2888:3888;
          #ZOO_INIT_LIMIT: 20
        volumes:
              - "zookeeperData1:/data"
              - "zookeeperDataLogs1:/datalog"
              - "zookeeperLogs1:/logs"
        networks:
          control_net:
            ipv4_address: 10.0.1.12
    volumes:
        zookeeperData1:
        zookeeperDataLogs1:
        zookeeperLogs1:
```

Figure 30. Volumes in Docker Compose

4.6.4.4 Networking

This is an essential part that must be carefully defined as it configures the communication between containers. The services can communicate between the in the same network by indicating the container name (or IP) and the port. It is necessary to expose the ports so that the containers can be reached. To expose ports, it is necessary to use the *ports* keyword.

The figure below shows an example that defines two networks. Here, in a similar way of the volumes, starts defining in the outer level of configuration, two networks named control net and processing net, which are situated at the end of the file. This allows using these networks in multiple containers if necessary.

Inside the 'connect' service, there are two keywords: ports and networks. The first one is used to expose the ports of the container and the second one is used to indicate to which networks the container belongs and assign an IP address.

In this concrete (and simplified) example the networks have been created with the bridge driver, which means the containers that do not belong to these networks cannot communicate with these containers.

```
kafka1:
    image: debezium/kafka:0.8
    container_name: mykafka1
    stop_grace_period: 10s
    ports:
     - 9092:9092
    depends_on:
     - zool
    volumes:
          - "kafkaData:/kafka/data"
          - "kafkaLogs:/kafka/logs"
          - "kafkaConfig:/kafka/config"
    networks:
      control_net:
        ipv4_address: 10.0.1.3
      processing_net:
        ipv4_address: 10.0.0.3
networks:
  control_net:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 10.0.1.0/24
          gateway: 10.0.1.1
  processing_net:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 10.0.0.0/24
          gateway: 10.0.0.1
```

Figure 31. Networking in Docker Compose

4.6.4.5 Deployment Order

It is usual that some services depend on others and in that case the order in which they are started is important in order to work properly. For this purpose, the keyword *depends_on* is used.

This is usually a problematic point because although it is possible to choose the starting order, Docker does not wait for the service to load, just wait for it to start. It is necessary to take this into account and solve it by applying other additional configurations if necessary.

4.6.4.6 Restart Services

The *restart* keyword can be used to manage the behaviour of the containers in case of stop. It can be configured in four ways.

- restart: “no” --> The container will never be restarted.
- restart: “always” --> The container will always restart.
- restart: “on-failure” --> The container will be restarted if it stopped due to a failure.
- restart: “unless-stopped” --> The container will always be restarted except if it is explicitly stopped or Docker is restarted.

4.6.4.7 Images

The *image* keyword allows selecting the image to create the container. There are two ways to use it. If the image is already built, it is only necessary to use the keyword *image* to indicate which image to use.

```
kafka1:
  image: debezium/kafka:0.8
  container_name: mykafka1
```

Figure 32. Use of Existing Image

But if it is necessary to build the image, the keyword *build* comes into play. It can be used to indicate a path to the Dockerfile or a URL.

```
kafka1:
  build: /pathToDockerfile
  container_name: mykafka1
```

```
kafka1:
  build: https://url
  container_name: mykafka1
```

Figure 33. Different Image Creation Modes

To give an image name when it is built it is necessary to use the keywords *image* and *build* in conjunction.

```
kafka1:
  build: /pathToDockerfile
  image: my-kafka-project
  container_name: mykafka1
```

Figure 34. Adding Image Name

4.6.4.8 Environment Variables

The keyword *environment* allows defining the necessary environments variables.

```
kafka2:
  image: debezium/kafka:0.8
  environment:
    - HOST_NAME=10.0.0.7
    - ADVERTISED_HOST_NAME=10.0.0.7
```

Figure 35. Environment Variables

5. PROJECT DESIGN

5.1 REQUIREMENTS ANALYSIS

Before starting the project, an analysis of the requirements to be met was performed, obtaining the following:

5.1.1 Functional Requirements

This section defines the list of software functional requirements.

- **SRS-001 Data Extraction**
  The pipeline must to be able to extract data from more than 30 PLCs simultaneously.

- **SRS-002 Data Flow**
  The pipeline must be able to store the extracted data in an event streaming platform.

- **SRS-003 Data Storage**
  The pipeline must be able to store the data in a database.

- **SRS-004 Data Monitoring**
  The project must provide a way to visualize the data flowing through the pipeline.

- **SRS-005 System Start-up and Shutdown**
  All system components (at the node level), must be able to be turned on and off individually.

5.1.2 Non-Functional Requirements

The requirements that define the criteria that judge the operation of a system are defined below.

- **SRS-006 Cost**
  The system must consist mainly of free or open-source technologies. In the case of non-free technologies, they should have a low cost

- **SRS-007 Performance**

  The extraction of data from the PLCs and its storage in an event streaming platform must be done in less than a second. The data storage from when it is extracted from the PLCs until it is stored in the final database must be done in less than 4 seconds.

- **SRS-008 Availability**

  Data availability must be guaranteed on two points. In the event streaming platform (central part of the pipeline) and in the final database (final part of the pipeline), in such a way that while these two technologies are in operation, there is always the possibility to access to the data they contain and that in the event of a sudden stoppage of any part of the pipeline, the data will be available again as soon as possible.

- **SRS-009 Portability**

  The system must be capable of fast deployment in physical or cloud systems.

- **SRS-010 Scalability**

  The system must be able to scale horizontally.

- **SRS-011 Fault Tolerance**

  The system must be stable and in the event of a sudden stop of any part of the pipeline, the minimum possible damage to the other parts of the pipeline must be guaranteed. In addition, the system must be distributed with multiple nodes whenever possible, to increase fault tolerance.

5.2 ARCHITECTURE DESIGN

For the development of the architecture, all the requirements collected have been considered and finally, as explained above, the architecture shown in the following image has been chosen.

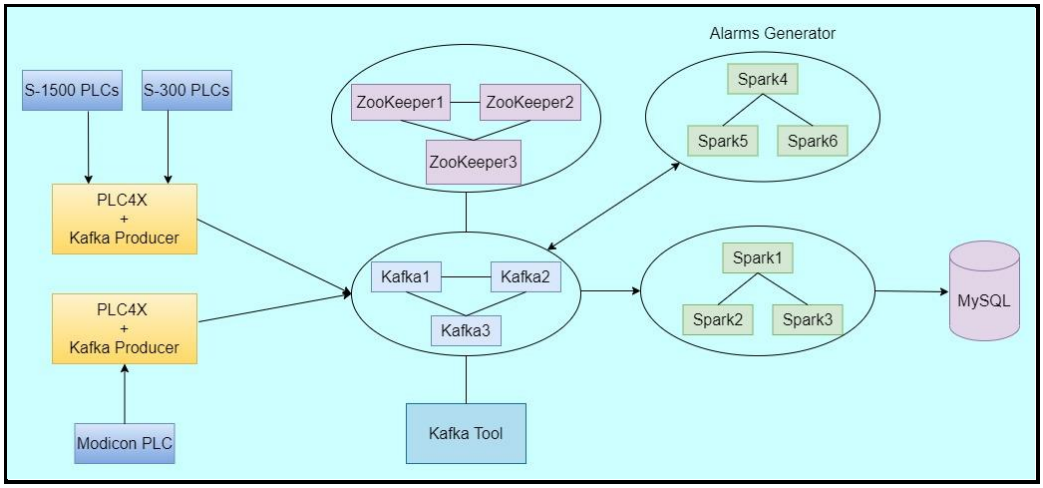


Figure 36. Architecture Design

This architecture is based on the following main ideas which are: decoupling of producers from consumers, modularity, containerization, scalability, high performance and fault tolerance. The following explains how each of them has been achieved in detail.

5.2.1 Decoupling

The decoupling of the systems and especially the decoupling of the producers from the consumers is of vital importance in this pipeline.

It has been tried that each developed application or implanted technology depended as little as possible on the rest in order to achieve the decoupling of the systems. In this way, it has been achieved that if a consumer fails, it does not make the rest of the pipeline stop. In the same way, if a producer stops, the rest of the pipeline continues working.

Therefore, decoupling producers from consumers is an essential part of this project.

5.2.2 Modularity

Modularity has been considered from the beginning of the project as something necessary since it mainly facilitates the implementation of two of the requirements such as: SRS-005 (System Star-up and Shutdown) and SRS-009 (Portability).

In this way, by containerizing all the technologies and their nodes, and all the applications and their nodes, we achieve the desired modularity. This modularity enables compliance with the requirement SRS-005 because it provides the ability to turn on and off each technology or application and even each node individually.

It also facilitates the implementation of the SRS-009 requirement because it provides the ability to easily and quickly deploy both in physical or cloud systems.

5.2.3 Containerization

Containerization, as explained above, has helped to achieve the necessary modularity to meet various requirements.

To carry out containerization, Docker technology has been used since, in addition to being open-source, it has demonstrated its efficiency for many years, being widely used worldwide and having been integrated into the most important cloud systems.

The use of Docker gives us the ability to control each and every one of the aspects related to architecture (such as networks), hardware (allowing to limit the use of CPU for each container) or software (allowing to set environment variables) to mention a few examples.

5.2.4 Scalability

The SRS-010 requirement states that the system must be able to scale horizontally. To achieve horizontal scalability, it has been decided to use typical Big Data technologies such as Apache Kafka, Apache Spark and Apache ZooKeeper because they are technologies with simple and fast horizontal scalability.

In addition, as these technologies are open-source, they facilitate compliance with the SRS-006 (Cost) requirement.

These technologies are widely used worldwide and have shown very high performance, so together with the fact that they are technologies that can work in a distributed way, they are perfect for this pipeline.

5.2.5 High Performance and Fault Tolerance

As explained in the previous section, the Big Data technologies used provide great performance. But they not only provide that but also have mechanisms to achieve a great fault tolerance and avoid many possible failures to the system.

Kafka is a very robust technology that allows that although some of its brokers have stopped due to a problem or failure, the other brokers are able to continue operating and maintaining the availability of the data.

Spark is a technology that allows us to recover from failures in such a way that if the application stops working, we have a checkpoint mechanism that will allow us to resume reading Kafka data right where it left off.

Although it is a system that works in a master-slave mode and it can be considered that its master is a single point of failure, that is, if the master fails, it will stop the entire system from working, Spark provides ways to solve this problem for example by using a multi-master configuration so that if one master fails, another master can boot up allowing the system to function normally.

ZooKeeper is used for the coordination of Kafka nodes and is also necessary if you want to use a multi-master configuration in Spark. ZooKeeper is a technology that achieves a great fault tolerance since in the same way that Kafka replicates its data to the rest of the nodes and as long as the quorum is maintained.

The quorum represents the minimum number of servers that have to be running for ZooKeeper to work correctly and is calculated using the following formula: (N + 1) / 2 being N the total number of servers (should be an odd number).

MySQL is a database that can be implemented with different topologies. In this project, the topology called standalone, where there is a MySQL instance designed to run as a single node with only one master process, has been implemented.

This is enough to be able to store all the incoming data stream quickly and efficiently. However, if it were necessary to add greater security and avoid the single point of failure, the topology would have to be changed to achieve a high availability configuration.

PLC4X technology has a very good performance but has the weakness of not being a distributed system, so we can consider it another single point of failure. The solution that has been provided in this project to try to reduce this problem as much as possible has been to create several producers, one for each protocol used, in such a way that if any producer fails, the others can continue working normally. This solution provides us with the avoidance of this problem as much as possible and a higher performance.

There could be other solutions that would take more work, such as running several PLC4X applications at the same time extracting the same data and sending it to Kafka. At Kafka the data would have to be deduplicated. Data deduplication is common in this type of systems but the approach used in this project has been different for time reasons.

PLC4X is a very new technology, so in the future it would be interesting to investigate how it can be converted into a distributed system by using technologies such as ZooKeeper or similar. This would eliminate the single point of failure and could even serve to balance the load between different nodes to always maintain a good performance. But this is only discussed as a possible future enhancement.

5.3 APPLICATIONS DESIGN

For the development of the software necessary for the correct operation of the pipeline, four applications have been programmed.

Two of them, the producers, have been programmed using PLC4X technology together with a Kafka Producer and the other two, the producers, have been programmed using Spark technology.

5.3.1 Producers (PLC4X Applications)

First of all, it is necessary to indicate that when we talk about producers and consumers we do it from Kafka's point of view, so an application that sends data to Kafka will be considered a producer and an application that reads Kafka’s data will be considered a consumer.

In this pipeline, two producer applications have been created in order to extract the data from the PLCs and send it to Kafka.

It should be noted that one of the two Spark applications that have been considered consumers (Alarm Generator) also performs a role of producer because it not only consumes data from Kafka but also produces data and sends it to Kafka, but for reasons of simplicity it has been decided to explain it in the section of consumers and explicitly indicate this particular condition.

The two producer applications have been programmed using PLC4X to extract the data from the PLCs and with a Kafka Producer to send the data to Kafka.

These applications have been developed with the Java programming language and under a Maven-type project. In order to make use of PLC4X and Kafka Producer, the necessary dependencies have been added to the pom.xml file, especially highlighting the drivers related to the protocol to be used with the PLCs, the one related to the API, the one related to the scraper tool, the one related to the connection pool, and the one related to the use of Kafka.

