Graduated in Computer Engineering
Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros Informáticos

BACHELOR’S DEGREE PROJECT

Development of an autoscaling Big Data system with Docker and Kubernetes

Author: Luis Ballestín Carrasco
Directors: Luis M. Gómez Henríquez
Rubén Casado Tejedor

MADRID, JULY 2017
A los profesores que han cambiado mi forma de pensar. En especial a Fernando y a Santos.
A mis amigos de siempre y en especial a Carlos, porque hemos llegado hasta aquí juntos.
A mis compañeros de paellas y molletes.
A Cris, por apoyarme todo este tiempo.
A mi familia, por hacer todo esto posible.
A mi abuela.
# Index

- Index 
- Figure Index
- Resumen
- Abstract
- 1. INTRODUCTION
  - 1.1. Aims and goals
  - 1.2. Chapters summary
- 2. STATE OF THE ART
  - 2.1. Big Data
  - 2.2. Big Data architecture
    - 2.2.1. Lambda Architecture
    - 2.2.2. Kappa Architecture
  - 2.3. Software containers
- 3. TECHNOLOGIES INVOLVED
  - 3.1. Docker
  - 3.2. Docker Compose
  - 3.3. Kubernetes
  - 3.4. Apache Kafka
  - 3.5. Apache Zookeeper
  - 3.6. Apache Flink
  - 3.7. Couchbase
- 4. ARCHITECTURE DEVELOPMENT
  - 4.1. Designing the Docker images
4.1.1. Baseimg ........................................................................................................... 17
4.1.2. Kafka ............................................................................................................. 17
4.1.3. Zookeeper ..................................................................................................... 18
4.1.4. Flink ............................................................................................................. 18
4.1.5. Couchbase .................................................................................................... 18
4.2. Integration tests with Docker Compose ........................................................... 19
4.3. Kubernetes implementation ........................................................................... 20
  4.3.1. Horizontal Pod autoscaling ...................................................................... 22
5. CONCLUSIONS AND RESULTS .......................................................................... 23
6. BIBLIOGRAPHY AND REFERENCES .................................................................... 24
## Figure Index

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Lambda architecture [4]</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Containers vs VMs [6]</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>Docker logo</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Docker Compose logo</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>Kubernetes logo</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>Kafka logo</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>Zookeeper logo</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>Flink logo</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>Couchbase logo</td>
<td>14</td>
</tr>
<tr>
<td>11</td>
<td>Couchbase cluster example</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>Architecture overview</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>Example of a Compose service definition</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>Pod example [23]</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>Kubernetes service definition example</td>
<td>21</td>
</tr>
</tbody>
</table>
**Resumen**

El objetivo principal de este Proyecto consistía en el desarrollo de una arquitectura Big Data autoescalable utilizando para ésto Docker y Kubernetes.

Para obtener dicho resultado, el Proyecto se ha dividido en 3 etapas principales: En la primera etapa había que primero, definir qué herramientas y aplicaciones utilizariamos para componer la arquitectura y, segundo, conseguir empaquetar dichas aplicaciones en contenedores software con Docker.

La segunda etapa del Proyecto, consistió en la elaboración de los tests de integración del Sistema necesarios para poder asegurar el correcto funcionamiento de los componentes del mismo antes de proceder con la tercera etapa.

Finalmente, se implementó el Sistema utilizando Kubernetes para conseguir que éste fuera capaz de reconocer, automáticamente, las cargas de proceso a las que está sometido en cada momento y poder así ajustar sus características en función de las necesidades de cada momento.
Abstract

The main objective of this project consisted on the development of an autoscaling Big Data architecture with Docker and Kubernetes.

To obtain this, the project was divided into 3 different stages: During the first stage, it was necessary to first, study the different options to conclude which were the best possible tools and applications to use in the architecture, and second, to package the chosen applications as software containers using Docker.

At the second stage, we created the system integration tests necessary to ensure the correct functioning of the system and each of its parts before we could proceed with the third stage.

