

BARCELONA SCHOOL OF INFORMATICS

Stream processing with Storm

Author:

Miquel SABATÉ SOLÀ

Director:

Jordi GARCIA ALMIÑANA

June 20, 2014

Contents

List of Figures	vii
Preface	ix
1 Introduction	1
1.1 The Problem	1
1.2 The Idea	2
1.2.1 Brief description	2
1.2.2 Wrapping the iCity Platform	3
1.2.3 Realtime	4
1.3 Goals and Motivations	4
1.3.1 Goals	4
1.3.2 Motivations	5
1.4 Challenges and limitations	5
1.4.1 Challenges	5
1.4.2 Limitations	6
1.5 Source Code	6
2 An overview of the Platform	9
2.1 The Big Picture	9

2.2	State of the Art	10
2.2.1	Big data	10
2.2.2	MapReduce	11
2.2.3	Storm	13
2.3	Used Technologies	15
2.3.1	Linux	15
2.3.2	Java & Scala	15
2.3.3	Storm	16
2.3.4	Cassandra	16
2.3.5	Go	17
3	Implementation of the Platform	19
3.1	An Overview of the Architecture	19
3.2	The core infrastructure	21
3.3	Air Quality Sensors	23
3.4	Barcelona Sensors Platform	25
3.5	Application programming interface (API)	28
4	The Cluster	31
4.1	Requirements	31
4.2	Pushing the limits	35
4.3	An ideal cluster	37
5	The Development of the Project	41
5.1	The Schedule	41
5.1.1	Initial planning	41
5.1.2	Changes in the midterm evaluation	44
5.1.3	Final changes	44

5.1.4	Conclusions	44
5.2	The Budget	45
5.2.1	Estimated Salary	45
5.2.2	Software	45
5.2.3	Hardware	46
5.2.4	Conclusions	47
6	Social and environmental impact	49
6.1	Environmental impact	49
6.2	Social impact	50
6.3	Laws and Regulation	50
7	Conclusions	53
Appendices		
	Appendix A Storm: A quick introduction	57
	Bibliography	63
	List of Acronyms	65

List of Figures

2.1	The architecture of the platform	10
2.2	The MapReduce algorithm visualized.	12
2.3	Storm topology example	14
3.1	Architecture of the platform	20
3.2	AQS Topology	24
3.3	BSP Topology	26
3.4	The life cycle of a Request	29
4.1	Running top for the API	32
4.2	Running top for Cassandra	32
4.3	Disk usage of Cassandra	33
4.4	Running top for the Storm application	34
4.5	Running top in the benchmark	36
4.6	Load distribution across CPU cores	37
5.1	Gantt chart	43

LIST OF FIGURES

A.1	The topology of the Storm example	57
A.2	The code of the RandomSentenceSpout	58
A.3	The code of the SplitSentence bolt	59
A.4	The code of the WordCount bolt	59
A.5	The code of the main function	60
A.6	The output from the Storm example	61

Preface

We are living exciting times. Technology and science are moving forward in a very fast pace and societies around the globe are benefiting greatly from this situation. These technologies have improved the way we, as a society, perceive the world around us.

At the same time, it is clear that the flow of information has increased over the past few years. This leaves us a very important opportunity: we can use this information to obtain new ways to improve our societies.

This project is quite humble and it does not try to change societies. Rather, it is focuses on cities. This project consists on building a platform that operates on all the information generated by cities in order to obtain useful data. This data can then be used to improve the quality of life of the citizens of the selected cities.

Back in January of 2014 I had no idea about the topic of my Bachelor Degree Thesis. I didn't know where to go, how the whole process worked, etc. Luckily, my director Jordi presented to me this topic and I was thrilled. I saw lots of opportunities and room for investigation.

My personal goal for this project is to explore the technologies surrounding *stream processing*, and how these technologies can then be applied to the concept of *Smart cities*. I am satisfied with the results and I think that this project has fulfilled my personal goals.

Therefore, I want to thank my director Jordi Garcia Almiñana for presenting me this topic and for giving me the opportunity to work on it. It has been a great experience and I hope the reader can feel this satisfaction of mine when reading this memoir. Here it is. I hope you enjoy it.

Chapter 1

Introduction

1.1 The Problem

As mankind moves forward, new technologies appear and, with them, new opportunities. This situation has been beneficial in any sector imaginable, especially in the cities.

Cities around the globe have looked at new technologies as a way to improve the quality of life of their citizens. First and foremost, cities that are willing to improve themselves have to answer questions such as: what do the citizens need ? What can be improved ? Is this city efficient enough with the current resources ?