```
<dependency>
    <groupId>org.apache.plc4x</groupId>
    <artifactId>plc4j-api</artifactId>
    <version>0.7.0</version>
</dependency>
<dependency>
    <groupId>org.apache.plc4x</groupId>
    <artifactId>plc4j-driver-s7</artifactId>
    <version>0.7.0</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.apache.plc4x</groupId>
    <artifactId>plc4j-driver-modbus</artifactId>
    <version>0.7.0</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.apache.plc4x</groupId>
    <artifactId>plc4j-scraper</artifactId>
    <version>0.7.0</version>
</dependency>
<dependency>
    <groupId>org.apache.plc4x</groupId>
    <artifactId>plc4j-connection-pool</artifactId>
    <version>0.7.0</version>
 </dependency>
 <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.0.0</version>
</dependency>
```

Figure 37. Dependencies

5.3.1.1 PLC4X

In the part of PLC4X, all the necessary steps have been carried out to configure its operation, making use of pooled connections that facilitates the use of multiple connections and the scraper tool that gives us the ability to extract data from multiple PLCs periodically by programming a trigger.

The programming related to data extraction has been done asynchronously to avoid blocking the program and to work faster and more efficiently.

Once the data from the PLCs is received, it is parsed and transformed to be able to add important data such as the timestamp and proceed to the creation of its scheme. As explained in chapter 4.1.3.3 (Schemas), it is very important to have a schema. This schema has been built with the JSON format which is a widely used format. This parsing is a minimum parsing to be able to have the data in an almost original state so that any consumer can access it through Kafka and modify it as needed.

From the first moment, the possibility of performing the final parsing in this application has been ruled out, that is, the idea of creating in this application the queries that will be inserted in the final database has been ruled out because this would cause great problems to the system such as the loss of performance due to work overload and, more importantly, the loss of access to the raw data that consumers might need at Kafka.

5.3.1.2 Kafka Producer

Regarding the development of the Kafka Producer, all the necessary configurations have been carried out, indicating among other things, the IP address and the port of the 3 Kafka nodes.

In terms of safety, an idempotent producer has been created. This means that if a network error occurs during the sending of a message and the producer does not receive Kafka's acknowledgment, he will resend the message.

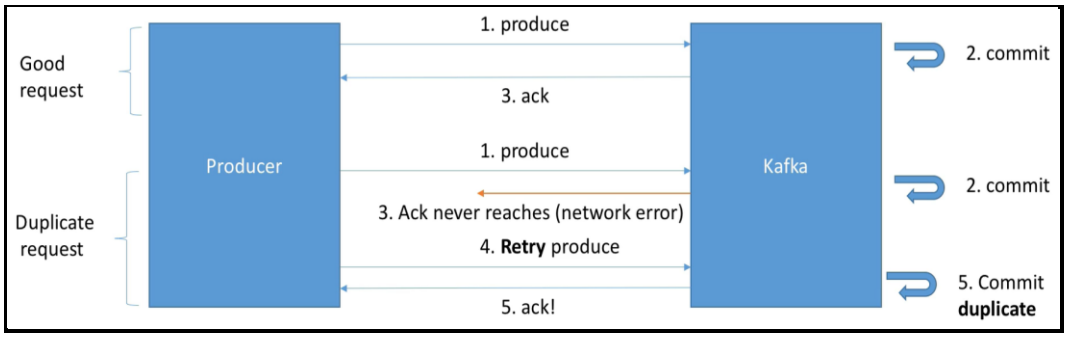


Figure 38. Kafka Duplicated Commit

But with the use of the idempotent producer, Kafka will be able to recognize this situation and avoid duplication of messages that could occur with this type of network failure. So, in this situation Kafka will send the ack but it will not commit twice.

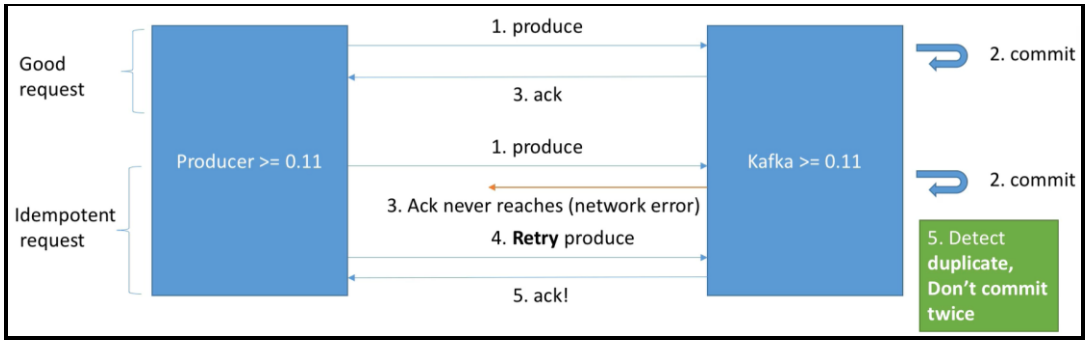


Figure 39. Kafka not Duplicated Commit

The idempotent producer includes the tuning of the following parameters:

1. **ENABLE_IDEMPOTENCE**: To enable the idempotent producer.

2. **ACKS**: The number of acknowledgments the producer requires the leader to have received before considering a request complete. “acks=all” means the leader will wait for the full set of in-sync replicas to acknowledge the record. This guarantees that the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest available guarantee. **[1]**
3. **RETRIES**: Setting a value greater than zero will cause the client to resend any record whose send fails with a potentially transient error. **[1]**
4. **MAX_IN_FLIGHTS_REQUESTS_PER_CONNECTION**: This controls how many messages the producer will send to the server without receiving responses. **[24]**. For Kafka versions equal to or greater than 1.0 (which is the case of the developed pipeline) it can guarantee the order for a maximum number of 5 requests.

Below is the code used to create the idempotent and safe producer.

```
// Create Safe Producer
    properties.setProperty(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,"true");
    properties.setProperty(ProducerConfig.ACKS_CONFIG,"all");
    properties.setProperty(ProducerConfig.RETRIES_CONFIG, Integer.toString(Integer.MAX_VALUE));
    properties.setProperty(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION,"5");
```

Figure 40. Idempotent Producer

In terms of performance, a special configuration has been created to ensure high throughput. This configuration includes the tuning of 3 very important parameters which are the following:

1. **COMPRESSION_TYPE_CONFIG**: Used to indicate the type of compression that you want to use (if used).
2. **LINGER_MS_CONFIG**: Used to indicate the time to wait before sending a batch to Kafka.
3. **BATCH_SIZE_CONFIG**: It is used to indicate the minimum size of the batch before it can be sent.

In these applications it has been decided to apply the snappy compression method as it provides great performance. Although compression adds a bit of latency and CPU usage, these consequences are minimal. In addition, a 10 millisecond linger setting has been chosen and a 32 KB batch setting has been used. This configuration provides benefits such as avoiding network saturation and greatly improving throughput.