Finally, the system was implemented using Kubernetes with the purpose of enabling the system to recognize by itself the amount of resources each of its parts is consuming and to auto scale consequently to adjust to every moment needs.
1. INTRODUCTION

1.1. Aims and goals

Over the last few years, individuals and organizations have become more and more dependent on the data they generate. More than 2.5 exabytes of data are produced and distributed online every day. We can see that nowadays this is affecting not only big enterprises like Amazon or Google, but also medium level companies or governments are leading with enormously large and complex datasets. This is what we call Big Data.

For an organization or company, to be prepared for analyzing Big Data may be something completely unaffordable, not so much because of the money but the time and human resources it needs. First they need the infrastructure to store this massive quantity of data. Second they need the technical capabilities to create a distributed software architecture able to process this data. And finally, if they manage to get their big data architecture working, maybe in a cloud environment, they need to keep watching over it making sure the system does not fail.

The main goal of this project is to avoid this steps creating a big data architecture that can be deployed in minutes and watches over itself auto scaling the number of servers for data processing in order to fit perfectly the demand. Wrapping up all the applications in software containers using Docker and orchestrating the full system with Kubernetes we can make an easy and fast to deploy Big Data architecture.
1.2. Chapters summary

On the following chapters of this Project we will first try to explain some necessary concepts to understand the project. Then, we will take a look at the main technologies involved on the architecture, discussing why are they important to the project. The 4th chapter will expose how the project was developed in three different stages. Finally we will show the results and conclusions of the project and also how it could be improved in the future.
2. STATE OF THE ART

In this chapter I will try to introduce some basic concepts for a complete understanding of this project.

2.1. Big Data

Big Data is usually defined as a collection of data sets enormously large and complex, which can be structured, semi-structured and/or unstructured, and therefore cannot be processed using traditional databases. The main characteristics of Big Data are known as the 3Vs: volume, velocity and variety. [1]

- Volume: the volume of data to process goes beyond terabytes of data.
- Velocity: the velocity concerns at the same time to the speed the data is generated and the need to process this data quickly (in real time).
- Variety: the variety refers to the different sources of information and the different ways the data is stored and obtained. For example data collected from twitter and from open data webs.

Big Data is taking a really important role for more and more companies since the beginning of this decade (although some of the biggest companies, like Google, have been dealing with it since far long ago). In the following years we are going to see how it appears in various domains, such as healthcare or research, but also for social purposes.

2.2. Big Data architecture

A Big Data architecture is essentially a set of processing tools and data pipelines which convert raw data into business insight. Of course, it goes beyond and it involves since the proper physical machines to the highest level layer of software in the cluster. [2]
As the years went by some standard architecture designs appeared:

2.2.1. Lambda Architecture

The first one was conceived as “Lambda Architecture” by Nathan Marz by the end of 2011. The Lambda Architecture is a design standard for a hybrid Big Data architecture. It consists of three layers, each one fulfilling a different and necessary purpose: [3]

- Batch layer: this layer is the one in charge of storing all the volume of data and processing the whole set cyclically.
- Serving layer: The serving layer takes the processed data from the batch layer and indexes it. This job is usually done by a non-SQL database.
- Speed layer: Because the job of the batch layer takes (by definition) a lot of time, we need an extra layer to perform a real time processing for the new income data that is being generated “on the fly” while the batch layer is processing the old data.

![Figure 1: Lambda architecture](image)

2.2.2. Kappa Architecture

Although the Lambda Architecture was the best approach between 2011 and 2013, its design was based on a statement that couldn’t last long: that stream processing tools could not handle too much throughput. As the time went by,
technologies advanced allowing to increase the weight that could be handled by the last layer (the speed layer) in the system. [3]

In 2014, Jay Kreps described a new solution to deal with the same problem the Lambda Architecture dealed with, but simplified; it received the name of “Kappa Architecture”.