A way to answer to these questions is simply to create a net of sensors that will generate data. Therefore, these cities have ended up having a huge amount of sensors scattered all around their streets. This results in cities having to handle an unreasonable amount of *raw data*.

And not only that: the kind of the fetched data can vary greatly. Therefore, these cities end up having a *set* of big flows of data. This chaotic situation can be a real headache for cities willing to innovate. Thus, we have two main problems:

1. The large amount of realtime data to be processed.
2. The variety of data types to be handled.

Luckily, the European Union (EU) has acknowledged the second problem. In order

to fix this second problem for any european city, the EU has built the **iCity** Platform¹. This platform builds a basic infrastructure and a set of rules in which any european city can integrate their services into this joint effort. Right now four cities have integrated into this platform: Barcelona, London, Genoa and Bologna. Even if this is a rather small list of european cities, more are coming: Rome, Helsinki, etc. For this reason, we can be sure that this project is alive and ready to take all over Europe. Because of this, we can perfectly say that our second problem is currently being addressed and it is just a matter of time to get it fixed.

This is great news, but it does not really fix our first problem: cities still have to figure out how to compute all this data being fetched on realtime and transform it into valuable information. A valid solution to this problem has to address the following points:

1. It has to be compatible with the iCity API for practical reasons.
2. It has to process data in realtime. The vast majority of sensors generate data on realtime, so it is important to get this information also in realtime.
3. It has to be open source. This way cities can rely on this platform without getting tied into any private company. Furthermore, this would expand the possibilities for this platform for any given city.

1.2 The Idea

1.2.1 Brief description

My Bachelor Degree Thesis consists on building a platform that addresses the first problem described in section 1.1. In particular, this platform is able to:

1. Fetch and process data in realtime from any given city.
2. Provide an easy way to extend it.
3. Wrap the iCity platform, providing rich services instead of raw data.

Thanks to the iCity platform, this platform is already able to respond to a wide variety of data types. Some examples being:

- Air pollution.

¹<http://icityproject.eu/>

- Traffic.
- Irrigation control.
- Pedestrian flow.

At first I thought that I could also use data from the *OpenDataBCN*² initiative. However, I later discovered that this initiative only provides static content, so I had to drop this idea.

Moreover, this platform has been designed to be as modular and agnostic as possible. One of the consequences of this is that we could ideally integrate more platforms (apart from the already existing iCity platform) without too much trouble. This is not something that I have deeply researched, but it should be doable.

1.2.2 Wrapping the iCity Platform

One of the main points of this project is that I am going to wrap the API of the iCity Platform with endpoints of my own that will provide rich information instead of raw data. This is really important because:

1. As a platform, we do not have to worry about the **wide variety** of data types, because the iCity platform is already abstracting away this issue.
2. It gives more freedom to this **platform**. As I pointed out in section 1.2.1, this platform is not hardly tied to the iCity platform. Therefore, even in the unlikely event that iCity gets deprecated or dies, this platform can still fetch the data from somewhere else without too much trouble. Of course, this other platform has to have the same guarantees as iCity. On the other hand, if iCity is up and running, we could even consider adding more sources of data to this platform without too much trouble.
3. It gives the **end developer** more freedom. This platform does not replace iCity in any way. Therefore, developers can target the iCity platform and this platform at the same time if they really want to. However, using this platform should be enough.
4. It is **reliable**. The iCity platform is backed by the EU. This means that we can feel safe when using this platform.

²<http://opendata.bcn.cat/opendata/en/>

1.2.3 Realtime

This is the core concept behind this project. The real deal here is that all the fetched data has to be processed in realtime. This requirement comes from the way that the majority of targeted sensors work. Let's consider that we want to track the levels of pollution of the air of our city. The levels of pollution might vary during the day, and we might want to study how are these variations occurring and how can we decrease the levels of air pollution from this observation. Therefore, for this case we need to be tracking the levels of pollution through the entire day. In this simple case, we realize that processing this data in realtime is the only reliable solution to this.

In order to address this major need I have chosen the Storm framework. This framework is the very base of this platform and it is thanks to this framework that this platform can be even considered in the first place. I do not want to get into many details in this section, but if you want to read more about why I chose Storm, you might want to read the section 2.2 of this memoir.

1.3 Goals and Motivations

1.3.1 Goals

Before going any further, it is important to write down the goals of this project. It is important because this way:

1. We can determine if the project has been successful.
2. We can focus on just these goals and not get distracted by other topics.