```
// High Throughput Producer (at the expense of a bit of latency and CPU usage)
    properties.setProperty(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy"); // snappy compression type
    properties.setProperty(ProducerConfig.LINGER_MS_CONFIG, "10"); // ms waiting before sending a batch out
    properties.setProperty(ProducerConfig.BATCH_SIZE_CONFIG, Integer.toString(32*1024)); // 32 KB batch size
```

Figure 41. High Throughput Producer

Once the two applications have been programmed with the indicated parameters, it has been found that they have good performance even in the case of the application corresponding to the reading of the s7 PLCs that has to read the data of 38 PLCs at the same time.

In addition, although it is outside the scope of the project, the two applications have also been merged into one, just for testing, noting that the performance was still very good, which seems to indicate that the two applications have not reached their limit in the handling of the data of the PLCs.

5.3.2 Consumers (Spark Applications)

This section explains how the two Spark applications have been programmed. These two applications are called Alarm Generator and Spark Transformer.

The role of the Alarm Generator is to read the data contained in the Kafka topic called s7_readings and if any of the inputs from I1.0 to I1.7 has been activated, generate an alarm indicating all the faults (each of these inputs will be considered as a damaged valve failure). This generated alarm will be sent to the Kafka topic called plcAlarms.

As explained, this application not only reads the data from Kafka (consumer) but also sends data to Kafka, so it also acts as a producer.

The role of Spark Transformer is to read the data contained in all Kafka topics (s7_readings, modbus_readings and plcAlarms), parse them and generate the necessary queries to be able to send this data to the MySQL database.

5.3.2.1 Data Reading

The two Spark applications read the data from Kafka topics. In order to perform this reading, the following steps must be followed (Scala code):

1. Create a Spark Session

```
20      val spark = SparkSession
21        .builder()
22        .appName("SparkTransformer")
23        .getOrCreate()
```

Figure 42. Spark Session

2. Indicate the source of the data reading

```
27    val dsl = spark.readStream.format("kafka")
28      .option("kafka.bootstrap.servers", "10.0.0.3:9092,10.0.0.7:9093,10.0.0.8:9094")
29      .option("subscribe", "modbusPLC0,s7PLC0,s7PLC1,s7PLC2,s7PLC3,s7PLC4,s7PLC5,s7PLC6,s7PLC7," +
30        "s7PLC8,s7PLC9,s7PLC10,s7PLC11,s7PLC12,s7PLC13,s7PLC14,s7PLC15,s7PLC16,s7PLC17,s7PLC18," +
31        "s7PLC19,s7PLC20,s7PLC21,s7PLC22,s7PLC23,s7PLC0,s7PLC24,s7PLC25,s7PLC26,s7PLC27,s7PLC28," +
32        "s7PLC29,s7PLC30,s7PLC31,s7PLC32,s7PLC33,s7PLC34,s7PLC35,s7PLC36,s7PLC37,plcAlarms")
33      .option("startingOffsets", "earliest")
34      .option("failOnDataLoss" , "false")
35      .load().withColumn("value", regexp_replace(col("value").cast("string"), "\\\\", ""))
36      .withColumn("value", regexp_replace(col("value"), "^\"|\"$", ""))
```

Figure 43. Stream Reading

In line 27 the *readStream.fromat("Kafka")* indicates the source from which we want to read the stream, in this case, Kafka. Line 28 provides the list of Kafka brokers. Lines 29,30,31 and 32 set the topic or topics to read from. Line 33 is set to stablish the reading behaviour, establishing in this case to start reading from the earliest offsets. This option will allow reading the events that happen even when the Spark application was stopped. Spark knows exactly where it must read because of the Spark checkpoint mechanism that tell Spark where it stopped the last time. Line 34 sets whether to fail the query when it's possible that data is lost. The last two lines do a parsing to avoid characters added by escaping special characters.

3. Infer the schema

To carry out the next step, it is necessary to know the scheme of the data to be read and for that purpose a portion of the data that is to be read has been captured and it has been located in a file called *batchNameParsed.txt*. After this we proceed to read the file and extract its schema as shown in the following figure.

```
43    // Infer the schema from a file
44    val fileStream = scala.io.Source.fromInputStream(getClass.getResourceAsStream("/batchNameParsed.txt"))
45    val contents =  fileStream.getLines().mkString("\n")
46
47    import spark.implicits._
48
49    val schemaInferred = spark.read.json(Seq(contents).toDS).schema
```

Figure 44. Inferring Schema

4. Select the JSON data

The data read from Kafka will arrive in JSON format and the next step is to take only the part of the JSON that contains the information we need. For this, the code shown in the figure below is used.

```
55    val eventWithSchema = dsl.selectExpr("CAST(value as STRING) as json")
56      .select(from_json($"json", schema = schemaInferred).as("data")).select("data.*")
```

Figure 45. JSON Data Selection

5. Extract the fields

Once we have the part of the JSON that contains the information we need, we proceed to extract the fields on which we are going to work. The following figure shows the extracted fields.

```
61  val extracted = eventWithSchema.select("address","model","sourceName","timestamp","values.Q0",
62    "values.Q1","values.Q2","values.Q3","values.I0","values.I1","values.I2","values.I3",
63    "values.coil1","values.holdingRegister1","alarm_message","alarm_timestamp","alarm_address")
```

Figure 46. Extracting Fields

And from this point, we are ready to use those fields as we need.

5.3.2.2 Data Parsing

In the case of the Spark Transformer application, the parsing is carried out within the class called InsertByRow that is in charge of determining if the data read corresponds to the topic s7_readings, modbus_readings or plcAlarms and based on this it parses the data and generates the appropriate query to be able to save the data in the MySQL database.

In the case of the Alarm Generator application, the parsing is carried out within the class called Parser, which is in charge of determining if the data read corresponds to the s7_readings topic and in that case, it analyses whether any input between I1.0 and I1.7 has been activated to generate the corresponding error message, that is, the alarm.

5.3.2.3 Data Sending

In the case of the Alarm Generator application, there are two options to send the alarms generated to Kafka, that is, to write in a Kafka topic. One is through Spark's own API by adding the following dependency.

```
groupId = org.apache.spark
artifactId = spark-sql-kafka-0-10_2.11
version = 2.2.0
```

Figure 47. Spark-SQL-Kafka Dependency

The other way is to create a custom Sink and program a Kafka Producer into it. Both ways are good, but this last way is the one that has been used. The producer that has been programmed into the custom Sink has been programmed asynchronously to achieve the highest possible performance.

In the case of the Spark Transformer application, to send the data to MySQL, a custom JDBC Sink has been created that is responsible for opening the connection with MySQL, sending the data in streaming, and closing the connection when necessary in a secure way. To be able to write data in streaming it is necessary to configure several things such as the checkpoint if there is, the trigger and then the JDBC Sink is called. The figure below shows this process.

```
66    val url = "jdbc:mysql://10.0.0.2:3306/plc4x"
67    val user = "root"
68    val pwd = "root"
69
70    val writer = new JDBCSink(url,user,pwd)
71
72    val startQueries = extracted.writeStream//.option("checkpointLocation", "/build/checkpoint")
73        .trigger(Trigger.ProcessingTime("1 seconds")).foreach(writer).start()
74    startQueries.awaitTermination()
```

Figure 48. Data Sending

The following describes how a custom Sink works (using the Spark Transformer application as an example). The JDBCSink class (which is the custom Sink) consists of three methods: open, process and close. The open method is used to program the opening of the database connection or any other initialization that is necessary.

```
1  import java.sql.{Connection, DriverManager}
2
3  /** Custom sink to send the results to MySQL
4    *
5    *
6    */
7  class JDBCSink(url: String, user: String, pwd: String)
8      extends org.apache.spark.sql.ForeachWriter[org.apache.spark.sql.Row] {
9
10     val driver = "com.mysql.jdbc.Driver"
11     var conn: Connection = _
12
13   /** Initializes the process
14     *
15     * @param partitionId
16     * @param version
17     * @return
18     */
19     def open(partitionId: Long, version: Long): Boolean = {
20       Class.forName(driver)
21       conn = DriverManager.getConnection(url,user,pwd)
22       conn.setAutoCommit(false);
23       System.out.println("got JDBCSink connection")
24
25       true
26     }
```

Figure 49. JDBC Sink Header and Open Method

The process method is used to process the data that arrives in streaming. It should be noted that when working within these methods, other methods cannot be called unless they are within an Object structure.

```
30   /** Executes the process
31     *
32     * @param value a list with the plc parameters and values
33     */
34     def process(value: org.apache.spark.sql.Row): Unit = {
35         val sendQuery = InsertByRow.insertParsing(value)
36         System.out.println("Query value is: " + sendQuery)
37         conn.createStatement.executeUpdate(sendQuery)
38     }
```

Figure 50. JDBC Sink Process Method

The close method is used to schedule the closure of the connection to the database or any other necessary termination. The close method has a parameter called *errorOrNull* that can be very useful since it indicates null in the event that there are no failures or indicates the type of error happened otherwise.

```
42    /** Ends the process
43      *
44      * @param errorOrNull a message error if there is an error or null if there is not an error
45      */
46      def close(errorOrNull: Throwable): Unit = {
47        conn.commit()
48        conn.close
49        System.out.println("JDBCSink: Result of errorOrNull: " + errorOrNull)
50      }
51    }
```

Figure 51. JDBC Sink Close Method

5.3.2.4 Triggers

To notify Spark how often to send the processed data, it is necessary to use a trigger. In the case of both Spark applications, this trigger has been set to one second to maintain a balance between acceptable latency and high throughput.

The following figure shows the code to set the trigger.