This new design can be seen as a simplification of the Lambda Architecture, where the batch layer has been eliminated. Now the data is stored directly in the HDFS and sent via Kafka (or a similar tool) to the different processing queues. A single streaming processing application is needed to carry all the throughput via different functions (one for the “old” data and another for the real-time income, for instance).

![Figure 2: Kappa Architecture](image)

2.3. Software containers

Software containers are a method of system virtualization that allows you to run a full application inside a completely isolated package. This package will contain everything this application needs (such as libraries, settings, system tools…). Containers make your applications easy to deploy quickly and consistently without considering the deployment environment. [4]
Because a container encapsulates everything the application needs to run, they can be deployed independently of the environment, regardless of the operating system or hardware configurations. As one of the main goals of this project is to develop an architecture which can be deployed in different systems quickly and efficiently the use of software containers was mandatory.

Each container will run an independent process on the operative system. This allows the user to run multiple instances of the same software container creating a cluster alike environment. Because of the fast boot and terminating times, containers are also a perfect way to enable the scale of applications up and down. Remember that each container will run completely isolated from the others as if it was placed in a different node of the system.
3. TECHNOLOGIES INVOLVED

In this chapter we will describe the main technologies involved in the development of this project. Including what they serve for and why were they chosen above other similar tools.

3.1. Docker

Docker is an open source platform for software container deployment. Although it was released open source in 2013, in only one year it had exceeded every other container platform. Nowadays it’s supported by some of the largest software organizations including Google, Microsoft and Red Hat. [7]

![Docker logo](figure4.png)

*Figure 4: Docker logo*

Docker is one of the most important pillars of this project as it is the tool we are using to create and deploy our software containers. The first thing we need to do is to learn how the Docker language works and how to take advantage of all it’s capabilities.

In this project, we are going to create and use five different Docker images:

- Baseimg
- Kafka
- Zookeeper
- Flink
- Couchbase
3.2. Docker Compose

Since the Docker platform has been growing so much during the last four years, a whole ecosystem has been built around it. One of the tools generated as a result of this expansion is Docker Compose. [8]

Compose is a tool for defining and running multi-container Docker applications. Once each of the Docker images has been correctly defined in its own “Dockerfile”, Docker Compose allows us to configure all our services together in a single file named “docker-compose.yml” and then start them all at the same time with a single command: “docker-compose up”.

In this project, Compose is going to serve as a way to test that each of our Docker images is working correctly and at the same time that they can work correctly as a whole.
3.3 Kubernetes

Kubernetes is an open source project for automating deployment and management of containerized applications. It was designed by Google and later open sourced. Although its version 1.0 was released on 2015, we have to take into account that it had been developed for more than a decade by google engineers before it was released open source. This is why it has grown so much in popularity in so little time. [9]

Figure 6: Kubernetes logo

Kubernetes is going to serve us as an orchestrator. When it’s well configured, it can keep track of how much throughput is handling each container so, if needed, it can scale up or down the number of containers attending a single service.

Let’s say that our Big Data architecture is functioning nonstop. Kubernetes may find, for example, that we do not need 6 Flink taskmanagers functioning at 4 a.m. when our data income is way lower than during the day. In that case Kubernetes would automatically decrease the number of containers and adjust to the real need we have, saving resources and, therefore, money.

Of course it would also be useful in the other way round. If we experience an unpredicted high demand at some point, it will scale automatically the number of containers to fulfill our need.
3.4. Apache Kafka

Apache Kafka is an open source project for handling real-time data feeds. It was originally developed by LinkedIn and then donated to the Apache project in 2011. It basically provides a “publisher/subscriber” architecture for streaming data delivery and it connects to external systems enabling the user to process this data with almost every tool available. [10]

Some of the biggest companies that rely on data streaming support Apache Kafka, like Spotify or Netflix.