The **goal** of this project is to build a base platform that is able to generate rich information about a set of cities in real time.

This goal can then be decomposed into three subgoals.

- Design the **base platform**. That is, to build the basic infrastructure in which the rest of the code will be based on.
- Design a couple of useful **services** showing the potential of this platform.
- Give some hints on the design of an ideal **cluster** that can run this platform.

The first two points will be explained throughout the sections 2 and 3. These two sections explain how I have designed and implemented the platform. After reading these two sections it will be clear that these two points have been successfully completed. The last point will be explained in the section 4 of this memoir.

1.3.2 Motivations

I think it is also important to write down my motivations for doing this project. The following are my reasons for doing this:

- **Research.** Because of my Thesis, I will be doing a lot of research. My main focus of research has been: Big data, streaming processing and smart cities.
- **Learn** some technologies as a side effect. My base technology has been the Storm framework, which is done in Java. The platform that I have done that lies on top of it has been done in Scala. Therefore, I have improved my skills in the Java and the Scala programming languages.
- **Provide** a useful technology base. Even if this platform has been designed to solve a specific problem, I am confident that it can be useful for other people interested in the technology. Moreover, this platform can potentially help european cities, specifically Barcelona in my case. And more than that, I am really optimistic about this platform because I believe that it can be the base for other thesis of other students.
- **Practice.** The Bachelor Degree Thesis is the final stop of this journey that I've been doing in the Barcelona School of Informatics. I've learnt a lot and I have applied some of the concepts learnt in this university to this Thesis.

1.4 Challenges and limitations

1.4.1 Challenges

When any person starts a project, he has to have clear that, regardless of the topic, any project will come with its challenges. This has been true for my Thesis, and I think it is important to describe the challenges that I have faced.

It has passed some months now since I started this project. When I started I hoped to obtain all the information also from the Barcelona OpenData initiative. However, this initiative turned out to be not what I expected: it is just a repository of static data. Since my platform requires dynamic and realtime data, fetching static data is pointless.

This situation did not affect this project greatly, since the iCity platform turned out to be exactly what I needed, and, therefore, I did not care about Barcelona's OpenData any longer.

Another challenge has been that I am doing this project with the Scala programming language. To be honest, I could have done this project with just Java, since Storm is available to any programming language that runs on the Java Virtual Machine (JVM). Scala, however, is more modern than Java, and I have had more fun writing Scala code than Java. So, even if I am new to Scala and it is a challenge, it is a challenge in which I am fully motivated.

Hardware has proven to be a challenge too. I have never done benchmarks regarding different computers, or given any specifications on the hardware that a program of mine requires. So it has been definitely a new experience, and of course a challenge.

Last but not least, a final challenge has been this report itself. First of all, it has been a challenge because I have never had to write down a report like this: extense, technical and detailed. Finally, it has been a challenge because of the language. When you are writing a document like this in a language other than your mother tongue, it can be tricky.

1.4.2 Limitations

In this sections I want to state the limitations of this project. This is useful to clarify some points.

This project does not intend to cover as much as possible in regards to the technologies being used. This is not a project about Storm or any other technology in particular that I might have used. Instead, this is a project focused on solving a specific problem.

When I started this project I even considered the idea of *designing* my own set of sensors. This is of course unrealistic, so I sticked with the iCity platform. As I have stated previously, I could have added support for more platforms than iCity, but it would have been impossible for time constraints. Anyways, I limited this project to be *iCity-specific*. This does not mean that this platform cannot run with other platforms.

1.5 Source Code

This project is open source. Therefore it can be retrieved, modified, read, etc. by anyone that is interested in it. I have developed this project with an open source approach

because:

1. I believe that any project developed under an **academic** course should be open source. In the end, one of the main goals of a thesis is to learn. If this project can help other students or teachers to have a deeper understanding of the topics explained here, the better.
2. It can be created a **business model** around this platform that focusses on the services it can provide, instead of selling licenses. I have not focussed on business models in my thesis, but I can see multiple ways to monetize this platform. In these models the fact that the platform is open source does not make any harm to them.
3. The code for this platform can be **useful** for other people with similar projects. Even if it seems an altruistic way of thinking (and it is), there are also selfish reasons to do this: if other people use this, it is likely that they will contribute back, thus making the platform better. Therefore, I believe that making this platform open source will increase the chances for it to be as successful as possible.
4. It is the **fair** thing to do. This is more a personal believe, but to me it is important. I have learnt a lot with open source software, and that is the main reason why I think it is the fairest thing to do: so others can have the same opportunities that I had.