```
72      val startQueries = extracted.writeStream//.option("checkpointLocation", "/build/checkpoint")
73          .trigger(Trigger.ProcessingTime("1 seconds")).foreach(writer).start()
```

Figure 52. Trigger

5.3.2.5 Run Modes

The developed system has been designed to be distributed and since it has been developed and tested entirely on the same computer, we can run Spark applications (from the same computer) in local mode, that is, executing it on the computer using single JVM (Java Virtual Machine), or in pseudo-distributed (standalone) mode where the spark's resource manager is used to be able to use a distributed architecture with (in this case) a master and two slaves.

5.4 CLASS DIAGRAM

As explained before, the pipeline has four applications running at the same time. Below is the class diagram for each of them.

5.4.1 S7-PLC4X Application Class Diagram.

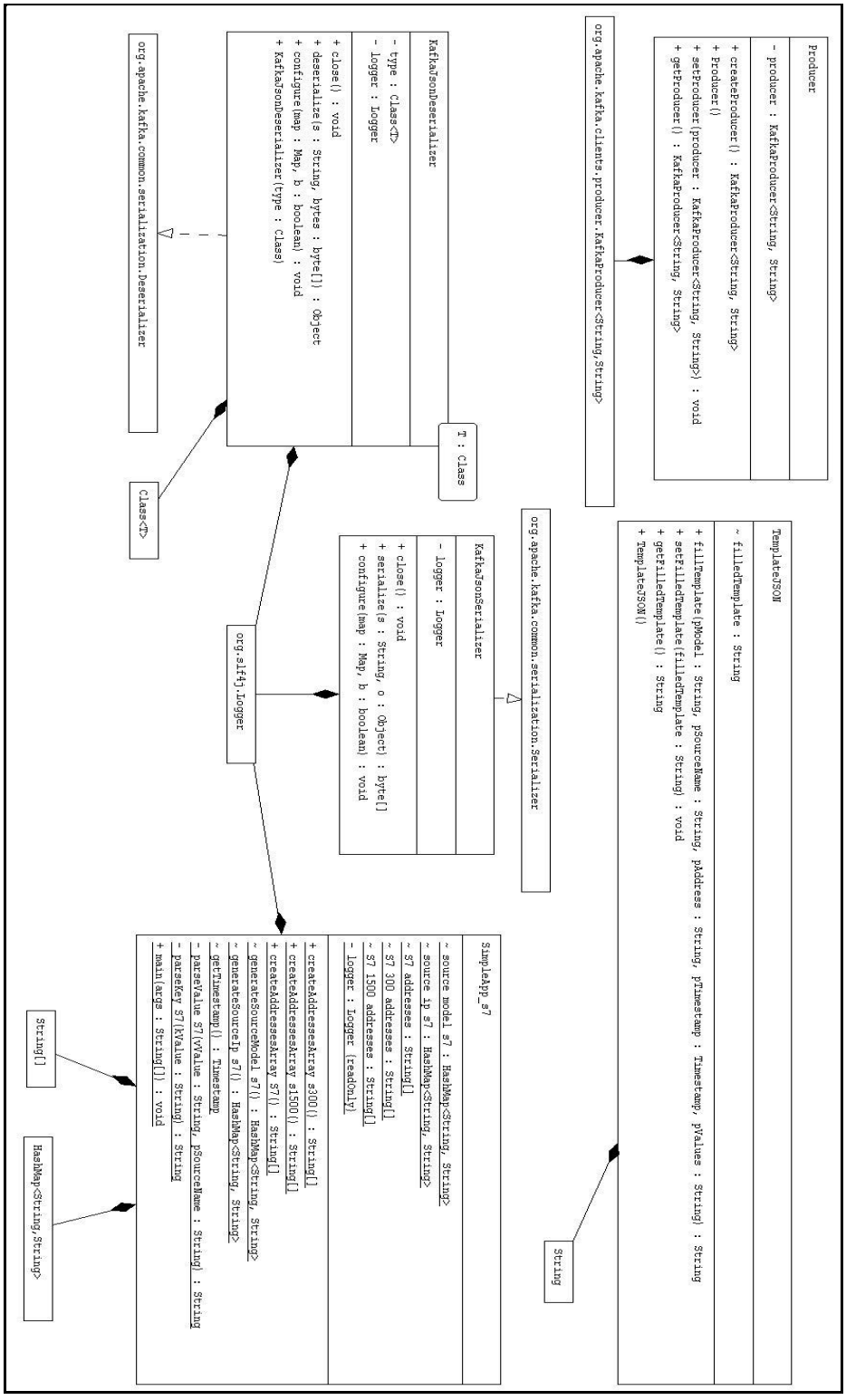


Figure 53. S7-PLC4X Application Class Diagram

5.4.2 Modbus-PLC4X Application Class Diagram.

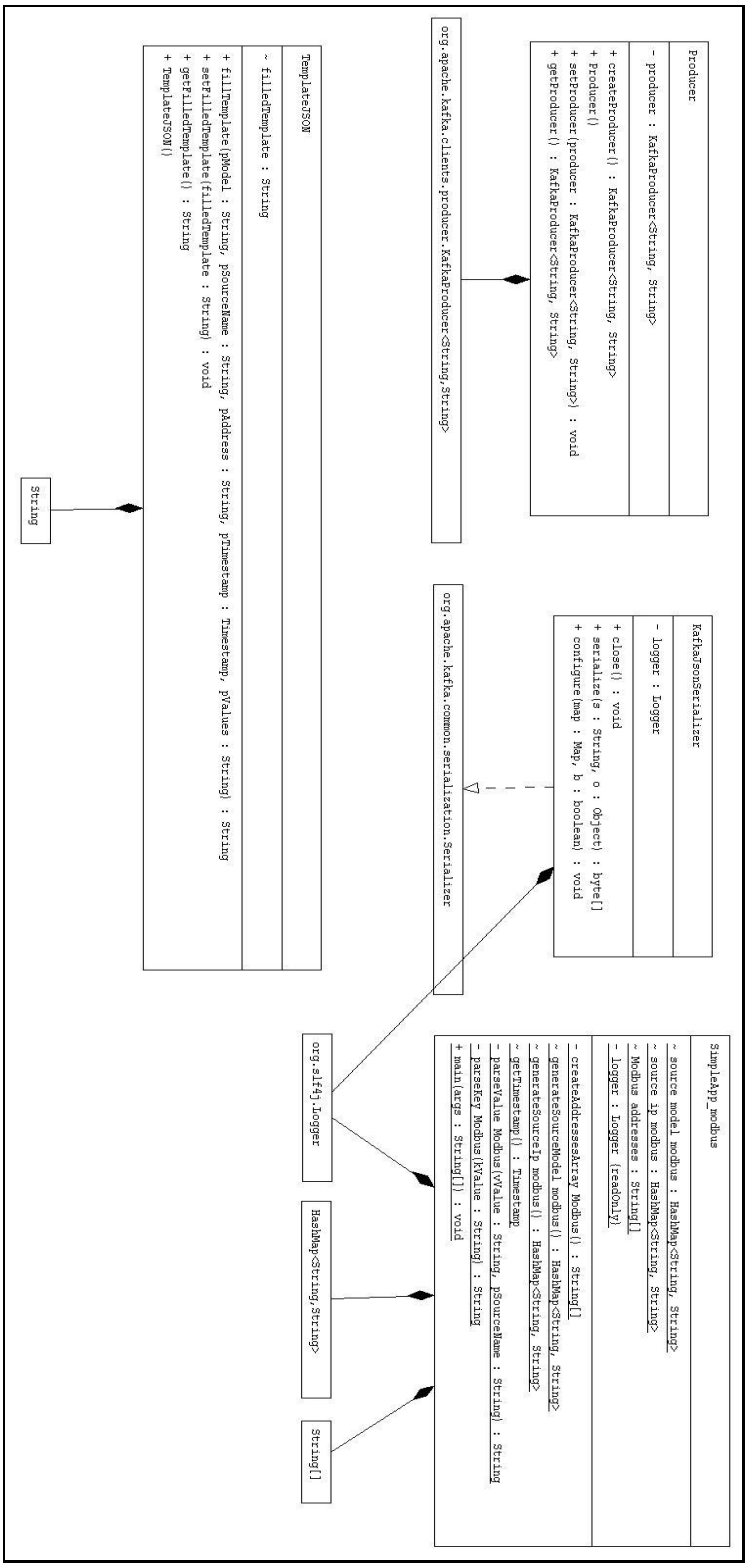


Figure 54. Modbus-PLC4X Aplication Class Diagram

5.4.3 Alarm Generator Application Class Diagram

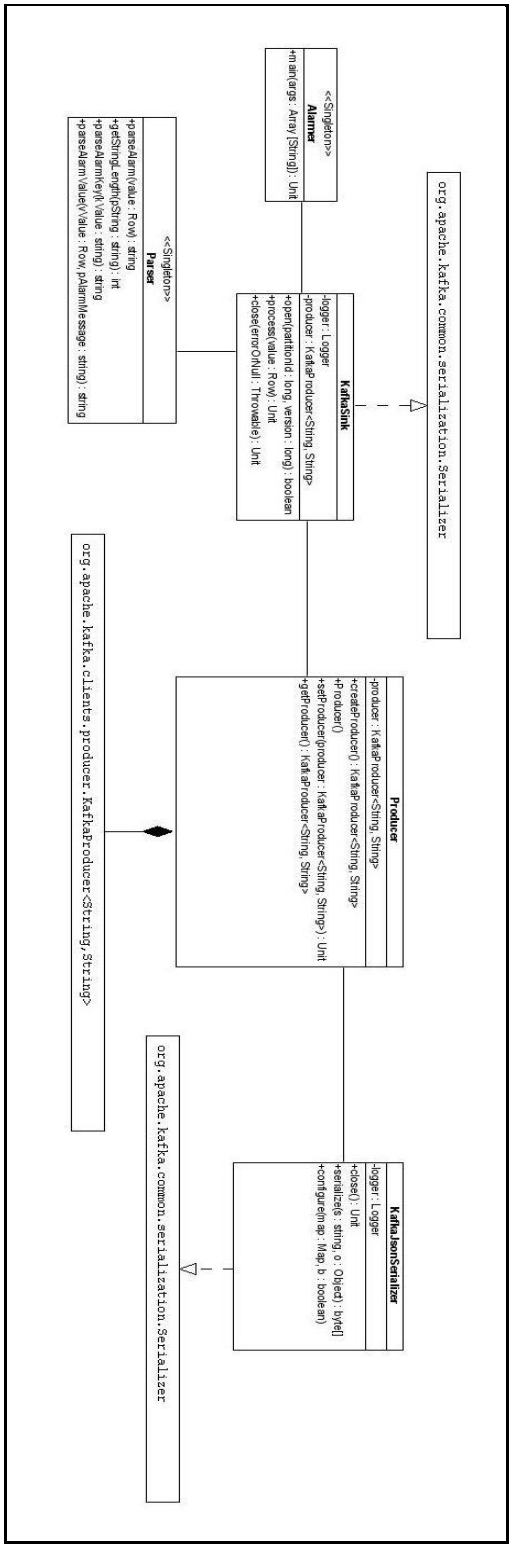


Figure 55. Alarm Generator Application Class Diagram

5.4.4 Spark Transformer Application Class Diagram

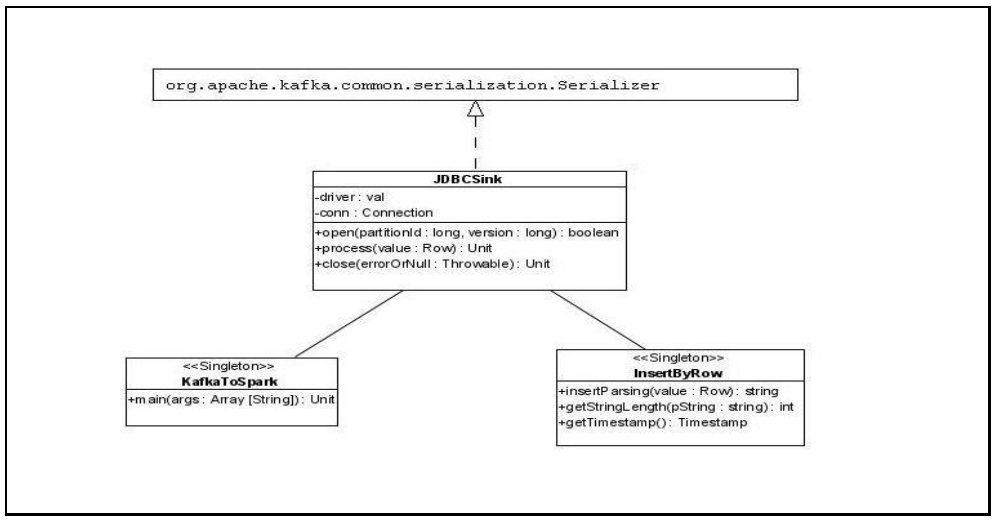


Figure 56. SparkTransformer Application Class Diagram

5.5 METHODS EXPLANATION

Below is a brief explanation of all the methods used in each application.

5.5.1 Modbus PLC4X Application

Method	Description
main(String[] args)	Main method where program execution starts
parseKey_Modbus(String kValue)	Formats the key to facilitate schema creation
parseValue_Modbus(String vValue, String pSourceName)	Formats the value to facilitate schema creation
getTimestamp()	Returns a valid MySQL Timestamp
generateSourceIp_modbus()	Generates a Map with source-IP key-value pair
generateSourceModel_modbus()	Generates a Map with source-model key-value pair
createAddressesArray_Modbus()	Generates the address list of the PLCs

Table 2. SimpleApp_modbus.java

Method	Description
getProducer()	Returns the producer
setProducer(KafkaProducer<String, String> producer)	Sets the producer
Producer()	Constructor
createProducer()	Creates, configures and returns the producer

Table 3. Producer.java

Method	Description
configure(Map map, boolean b)	Empty body
serialize(String s, Object o)	Serializes an object
close()	Empty body

Table 4. KafkaJsonSerializer.java

Method	Description
TemplateJSON()	Constructor
getFilledTemplate()	Returns a filled template
setFilledTemplate(String filledTemplate)	Copies the template to a class attribute
fillTemplate(String pModel,String pSourceName,String pAddress,java.sql.Timestamp pTimestamp, String pValues)	Creates the template with the given values

Table 5. TemplateJSON.java

5.5.2 S7 PLC4X Application

Method	Description
main(String[] args)	Main method where program execution starts
parseKey_S7(String kValue)	Formats the key to facilitate schema creation
parseValue_S7(String vValue, String pSourceName)	Formats the value to facilitate schema creation
getTimestamp()	Returns a valid MySQL Timestamp
generateSourceIp_s7()	Generates a Map with source-IP key-value pair
generateSourceModel_s7()	Generates a Map with source-model key-value pair
createAddressesArray_S7()	Returns an array with the IPs of all the PLCs
createAddressesArray_s1500()	Returns an array with the IPs of all the S-1500 PLCs
createAddressesArray_s300()	Returns an array with the IPs of all the S-300 PLCs

Table 6. SimpleApp_s7.java

Method	Description