3.5. Apache Zookeeper

Apache Zookeeper is an open source project from the Apache Software Foundation. It is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. Zookeeper was created to provide highly reliable coordination in distributed applications of any kind. [11]
3.6. Apache Flink

Apache Flink is an open source framework for distributed stream processing from the Apache Software Foundation. Flink provides a high-throughput, low-latency streaming engine written in Java and Scala. It also provides connectors to many other tools such as Kafka. [12]

Unlike its greater competitor Spark, which is developed for Batch processing and has adapted to stream using what they call “microbatch”, Flink was developed from the core for stream processing. This makes the difference when your system requires your data to be processed as soon as it arrives.

In our architecture, Apache Flink has a great demanding role as the unique processing engine we are counting on. Therefore, our flink containers need to be capable of processing every single piece of data that we receive as income.
3.7. Couchbase

Couchbase Server is a distributed, open source NoSQL document-oriented database. It was developed to offer great performance when dealing with large quantities of data and it provides a query engine for SQL-alike queries. [13]

For the development of this project, the reason I have decided to select couchbase among other solutions such as MongoDB is because how well prepared it is for scaling, which is the most important thing I was looking for in this project.
One of the main characteristics of Couchbase Server is how it works with a single type of node. This is what makes this application so convenient for this kind of project.
4. ARCHITECTURE DEVELOPMENT

The main purpose of this project was to develop, as a proof of concept, a Kappa Architecture with the tools described in the previous chapter. We used Kafka to create the data queues and carry them to Flink, our streaming processing service. Finally the processed data is stored in couchbase.

The project has been developed in 3 stages: designing the Docker images, testing them together with Docker compose and finally implementing the Kubernetes’ pods.
4.1. Designing the Docker images

A Docker image is defined by a text file called Dockerfile. This definition must include every step needed to obtain a reliable, self-contained and well packaged application. [14]

Notice that in other case it would have been easier and more practical to use predefined images for the applications we want to use. In fact, some of the applications have even an official docker image built by the team in charge of it but, since this project was developed mostly with learning purposes, we found quite more interesting to create our own images.

The docker images described below can be found online in my own repository at Docker Hub. [15]

4.1.1. Baseimg

The first step to follow was to create a solid base image that could be used as a floor, above which every other application image was going to be built: To define this “baseimg” and keep the weight of all the images low, we used a shrunken version of a Debian image and installed Java 1.8 above it. This Debian image is “bitnami/minideb”. [16]

Every Docker image following is based on this image.

4.1.2. Kafka

To create a Kafka image, we first needed to install Apache Kafka above the “baseimg” and then add three scripts for making it easier and faster to operate with. [17]
4.1.3. Zookeeper

As we did with the Kafka image, to define a correct image for Zookeeper it was necessary to download and install the application, set some config values, expose the ports we need and finally add one script to start the server easier. [18]

4.1.4. Flink

Making the Flink image was less complicated than expected at first. Contrary to what I was trying to do, only one Docker image is needed to serve as both Jobmanager and Taskmanager, so you need to specify while deploying the service whether you are deploying it as any of both. In the image definition, we download and install Flink and expose its ports. [19]

4.1.5. Couchbase

To make the Dockerfile of Couchbase took more time because it needs some little configurations to be taken care of to work correctly. Like with the other images, we downloaded and installed Couchbase Server, made some configuration arrangements and expose the needed ports. [20] [21]
4.2. Integration tests with Docker Compose

A top priority for me while I was planning how to approach the development of this project was to find a way of testing my Docker Images before I started developing the Kubernetes pods. I knew that I could save a lot of time preventing to start over and over again with the “translation process” from the docker images to the deployment in Kubernetes. I found Docker Compose shortly after and I couldn’t believe how well it fit with what I was searching for.