The license for this project is the GPL version 3. You will find a copy of this license in the root directory of the project. In order to retrieve the project, you will need **Git**³. Git is the version control system that I have used to develop this project. Now, with git installed you have to execute the following command:

```
git clone https://github.com/mssola/thesis
```

With this command you will download the project in the current directory. It is extremely recommended to read the “README.md” file before going any further. In this document I have carefully explained have a developer can:

- Download and install all the dependencies.
- Build the project.
- Run the platform.

There are also “README.md” files inside some of the subdirectories. This files contain more information for the given subdirectory. It is also adviceable to read the contents of these files in order to have a better understanding of the project.

³<http://git-scm.com>

Chapter 2

An overview of the Platform

2.1 The Big Picture

In this section I am going to explain the architecture of the platform. It does not enter into many details, but it is helpful in order to get a quick overview on the design of the platform.

1. The cities of London, Barcelona, Genoa and Bologna are **producers**. This list of cities will expand over time. Users are **consumers**. They can consume from either the iCity API, from the API from my platform, or from both.
2. The **iCity** platform offers a common API that integrates all the cities. It gets the data from the different cities and allows the access of this data to any user registered in the iCity platform.
3. My **platform** acts as a user of the iCity API and users can access the processed data.
4. This platform is formed by **several** services. These services consume the data from the iCity platform in order to produce the requested information.
5. All the services from this platform are available through an **API**.
6. All the services are part of a **Storm** topology.
7. All the services share the **state** of the platform, and it is stored in a Cassandra instance.

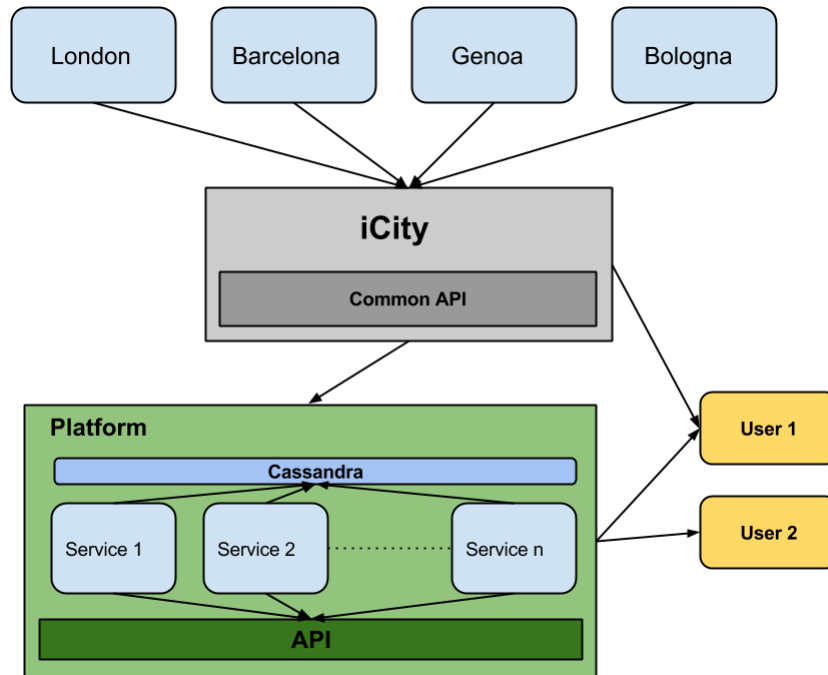


Figure 2.1: The architecture of the platform

2.2 State of the Art

2.2.1 Big data

As I have pointed out in the section 1.1, this platform has to handle a huge amount of raw data. This concept is known as **Big data**. There is no specific definition of Big data. However, we can see Big data as a term that applies to any collection of data sets so large and complex that it becomes difficult to manipulate this data with traditional approaches.

This term is relatively new, and it brings with it a set of challenges that are not trivially solvable. This concept has been applied to a wide variety of fields. Some examples:

- **Science:** the Large Hadron Collider (LHC)¹ has a set of 150 sensors that deliver data 40 times per second each. After applying some filters, we have around 100 collisions of interest per second. This means that the LHC has to handle 500 exabytes of data every day.
- **Government:** it is known that the Obama administration has used supercomputers

¹http://en.wikipedia.org/wiki/Large_Hadron_Collider

and Big data techniques to address important problems faced by the government².

- **Private sector:** companies such as Google, Facebook, Twitter, Amazon, etc. use big data technologies to handle the large quantity of data that their servers receive.