getProducer()	Returns the producer
setProducer(KafkaProducer<String, String> producer)	Sets the producer
Producer()	Constructor
createProducer()	Creates, configures and returns the producer

Table 7. Producer.java

Method	Description
configure(Map map, boolean b)	Empty body

serialize(String s, Object o)	Serializes an object
close()	Empty body

Table 8. KafkaJsonSerializer.java

Method	Description
TemplateJSON()	Constructor
getFilledTemplate()	Returns a filled template
setFilledTemplate(String filledTemplate)	Copies the template to a class attribute
fillTemplate(String pModel,String pSourceName,String pAddress,java.sql.Timestamp pTimestamp, String pValues)	Creates the template with the given values

Table 9. TemplateJSON.java

5.5.3 AlarmGenerator Application

Method	Description
main(args: Array[String])	Main method where program execution starts

Table 10. Alarmer.scala

Method	Description
getProducer()	Returns the producer
setProducer(KafkaProducer<String, String> producer)	Sets the producer
Producer()	Constructor

createProducer()	Creates, configures and returns the producer

Table 11. Producer.java

Method	Description
configure(Map map, boolean b)	Empty body
serialize(String s, Object o)	Serializes an object
close()	Empty body

Table 12. KafkaJsonSerializer.java

Method	Description
open(partitionId: Long, version: Long)	To perform initializations. Creates the producer.
process(pValue: org.apache.spark.sql.Row)	Processes the data that arrives in streaming. Here the producer sends the data
close(errorOrNull: Throwable)	It is used to schedule any necessary completion. Here the producer is closed.

Table 13. KafkaSink.scala

Method	Description
parseAlarm(value: org.apache.spark.sql.Row)	Given a list with the Kafka topic source fields returns an insert query
getStringLength(pString: String)	Calculates the length - 1 of a String and returns that length.
parseAlarmKey(kValue: String)	Given a key it returns a parsed String
parseAlarmValue(vValue: org.apache.spark.sql.Row, pAlarmMessage:String)	Given a key it returns a Row it returns a parsed String

Table 14. Parser.scala

Method	Description
TemplateJSON()	Constructor
getFilledTemplate()	Returns a filled template
setFilledTemplate(String filledTemplate)	Copies the template to a class attribute
fillTemplate(String pModel,String pSourceName,String pAddress,java.sql.Timestamp pTimestamp, String pValues)	Creates the template with the given values

Table 15. TemplateJSON.java

5.5.4 SparkTransformer Application

Method	Description
main(args: Array[String])	Main method where program execution starts

Table 16. KafkaToSpark.scala

Method	Description
open(partitionId: Long, version: Long)	Used to program the opening of the database connection.
process(pValue: org.apache.spark.sql.Row)	Processes the data that arrives in streaming. Here the queries are sent.
close(errorOrNull: Throwable)	It is used to schedule the closure of the connection to the database.

Table 17. JDBCSink.scala

Method	Description
insertParsing(value: org.apache.spark.sql.Row)	Given a list with the source kafka topic fields it returns an insert query
getStringLength(pString: String)	Given a string it returns the length of the String minus 1
getTimestamp	Generates the timestamp with the correct format to use in MySQL

Table 18. InsertByRow.scala

5.6 SYSTEM OPERATION

The steps to start up the pipeline are detailed below.

5.6.1 Pipeline deployment

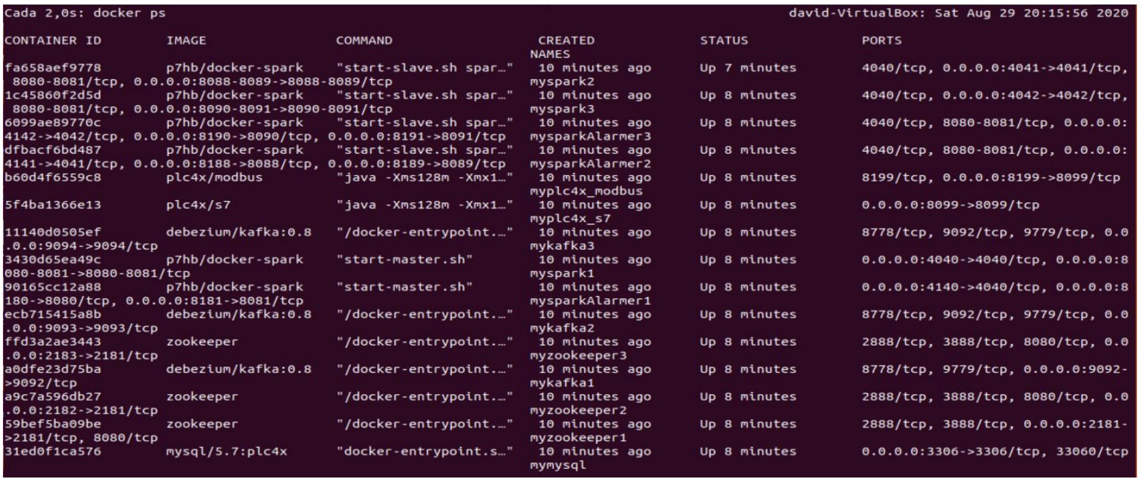
To deploy the pipeline, just go to the directory where the docker-compose.yml file is located and type the following command in the console:

```
docker-compose up -d
```

After a few seconds, all containers should be up and running. This command executes all the commands written in the docker-compose file including the creation of networks, volumes, ... etc. To check the status of the containers you can use a very useful command that is shown below:

```
watch docker ps
```

This command will show us a list of all the containers that are currently running and some important data. The following image shows an example of this.



```
Cada 2,0s: docker ps                                                        david-VirtualBox: Sat Aug 29 20:15:56 2020

CONTAINER ID        IMAGE                   COMMAND                   CREATED             STATUS              PORTS
                                                                     NAMES
fa658aef9778        p7hb/docker-spark      "start-slave.sh spar…"    10 minutes ago      Up 7 minutes        4040/tcp, 0.0.0.0:4041->4041/tcp,
 8080-8081/tcp, 0.0.0.0:8088-8089->8088-8089/tcp                     myspark2
1c45860f2d5d        p7hb/docker-spark      "start-slave.sh spar…"    10 minutes ago      Up 8 minutes        4040/tcp, 0.0.0.0:4042->4042/tcp,
 8080-8081/tcp, 0.0.0.0:8090-8091->8090-8091/tcp                     myspark3
6099ae89770c        p7hb/docker-spark      "start-slave.sh spar…"    10 minutes ago      Up 8 minutes        4040/tcp, 8080-8081/tcp, 0.0.0.0:
4142->4042/tcp, 0.0.0.0:8190->8090/tcp, 0.0.0.0:8191->8091/tcp       mysparkAlarmer3
dfbacf6bd487        p7hb/docker-spark      "start-slave.sh spar…"    10 minutes ago      Up 8 minutes        4040/tcp, 8080-8081/tcp, 0.0.0.0:
4141->4041/tcp, 0.0.0.0:8188->8088/tcp, 0.0.0.0:8189->8089/tcp       mysparkAlarmer2
b60d4f6559c8        plc4x/modbus           "java -Xms128m -Xmx1…"    10 minutes ago      Up 8 minutes        8199/tcp, 0.0.0.0:8199->8099/tcp
                                                                     myplc4x_modbus
5f4ba1366e13        plc4x/s7               "java -Xms128m -Xmx1…"    10 minutes ago      Up 8 minutes        0.0.0.0:8099->8099/tcp
                                                                     myplc4x_s7
11140d0505ef        debezium/kafka:0.8     "/docker-entrypoint.…"    10 minutes ago      Up 8 minutes        8778/tcp, 9092/tcp, 9779/tcp, 0.0
.0.0:9094->9094/tcp                                                  mykafka3
3430d65ea49c        p7hb/docker-spark      "start-master.sh"         10 minutes ago      Up 8 minutes        0.0.0.0:4040->4040/tcp, 0.0.0.0:8
080-8081->8080-8081/tcp                                              myspark1
90165cc12a88        p7hb/docker-spark      "start-master.sh"         10 minutes ago      Up 8 minutes        0.0.0.0:4140->4040/tcp, 0.0.0.0:8
180->8080/tcp, 0.0.0.0:8181->8081/tcp                                mysparkAlarmer1
ecb715415a8b        debezium/kafka:0.8     "/docker-entrypoint.…"    10 minutes ago      Up 8 minutes        8778/tcp, 9092/tcp, 9779/tcp, 0.0
.0.0:9093->9093/tcp                                                  mykafka2
ffd3a2ae3443        zookeeper              "/docker-entrypoint.…"    10 minutes ago      Up 8 minutes        2888/tcp, 3888/tcp, 8080/tcp, 0.0
.0.0:2183->2181/tcp                                                  myzookeeper3
a0dfe23d75ba        debezium/kafka:0.8     "/docker-entrypoint.…"    10 minutes ago      Up 8 minutes        8778/tcp, 9779/tcp, 0.0.0.0:9092-
>9092/tcp                                                            mykafka1
a9c7a596db27        zookeeper              "/docker-entrypoint.…"    10 minutes ago      Up 8 minutes        2888/tcp, 3888/tcp, 8080/tcp, 0.0
.0.0:2182->2181/tcp                                                  myzookeeper2
59bef5ba09be        zookeeper              "/docker-entrypoint.…"    10 minutes ago      Up 8 minutes        2888/tcp, 3888/tcp, 0.0.0.0:2181-
>2181/tcp, 8080/tcp                                                  myzookeeper1
31ed0f1ca576        mysql/5.7:plc4x        "docker-entrypoint.s…"    10 minutes ago      Up 8 minutes        0.0.0.0:3306->3306/tcp, 33060/tcp
                                                                     mymysql
```

Figure 57. Containers Status

The first time the system is started, it will be necessary to copy the jars of the two Spark applications into the /build folder of their containers corresponding to the spark master (mySpark1 and mySparkAlarmer1).

5.6.2 How to Copy Files from the Host to a Container

To copy files from a host (our computer) into a container it is possible to use the following command.

docker cp PATH_IN_HOST CONTAINER_NAME:PATH_IN_CONTAINER

Where PATH_IN_HOST corresponds to the path on our computer where the file to be copied is located, CONTAINER_NAME corresponds to the name of the container to which we want to copy the file and PATH_IN_CONTAINER corresponds to the path within the container where we want to copy the file.

5.6.3 How to Copy Files from a Container to the Host

To copy files from a container to a host (our computer) it is possible to use the following command.

docker cp CONTAINER_NAME:PATH_IN_CONTAINER PATH_IN_HOST

Where CONTAINER_NAME corresponds to the name of the container from which we want to copy the file and PATH_IN_CONTAINER corresponds to the path within the container from where we want to copy the file. And PATH_IN_HOST corresponds to the path on our computer to where we want to copy the file.

5.6.4 Container Access

If it is necessary to access inside of the containers, this can be done using the following command.

```
docker exec -it containerName bash
```

where containerName should be replaced by the name of a container. And after executing this command, we will enter inside the container directly to the folder that has been set as entry point in the docker-compose.

5.6.5 Stop Containers

To stop a container there is the following command.

```
docker stop containerName1
```

Where containerName1 corresponds to the name of the container we want to stop. If there is more than one container that we want to stop at the same time, we can add a space and continue writing more names.

5.6.6 Start Containers

To start a container there is the following command.

```
docker start containerName1
```

Where containerName1 corresponds to the name of the container we want to start. If there is more than one container that we want to start at the same time, we can add a space and continue writing more names.

5.6.7 Stop and Remove Containers

To stops containers and removes containers, networks, volumes, and images created by `up` the following command is available.

```
docker-compose down
```

5.6.8 Restart Containers

To restart containers the following command is available.

```
docker restart containerName1
```

Where containerName1 corresponds to the name of the container we want to restart. If there is more than one container that we want to restart at the same time, we can add a space and continue writing more names.

5.6.9 Launch the Spark applications

Once the Spark applications have been copied to their corresponding directories within the Spark1 and SparkAlarmer1 containers, the applications can be launched.

It should be noted that this copy should only be done once, since unless the volumes are deleted, they will store all the container data, including the copied applications.

The two available launch modes for these Spark applications will be explained below. For an explanation about the Run Modes, please refer to chapter 5.3.2.5 (Run Modes).

5.6.9.1 Local Mode

To launch an application in local mode, we must enter the container (which will take us directly to the build folder since we have specified it in the entry point of the docker-compose.yml file) and use the following command.

```
spark-submit --master local[*] alarmGenerator.jar
```

Where alarmGenerator.jar corresponds to the jar name of the Alarm Generator application. If we wanted to launch an application with another name (as is the case with the other application called Spark Transfomer) we should change it to sparkTransformer.jar.

It should be noted that the asterisk indicates the maximum number of system cores that can be used. If we change the asterisk for a number, we can directly indicate how many cores we want to use at most.

5.6.9.2 Pseudo-Distributed (Standalone) Mode

To launch an application in pseudo-distributed mode, we must enter the container (which will take us directly to the build folder since we have specified it in the entry point of the docker-compose.yml file) and use the following command.