After creating all the docker images and after trying a few times with some tutorials I could easily build my own “docker-compose.yml” files and start testing how the images worked out together.

```
kafka:
    image: kafka
    container_name: kafka
    hostname: 172.25.0.4
    ports:
        - "9092:9092"
    environment:
        KAFKA_ADVERTISED_HOST_NAME: 172.25.0.4
        KAFKA_ADVERTISED_PORT: 9092
        KAFKA_ZOOKEEPER_CONNECT: 172.25.0.3:2181
    volumes:
        - /var/run/docker.sock:/var/run/docker.sock
    networks:
        redVirtual:
            ipv4_address: 172.25.0.4
    depends_on:
        - zookeeper

Figure 13: Example of a Compose service definition
To prove if the architecture was really working, testing Flink-jobs were developed to make sure every single piece of the system was doing its part. For this jobs, I took as input data JSON files with outdated information about number of clicks on usa.gov webpages. [22]

To simulate the input was a stream of real-time generated data I added a short delay in the Kafka sending loop. This way I could finally confirm that the system was working correctly in stage 2.

4.3. Kubernetes implementation

Once the docker images were completely tested with Docker Compose in the second stage of the project, it was time to begin with the deployment of the images in the Kubernetes environment.

At this point a really important matter was to know how to “translate” from Docker language into Kubernetes language. One of the main difficulties in this process of translation is the difference between a Docker Image and a Kubernetes Pod. While they are both treated as “the basic unit” in each ecosystem, the truth is a pod can be formed by one or more Docker Images.

![Figure 14: Pod example][23]
After thinking about this differences, I decided to translate each Service to a Pod (having more than one Container per pod), and so each service needed to be described again with a new '*.yaml' file [24].

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  creationTimestamp: null
labels:
  io.kompose.service: flink-taskmanager
name: flink-taskmanager
spec:
  replicas: 1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        io.kompose.service: flink-taskmanager
    spec:
      containers:
      - args:
        - taskmanager
        env:
        - name: JOB_MANAGER_RPC_ADDRESS
          value: 172.25.0.5
        image: lballestin/blink:v1
        name: flink-taskmanager
        resources: {}
        restartPolicy: Always
      status: {}
```

*Figure 15: Kubernetes service definition example*
4.3.1. Horizontal Pod autoscaling

The main difficulty of this project was not to develop a Big Data architecture, but to develop an autoscaling one. And that is exactly what the horizontal pod autoscaling function of kubernetes is in charge of doing.

This function has to be set with the kubernetes command line interface (kubectl) and is implemented as a loop. In each loop, the controller seeks for the resources that are being used by the service and compares it with the max/min resources utilization which was set in the controller definition. If the numbers are off limits, it will automatically scale up or down to keep between this fixed numbers. [25]
5. CONCLUSIONS AND RESULTS

In this chapter I will compare the results obtained with the goals expected and I will express my final conclusions concerning this project.

The result obtained in this project was a fully functional and autoscaling Big Data “Kappa” architecture. Since this was the main goal of this project, I can proudly say that I did in fact fulfill the goals I could expect to cover taking into account the duration of the project and my limited resources.

That said, I would have liked to test the project in more realistic conditions, since my laptop obviously can’t be considered a “production alike environment”. This is why I consider that this project could receive lots of improvements with some further future work.

With all this in mind, I have to say that I have enjoyed a lot the challenges this project has supposed to me. Almost every step further in the development of the project supposed learning something completely new to me and that was exactly what I was looking for when I started studying Computer Science.
6. BIBLIOGRAPHY AND REFERENCES


Este documento está firmado por

<table>
<thead>
<tr>
<th>Firmante</th>
<th>CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fecha/Hora</td>
<td>Fri Jul 07 23:03:41 CEST 2017</td>
</tr>
<tr>
<td>Emisor del</td>
<td>EMAILADDRESS=<a href="mailto:camanager@fi.upm.es">camanager@fi.upm.es</a>, CN=CA Facultad de Informatica,</td>
</tr>
<tr>
<td>Certificado</td>
<td>O=Facultad de Informatica - UPM, C=ES</td>
</tr>
<tr>
<td>Numero de Serie</td>
<td>630</td>
</tr>
<tr>
<td>Metodo</td>
<td>urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.shal (Adobe Signature)</td>
</tr>
</tbody>
</table>