This idea of Big data has revolutionized technology because it is a very important problem that needs to be fixed. To fix this, we can do two things:

1. Add more **hardware**. This is the simplest solution: if one server cannot handle the flow of data, let us add more servers. The problem of this is that it might not always work and that it will make the infrastructure more expensive to maintain.
2. Improve the **algorithms** behind the handling of data flows. This is what an engineer should always do before blindly adding new servers. Algorithms will usually apply to how the tasks are distributed across the infrastructure and how this data is then treated.

The first solution is really simple, and it is something that we can always do. Therefore, this solution is not really interesting from an engineering stand point. What is really interesting, though, is the second solution.

2.2.2 MapReduce

The first algorithm to tackle the Big data problem was the **MapReduce** algorithm. This algorithm was introduced by Google in the paper “MapReduce: Simplified Data Processing on Large Clusters”[1].

This is a two-steps, parallel, distributed algorithm that works on *clusters* and *grids*. The data can be stored either in a filesystem or in a database. The two steps are as follows:

1. **Map:** The master node takes the input, divides it into smaller problems and distributes them to worker nodes. A worker node may do this again in turn.
2. **Reduce:** The master node collects all the answers to all the sub-problems and combines them in some way to form the output.

This algorithm can be visualized as follows:

²<http://www.whitehouse.gov/blog/2012/03/29/big-data-big-deal>

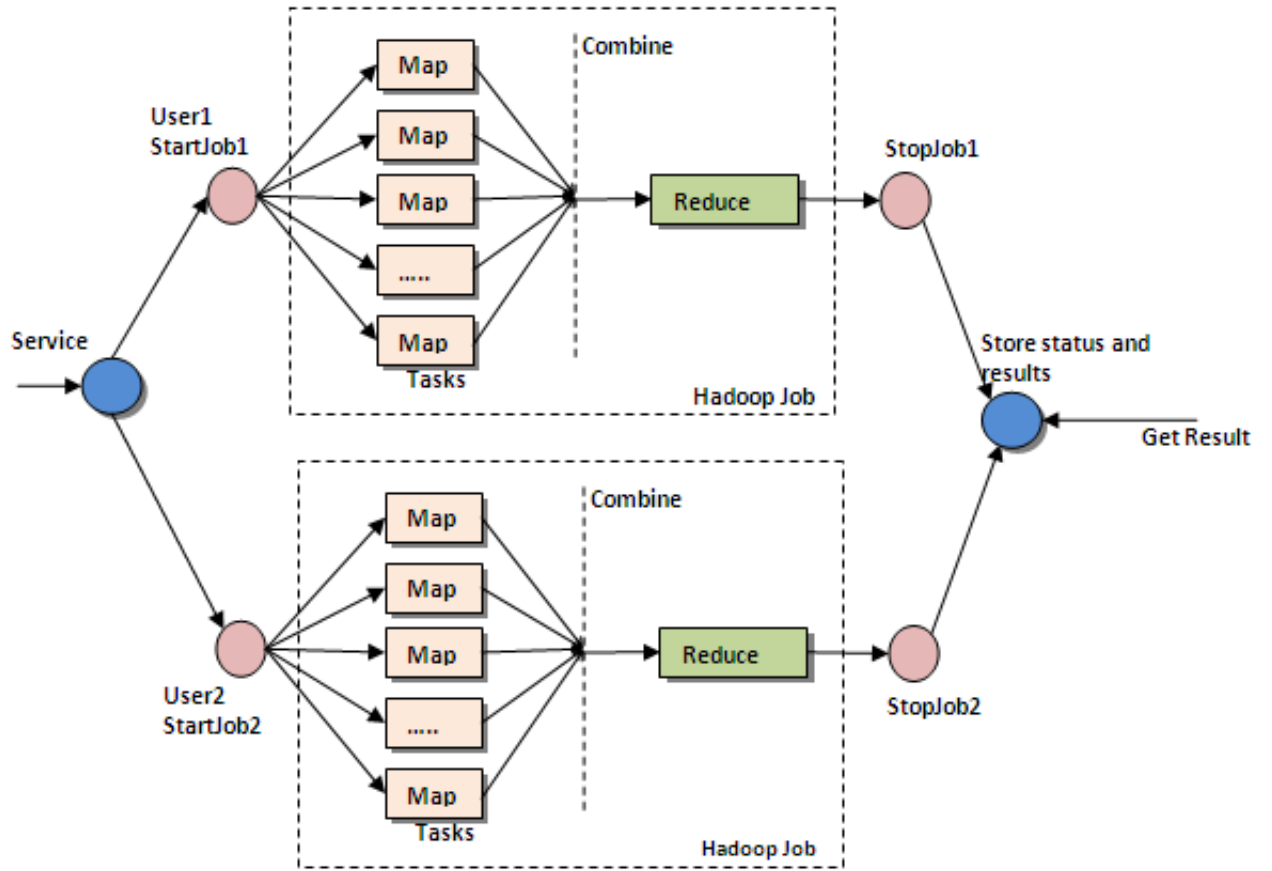


Figure 2.2: The MapReduce algorithm visualized.
 source by: <http://map-reduce.wikispaces.asu.edu/>