```
spark-submit --master spark://10.0.0.18:7077 alarmGenerator.jar
```

Where alarmGenerator.jar corresponds to the jar name of the Alarm Generator application. If we wanted to launch an application with another name (as is the case with the other application called Spark Transfomer) we should change it to sparkTransformer.jar.

The word "spark" before the IP address indicates that the resource manager to be used is Spark's own (and that is why it is called Standalone). If we had a different resource manager such as "Apache Hadoop Yarn" we would have to change the word spark to yarn. The IP 10.0.0.18 corresponds to the container where the alarmGenerator application is located. The number 7077 corresponds to the port we must use with this Spark image.

6. TEST PLAN

6.1 TEST ORGANIZATION STRATEGIES

In order to check the operation of the pipeline in general and of the different modules that compose it (understanding module at the application or cluster level) in particular, a series of tests has been carried out in a real environment with 39 PLCs operating at the same time. Of these, 38 correspond to S7-300 and S7-1500 PLCs and 1 corresponds to Modicon TSX Quantum.

The tests specified below cover all the validation tests specified in chapter 3.3.2 (Validation Tests) that must be fulfilled to validate the system.

6.1.1 General Tests Runs

Regarding the general tests, the overall operation of the pipeline has been tested. For this, the following test has been carried out.

6.1.1.1 Test Procedure

The procedure to carry out the test has been exactly the one indicated below.

Turn on all parts of the pipeline and check that the pipeline can operate flawlessly and store PLC data and alarms generated if all parts of the pipeline are in operation.

6.1.1.2 Test Case

The test that has been carried out consists of testing the general operation of the pipeline. The only precondition is to start with the pipeline completely off.

6.1.1.3 Test Results

After performing this test, satisfactory results have been obtained, finding no faults during the process. As a consideration, it should be noted that to generate the alarms, the Kafka data must be read by the Alarm Generator application, process it and then send it again to Kafka so that finally the other Spark application takes that data and sends it to the database. Due to this, there is a small latency (but greater than in the case of the rest of the data that does not have to be processed by the Alarm Generator) that does not significantly impact the performance of the system.

During this test it has also been verified that the data visualization was carried out normally by using the Kafka Tool that allows us to see the data that is in the Kafka topics. In addition, it has

also been proceeded to verify that the data arrived correctly at the database and that this data could be displayed in all the tables without any problem.

It should be noted that the pipeline has been tested by running the two Spark applications both in local mode and in pseudo-distributed mode, obtaining satisfactory results in both cases. However, given the number of containers running at the same time (fifteen) and the limited computer resources to deal with all of them, slightly better performance is seen when running in local mode.

6.1.2 Consumer Shutdown Test (Alarm Generator)

6.1.2.1 Test Procedure

The procedure to carry out the test has been exactly the one indicated below.

Turn on the pipeline except the alarm generator and check that it is able to operate flawlessly and store the data of the PLCs.

6.1.2.2 Test Case

The test that has been carried out consists of testing the operation of the pipeline while one of its consumers (Alarm Generator) is turned off. The only precondition is to start with the pipeline completely off.

6.1.2.3 Test Results

It has been verified that by starting up the entire pipeline, except the Alarm Generator, the pipeline works flawlessly, being able to store the data sent by the PLCs in the database.

Note that since the function of the Alarm Generator is to generate the alarms and in this test this application is off, the alarms will not be generated until it is turned on again so the only data that will be stored in the database will be the data generated by the PLCs.

6.1.3 Consumer Shutdown Test (Spark Transformer)

6.1.3.1 Test Procedure

The procedure to carry out the test has been exactly the one indicated below.

Turn on the pipeline except the data load application (Spark Transformer) and check that it is able to operate flawlessly and generate the alarms

6.1.3.2 Test Case

The test that has been carried out consists of testing the operation of the pipeline while one of its consumers (Spark Transformer) is turned off. The only precondition is to start with the pipeline completely off.

6.1.3.3 Test Results

It has been verified that after turning on the entire pipeline except the Spark Transformer application (which is responsible for sending the data from Kafka to the database), the alarms generated by the Alarm Generator application are generated normally and stored in Kafka as intended.

Note that since the function of the Spark Transformer is to send the Kafka data to the database, while this application is turned off, the data will not flow from Kafka to the database. But when this application resumes, it will continue to send Kafka's data to the database, right where it left off.

6.1.4 Producers Shutdown Test

6.1.4.1 Test Procedure

The procedure to carry out the test has been exactly the one indicated below.

Turn on the pipeline and stop the producers to verify that the pipeline remains in a stable state, keeping it waiting without the other systems presenting an error.

6.1.4.2 Test Case

La prueba que se ha realizado consiste en probar el funcionamiento del oleoducto mientras ambos productores están apagados. La única condición previa es comenzar con la tubería completamente apagada.

6.1.4.3 Test Results

It has been verified that after turning on the pipeline and turning off the consumers, the rest of the pipeline is able to remain in a stable state.

If, during the producers' shutdown, there is still data in Kafka to be processed by any of the Spark applications, this processing continues normally until all Kafka data is finished. Once one or both producers are turned back on, the system continues running smoothly.

6.1.5 Consumers Resumption Test

6.1.5.1 Test Procedure

The procedure to carry out the test has been exactly the one indicated below.

Turn on the pipeline, launch consumers, and verify that consumers are able to pick up reading Kafka data from where they left off before stopping

6.1.5.2 Test Case

The test that has been carried out consists of testing the operation of the pipeline when the consumers resumes. The only precondition is to start with the pipeline completely off.

6.1.5.3 Test Results

It has been verified that after turning on the pipeline and then turning on the consumers (the two Spark applications), they are able to continue reading the Kafka data where they left off before shutting down.

7. PROJECT PLANNING

7.1 TIME PLANNING

Time planning is compound of phases, tasks and a final delivery

7.1.1 Work Breakdown Structure

WBS depicts in a hierarchical way the different tasks and the delivery that make up the work. The purpose of the WBS is to organize and define the total scope of the project. Figure below depicts the tasks decomposition.

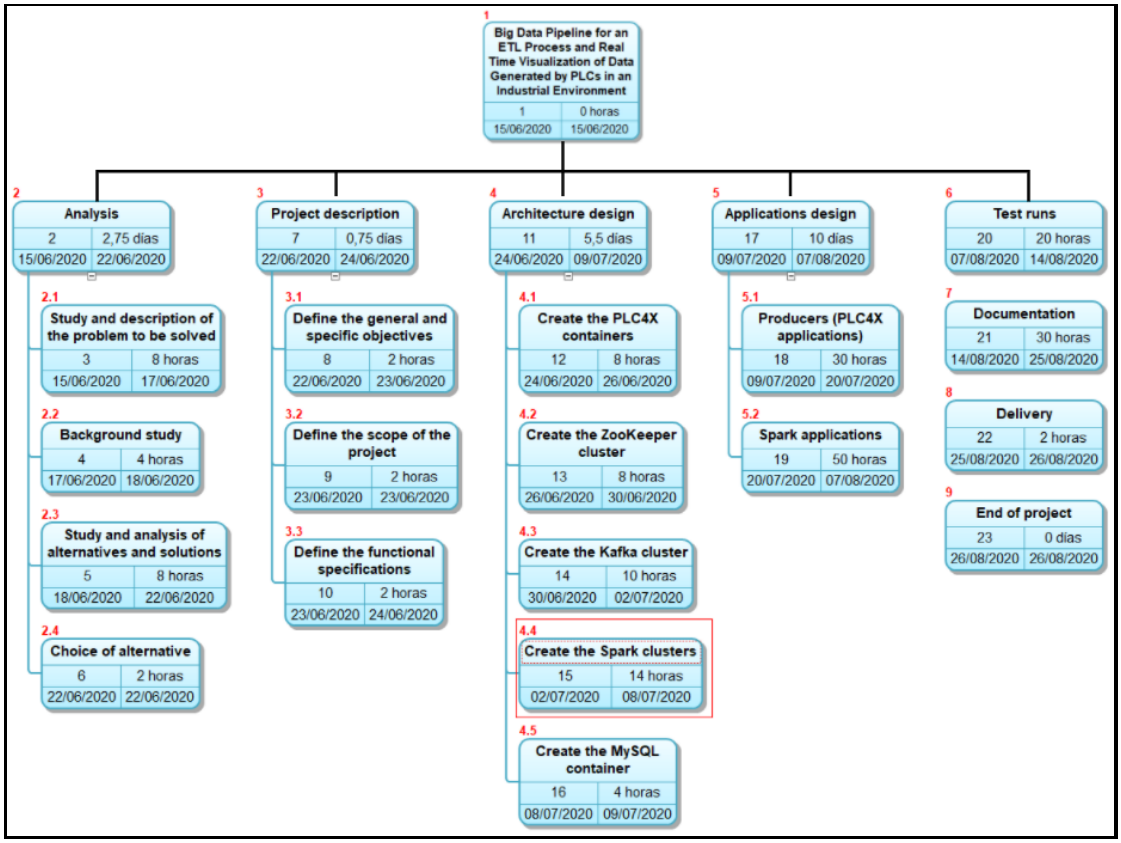


Figure 58. WBS

7.1.2 Phases and Tasks

This section describes the phases and tasks that make up the whole project. Below is shown the project tasks organized in order of execution.

1. Project start
2. Analysis
    2.1 Study and description of the problem to be solved
    2.2 Background study
    2.3 Study and analysis of alternatives and solutions
    2.4 Choice of alternative
3. Project description
    3.1 Define the general and specific objectives
    3.2 Define the scope of the project
    3.3 Define the functional specifications
4. Architecture design
    4.1 Create the PLC4X containers
    4.2 Create the ZooKeeper cluster
    4.3 Create the Kafka cluster
    4.4 Create the Spark clusters
    4.5 Create the MySQL container
5. Applications design
    5.1 Producers (PLC4X applications)
    5.2 Spark applications
6. Test runs
7. Documentation
8. Delivery
9. End of project

7.1.3 Description of each Task and the Estimated Time

Below are the description of each task and the estimated time (Project start and End of project are excluded because they only mark the start and end milestone).

Task: 2

Name: Analysis

Description: Perform an initial analysis to study the main objectives and the different possibilities of development and implementation. It will also study what knowledge is necessary to carry out the project.

Estimated Time: 22 hours (sum of subtasks)

Task: 2.1

Name: Study and description of the problem to be solved.

Description: A study is done to understand the problem and to be able to describe it clearly.

Estimated Time: 8 hours

Task: 2.2

Name: Background study

Description: A study is carried out to understand the background.

Estimated Time: 4 hours

Task: 2.3

Name: Study and analysis of alternatives and solutions

Description: A study is carried out to analyse all the alternatives and possible solutions.

Estimated Time: 8 hours

Task: 2.4

Name: Choice of alternative

Description: The best alternative for the problem is chosen.

Estimated Time: 2 hours

Task: 3

Name: Project description

Description: Description of the project indicating the objectives, scope and specifications.

Estimated Time: 6 hours (sum of subtasks)

Task: 3.1

Name: Define the general and specific objectives

Description: This task is responsible for clearly defining all the objectives, both general and specific.

Estimated Time: 2 hours

Task: 3.2

Name: Define the scope of the project

Description: This task is responsible for defining the scope of the project and what is outside of it.

Estimated Time: 2 hours

Task: 3.3

Name: Define the functional specifications

Description: This task is responsible for defining the functional specifications unequivocally

Estimated Time: 2 hours

Task: 4

Name: Architecture design

Description: This task is responsible for designing the architecture of the pipeline so that the pipeline can work properly.

Estimated Time: 44 hours (sum of subtasks)

Task: 4.1

Name: Create the PLC4X containers

Description: This task is responsible for creating and configuring the PLC4X containers.

Estimated Time: 8 hours

Task: 4.2

Name: Create the ZooKeeper cluster

Description: This task is in charge of creating and configuring the ZooKeeper cluster with 3 nodes.

Estimated Time: 8 hours

Task: 4.3

Name: Create the Kafka cluster

Description: This task is responsible for creating and configuring the Kafka cluster with 3 nodes.

Estimated Time: 10 hours

Task: 4.4

Name: Create the Spark clusters