In the figure 2.2, two users start a Hadoop job. As we can see, a job consists of the map and reduce steps.

Many programs and frameworks that implement the MapReduce algorithm have appeared since the initial announcement from Google. For example, some well known programs implementing in some way or another the MapReduce algorithm are: Hadoop³, Cassandra⁴ and MongoDB⁵.

³<http://hadoop.apache.org/>

⁴<http://cassandra.apache.org/>

⁵<http://www.mongodb.org/>

2.2.3 Storm

Hadoop is the most notable implementation of the classic MapReduce algorithm. It guarantees reliability and scalability. Moreover, Hadoop is really fast and efficient, and because its popularity some big projectes like Apache Hive⁶ or Apache Spark⁷ are closely related to it.

However, Hadoop has one major drawback: it uses **batch processing**. This is not a flaw or a weakness, it is like this by design. Hadoop is designed to process data through batches: handling large amount of data in a single round. But sometimes the data to be processed is not available yet. This happens when you receive a lot of data that has to be processed in realtime: you do not have the data available right now, but you do not want to setup a batch for all the packets of data that might come in.

Some people have tried to hack Hadoop to workaround these issues. However, all the attempts have failed so far because:

- It is **tedious**. People have to implement message queues and protocols around them.
- There is no real **fault tolerance**. Complex systems have to be developed in order to handle fault tolerance gracefully. All the attempts on this have failed horribly.
- It does not **scale**. When you add or remove more servers, you have to manually do the changes. This happens quite frequently, so it is a pain that the developer has to usually face.

As we have seen, there is no hack that will turn Hadoop into a realtime system; realtime data processing has a fundamentally different set of requirements than batch processing. Therefore we need some sort of “Hadoop of realtime”. Two projects aspired to get this “title”:

1. Yahoo! **S4**[3]. One of the first implementations of this idea. It is now under the umbrella of the Apache foundation. However, as of the writing of this document, this project seems to have died: the development has stopped since a couple of years ago, lack of documentation, etc.
2. Twitter **Storm**. It is the natural competitor of S4, and the one that is still alive and in production in companies such as Twitter. It is now under the umbrella of the Apache foundation just like S4.

⁶<https://hive.apache.org/>

⁷<https://spark.apache.org/>

I picked Storm for this project since there is no real alternative to it and quite honestly Storm is really good at it. Storm has the following main points:

- It **scales**. Companies such as Twitter or Zookeeper have Storm in production and they get nice results. As an example, an initial Storm application on a cluster of 10 nodes can handle 1,000,000 messages per second.
- It guarantees **no data loss**.
- It is **fault-tolerant**. If some task fails, it will be re-assigned later on the execution of the application.

Each Storm application has to define a **topology**. This is accomplished with the idea of **spouts** and **bolts**. Spouts and bolts are the way to specify how data streams have to be consumed. Spouts are used to fetch the data that has to be processed and bolts consume this data in order to get results. There can be several spouts and bolts in a topology. Bolts get the data from spouts or from other bolts. The following picture shows how Storm understands the notion of topology:

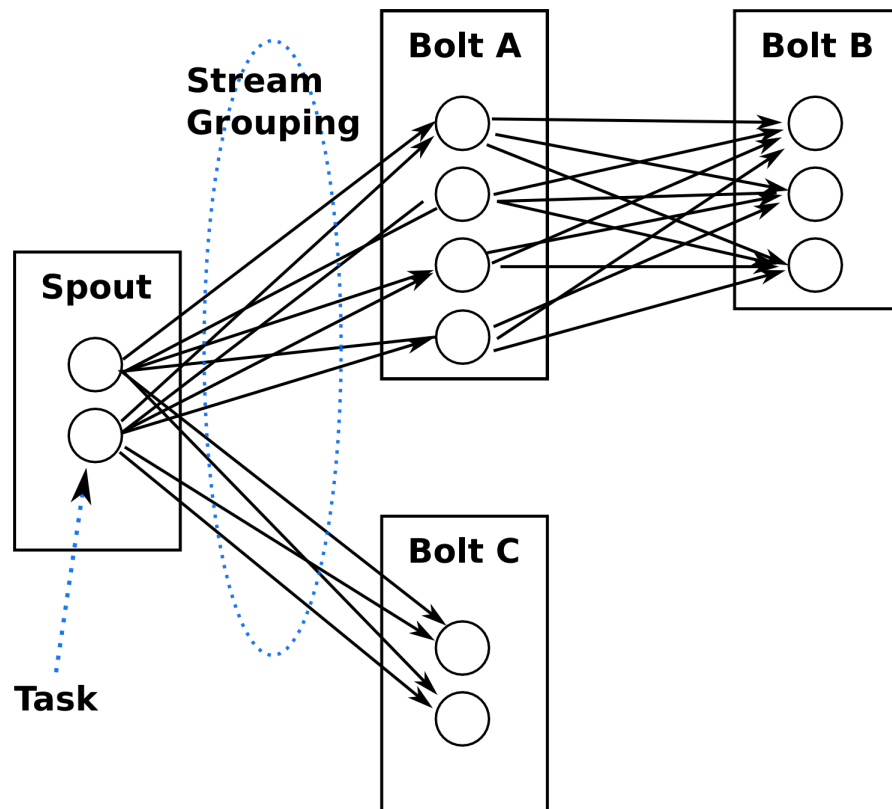


Figure 2.3: Storm topology example

source by: <https://storm.incubator.apache.org/documentation/Tutorial.html>

2.3 Used Technologies

2.3.1 Linux

This platform targets Linux⁸. It is the Operating System that I have used to develop this platform and the target of this platform. This is because, for a variety of reasons, the vast majority of Internet servers use Linux as their Operating system. Therefore, it makes perfect sense to use Linux as the target operating system.

Note that this platform should also work in MacOS X and other BSD variants. However, I have not tested this platform in these operating systems, so I cannot claim anything in regards to availability here.

2.3.2 Java & Scala

Storm is implemented with Java and Clojure⁹. These two languages sit in top of the JVM. The JVM is a virtual machine that can be used by any language that knows how to produce bytecode for it. This includes languages such as: Java, Clojure, Scala, Groovy, etc.

Moreover, all the languages on top of the JVM can share packages. That is, a JAR file can be used as a library from any of these languages, regardless of the language that was used in the library inside the JAR file. This means that languages like Scala can re-use the huge list of Java packages that the community has implemented.

This means that in order to implement this platform I could have used any language that sits on top of the JVM without any real problem. I have chosen Scala[4] because:

- It is as **fast** as Java, so there is no performance penalties because of the language if we compare it with Java.
- It is **modern**. Scala is a more recent language. This means that it has had the influence of languages that are more recent than Java. This results in Scala having many concepts from functional programming languages, concepts from Python, Ruby, etc. It is an absolute pleasure to write Scala code.
- It has a **robust** approach of concurrency. It is far more intuitive than Java's lock/unlock mechanisms.

⁸<http://www.linux.com/>

⁹<http://clojure.org/>

- It is **stable**. Even if it is more recent than Java, Scala is rock solid. As an example, big companies like Twitter and Foursquare have lots of Scala code running on production.

2.3.3 Storm

As I have said in the section 2.2.3, the technology that I am using to process all the data is Storm.

2.3.4 Cassandra

I do not store a lot of data in this platform, but I do store some data like the state of the cluster. To achieve this I use Apache Cassandra[2]. We could have chosen any DataBase Management System (DBMS) here to do the job, but I have chosen Cassandra for the following reasons:

- It has support for MapReduce. This is the main reason that I have chosen Cassandra instead any other traditional DBMS like PostgreSQL.
- It is **fault-tolerant**, and supports several replication policies across multiple clusters.
- It is **descentralized**. There is no single point of failure.
- Its NoSQL nature has been helpful throughout the development of this platform. Moreover, its query language (CQL) is quite similar to standard SQL, so there has not been any major learning curve for me to use Cassandra.

One could argue that we do not need any DBMS at all: we could just write into temporary files or something like that. Even though would have simplified things, this is not possible for the following reasons:

1. This platform can run in a multi-node cluster. Therefore, we would have to pick a node among the others to store these temporary files. Doing this is, of course, undesirable.
2. Even if we pick a node from the cluster to keep these files: what if this node melts down ? We cannot risk the integrity of the entire cluster to a node that stores temporary files.
3. CQL is a nice SQL-like language that gives us more flexibility doing some operations on files.