Description: This task is responsible for creating and configuring the two Spark clusters, each with 3 nodes.

Estimated Time: 14 hours

Task: 4.5

Name: Create the MySQL container

Description: This task is responsible for creating the database container and creating the initial configuration as well as creating its tables.

Estimated Time: 4 hours

Task: 5

Name: Applications design

Description: This task is responsible for programming the 4 applications necessary for the operation of the pipeline.

Estimated Time: 80 hours (sum of subtasks)

Task: 5.1

Name: Producers (PLC4X applications)

Description: This task is in charge of programming the 2 producer applications (PLC4X applications)

Estimated Time: 30 hours

Task: 5.2

Name: Spark applications

Description: This task is responsible for programming the 2 Spark applications

Estimated Time: 50 hours

Task: 6

Name: Test runs

Description: This task is in charge of performing the tests to validate the operation of the complete system (composed of the pipeline and the data visualization)

Estimated Time: 20 hours

Task: 7

Name: Documentation

Description: This task is responsible for preparing the documentation

Estimated Time: 30 hours

Task: 8

Name: Delivery

Description: This task is responsible for delivering the complete system.

Estimated Time: 2 hours

7.1.4 Deliverables

When the whole system is finished, the developed software and project documentation will be delivered.

7.1.5 Project Agenda

The planning of work days has been done by counting 8-hour workdays from Monday to Friday following a standard calendar of the city where the work is performed (Bilbao). The holidays according to the Bilbao work calendar are as follows.

Date	Holidays
25th July	Santiago Apostol
31st July	San Ignacio de Loyola
15th August	Asuncion de la Virgen
28th August	Viernes de la Semana Grande

Table 19. Calendar

The estimated time for the project completion is projected in 204 hours.

7.1.6 Schedule

The figure below shows a Gantt chart that represents the temporal structure of the tasks where it is possible to see the development of each task over time.

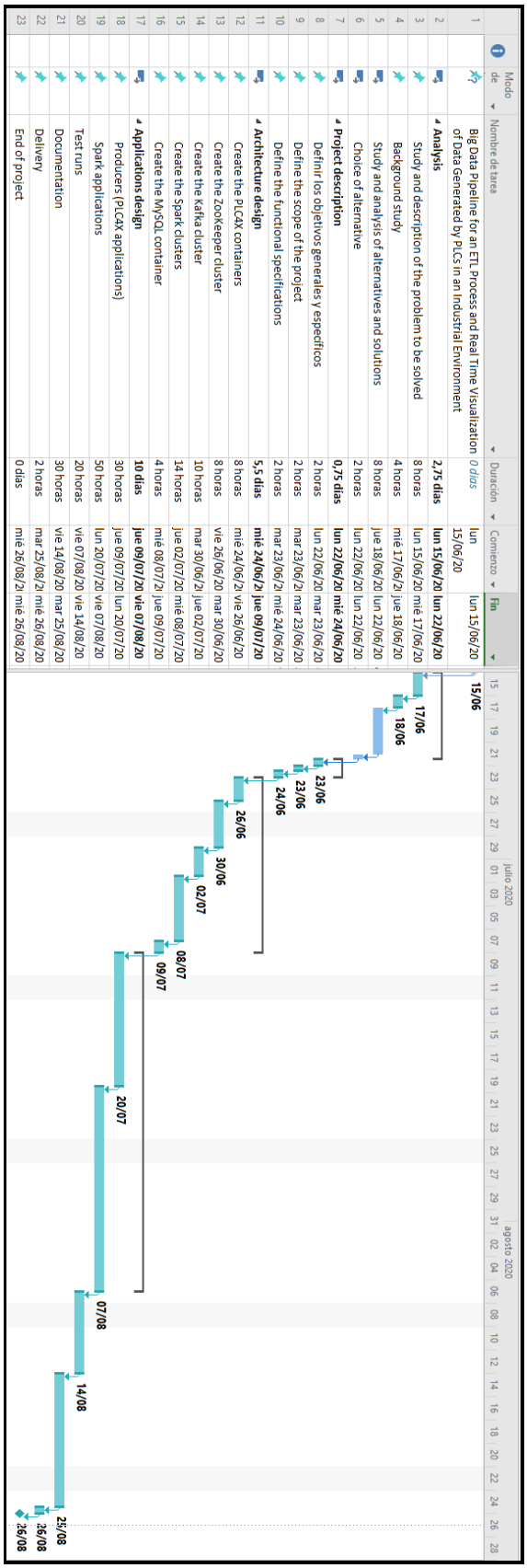


Figure 59. Gantt Chart

8. COST MANAGEMENT

To estimate the cost of the project, the following data are considered: human resources, material resources (both software and hardware), amortization and applicable taxes. Human resources have been calculated at a rate of 20 euros for each hour corresponding to the salary of a Big Data developer.

A fixed cost of 1,45 € /use has been determined for internet connection, electricity and other fixed costs derived from the use of the office and that can be considered independent with respect to the time of use.

Amortizations have been calculated considering a 2 years amortization time, 250 workdays and 8-hour workdays. Therefore, the amortization time is as follows: *2 years x 250 workdays x 8-hour workdays = 4000 hours.* The calculation of the unit cost of amortization is the division of the unit cost by the amortization time. The tables with the project cost calculations are shown below.

Concept	Cost
David Zamora	20 € / h

Table 20. Labor Resources

Concept	Cost
Laptop	1300 €

Table 21. Material Resources (Hardware)

Concept	Unit Cost	Number of Licenses
Linux Ubuntu	0 €	1
Virtual Box	0 €	1
Draw.io	0 €	1
Kafka Tool	0 €	1
Microsoft Project	0 €	1 (Student license)
Apache PLC4X	0 €	1
Apache ZooKeeper	0 €	1
Apache Kafka	0 €	1
Apache Spark	0 €	1

Table 22. Material Resources (Software)

Concept	Work Hours	Overtime	Cost	Overtime Cost	Amount
David Zamora	204	0	20 € / h	0	4080 €
Total	204	0	20 € / h	0	4080 €

Table 23. Labor Resources Costs

Concept	Units	Uses	Cost	Amount
Laptop	1	58	1.45 € / use	84.1 €
Total				84.1 €

Table 24. Material Resources Costs

Concept	Unit Cost	Amortization Time	Amortization Unit Cost	Time of Use	Amount
Laptop	1300 €	*4000 hours*	0.325	204 hours	66.3 €
Linux Ubuntu	0 €	*4000 hours*	0.000	204 hours	0 €
Virtual Box	0 €	*4000 hours*	0.000	204 hours	0 €
Draw.io	0 €	*4000 hours*	0.000	2 hours	0 €
Kafka Tool	0 €	*4000 hours*	0.000	40 hours	0 €
Microsoft Project	0 €	*4000 hours*	0.000	10 hours	0 €
Apache PLC4X	0 €	*4000 hours*	0.000	55 hours	0 €
Apache ZooKeeper	0 €	*4000 hours*	0.000	40 hours	0 €
Apache Kafka	0 €	*4000 hours*	0.000	40 hours	0 €
Apache Spark	0 €	*4000 hours*	0.000	40 hours	0 €
MySQL	0 €	*4000 hours*	0.000	30 hours	0 €
Total					66.3 €

Table 25. Amortization Table (Hardware and Software)

Concept	Amount
Work Resources	4080.00 €
Material Resources	84.1 €
Fixed Cost	0.00 €
Amortization	66.3 €
SUMA	
General Expenses (10 %)	423.04 €
Profit (16%)	676.864 €
SUBTOTAL	
IVA (21 %)	1119.364 €
GRAND TOTAL	6449.668 €

Table 26. Budget Table

Therefore, the budget amounts to six thousand four hundred and forty-nine point six hundred sixty-eight euros (6449.668 €).

Signature: David Zamora Arranz

Bilbao, 30th August 2020

9. CONCLUSIONS

The final results obtained with the development of this project demonstrates how the pipelines developed with big data technologies such as Apache ZooKeeper, Apache Kafka and Apache Spark, in combination with the set of libraries that Apache PLC4X provides, are a very powerful tool to treat and manage the data of a large number of PLCs in real time.

In addition, this pipeline provides great advantages that are not usually common in software applications developed in industrial environments to communicate with PLCs, such as horizontal scalability, fault tolerance, decoupling of producer systems from consumers and an easy and fast deployment without have to stop any other systems.

Also noteworthy is the use of Apache Kafka as a central part of the pipeline that provides the ability to decouple producers from consumers, so any new application that needs to connect to the pipeline either as producer or consumer, can be connected and disconnected without affect the rest of the systems. This avoids the problems that are generated with downtime in factories, so the advantage it brings is great.

And finally, it is important to highlight the importance of Apache PLC4X within this pipeline, which provides a set of libraries to be able to access PLCs that use different protocols and extract data from them periodically. This technology is still very new and, in the future, it may add new features, but what is certain today is that it is a very interesting tool, with many possibilities and very good performance.

For all these reasons, I consider that this pipeline is especially suitable for industrial-level implementations and that it offers very good characteristics and exceptional performance for industrial needs.

10. LIST OF ACRONYMS

ETL	Extract, Transform and Load
PLC	Programmable Logic Controller
SCADA	Supervisory Control and Data Acquisition
OPC-UA	Open Platform Communications Unified Architecture
PC	Personal Computer
TCP	Transmission Control Protocol
IEC	International Electrotechnical Commission
MPI	Multi Point Interface
PPI	Point to Point Interface
API	Application Programming Interface
SQL	Structured Query Language
NoSQL	Non-SQL / Not only SQL
HDFS	Hadoop Distributed File System
JSON	JavaScript Object Notation
XML	Extensible Markup Language
SSD	Solid State Drive
DDR	Double Data Rate
RAM	Random Access Memory
RDD	Resilient Distributed Datasets
RDBMS	Relational database management system
GNU	General Public License
ACL	Access Control List
REST	Representational State Transfer
SRS	Software Requirements Specification
JDBC	Java Database Connectivity
JVM	Java Virtual Machine

WBS	Work Breakdown Structure

11. BIBLIOGRAPHY / REFERENCES

[1] Apache Kafka – Documentation.
http://kafka.apache.org/documentation.html#producerconfigs

[2] Apache Kafka – Official Website. https://kafka.apache.org/

[3] Apache PLC4X – Tools. (2020). https://plc4x.apache.org/users/tools/capture-replay.html

[4] Apache PLC4X - the universal protocol adapter for industrial IoT. (2020).
https://plc4x.apache.org/index.html

[5] Apache PLC4X - Official Website. (2020). https://plc4x.apache.org/

[6] Apache Spark - Official Website. https://spark.apache.org

[7] Apache ZooKeeper – Official Website. (2020) https://zookeeper.apache.org/

[8] API spark 2.1.1: Machine learning library (MLlib) guide
.https://spark.apache.org/docs/2.1.1/ml-guide.html

[9] API spark 2.2.0 - structured streaming programming
guide.https://spark.apache.org/docs/2.2.0/structured-streaming-programming-guide.html#input-sources

[10] API spark 2.3.0 - structured streaming programming guide.https://spark.apache.org/docs/2.3.0/structured-streaming-programming-guide.html#input-sources

[11] Carlsson, T. (2020). Industrial network market shares 2020 according to HMS networks.https://www.hms-networks.com/news-and-insights/news-from-hms/2020/05/29/industrial-network-market-shares-2020-according-to-hms-networks

[12] Chambers, B., & Zaharia, M. (2018). Spark: The definitive guide. Sebastopol, CA: O'Reilly.

[13] Codd, E. F. (1970). A relational model of data for large shared data banks. Communications of the ACM, 13(6), 377-387.

[14] Deshpande, S. (2019). Scala vs python vs R vs java - which language is better for spark & why? https://www.knowledgehut.com/blog/programming/scala-vs-python-vs-r-vs-java

[15] Docker - Official Website. https://www.docker.com/why-docker

[16] Docker Official Guide. (2019). https://docs.docker.com/engine/docker-overview/#docker-objects

[17] ETL (extract, transform, and load) process.https://www.guru99.com/etl-extract-load-process.html

[18] Gandhi, P. (2018). Apache Spark: Python vs. scala.https://www.kdnuggets.com/2018/05/apache-spark-python-scala.html

[19] Hunt, P., Konar, M., Junqueira, F.P., Reed, B. (2010). ZooKeeper: Wait-free coordination for internet-scale systems. USENIX annual technical conference.

[20] Kafkatool - official website. (2020). https://www.kafkatool.com/

[21] Kleppmann, M. (2017). Designing data-intensive applications. Sebastopol, CA: O'Reilly Media.

[22] LIBNODAVE -- exchange data with siemens PLCs. http://libnodave.sourceforge.net/

[23] Mejran, M. (2018). Apache spark: Scala vs. java v. python vs. R vs. SQL.https://mindfulmachines.io/blog/2018/6/apache-spark-scala-vs-java-v-python-vs-r-vs-sql26

[23] Narkhede, N. (2017). Exactly once semantics are possible: Here's how kafka does it.https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

[24] Narkhede, N., Shapira, G., & Palino, T. (2017). Kafka: The definitive guide. Sebastopol, CA: O'Reilly. Retrieved from /z-wcorg/

[25] Snap7 - Step7 open source ethernet communication suite. http://snap7.sourceforge.net/

[26] Turnbull, J. (2014). The docker book Turnbull Press.

[27] Viswanath, V. (2018). A tour of spark structured streaming.http://vishnuviswanath.com/spark_structured_streaming.html

[28] Weiß, T. (2018). STEP7-Workbook für S7-1200/1500 und TIA Portal.

[29] White, T. (2015). Hadoop:The definitive guide. Sebastopol, CA: O'Reilly.

[30] Zamora, D. (2019). Big Data Pipeline for an ETL Process in a Banking System. UPV / EHU).

[31] ZooKeeper 3.5 documentation. (2019). https://zookeeper.apache.org/doc/r3.5.5/zookeeperOver.html