2.3.5 Go

Go¹⁰ is a programming language that emphasizes concurrent programming. It was first developed by Google but it is now a truly open source project. I use Go in the API layer. I have chosen Go for the following reasons:

1. Go puts special emphasis on **concurrency**. Concurrency is also a big deal in the Storm application. Therefore, it is a perfect match to use a language that deals with concurrency and parallelism in an elegant and powerful way.
2. Go is a **compiled** language. Therefore it is fast.
3. It is **simple**. As you can see in the language reference¹¹, this language is pragmatic and simple. Plus, programmers coming from languages like C and C++ will find a lot of similarities with this language.
4. It is **stable**. Even if it is a fairly recent language, it is rock solid. For example, companies like Google, Youtube, Dropbox, Soundcloud, etc. are already using Go in production.
5. The **net/http** package is simple but really powerful. It abstracts away a lot of problems regarding HTTP requests and responses.

¹⁰<http://golang.org/>

¹¹<http://golang.org/ref/spec>

Chapter 3

Implementation of the Platform

3.1 An Overview of the Architecture

In this chapter I am going to enter into the details of the implementation of the platform. The platform consists of three main pieces:

- The **Storm** application. This application is responsible of all the computations being made. It is the main piece of the platform and it will be thoroughly explained in the sections 3.2, 3.3 and 3.4.
- The **Cassandra** instance. This instance is shared among all the other pieces and stores the state of the platform. We will enter in more detail later on this chapter about what is the state of the platform.
- The **API** layer. This is a very thin layer that is placed between the Storm application and the end developer. The API is explained in section 3.5.

The architecture itself is a bit complicated, since we want to provide a realtime API. Nonetheless, one can have a general idea of the architecture with the figure 3.1

iCity

As you can see in the figure 3.1, the iCity platform is represented as a cloud. This is because we do not really care about the internal infrastructure of the iCity platform. The only thing we care about iCity, from a developer perspective, is that it has an API that integrates all its resources.

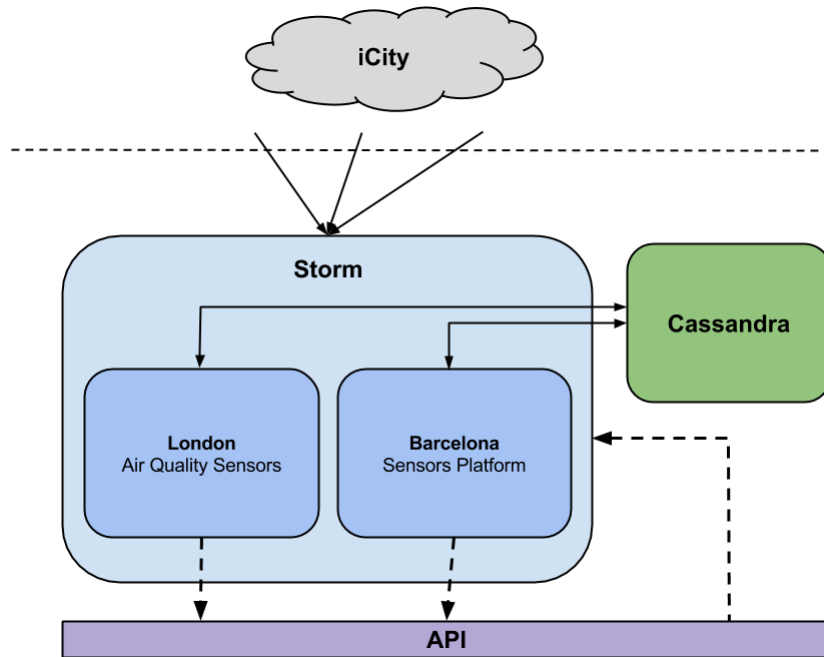


Figure 3.1: Architecture of the platform

Therefore, our point of view is: we receive data from a platform called iCity, and we do not want to know what is going on inside this platform. Of course, since this platform is totally independent to the source (iCity), we could ideally add more “clouds” without too many problems. This change would require a few changes in the code of the Storm application.

Cassandra

Cassandra is a NoSQL Database Management System that works wonderfully in clusters. For more information about Cassandra, you might want to read section 2.3.4. The main point of Cassandra is that it is designed to work on clusters and that it can be fully integrated in a Storm application.

Now, why would we want a Database Management System? The answer is that we need to store the **state** of the application. By state we mean some data that is relevant to the whole application. An example of the data that we store in Cassandra are devices. Each city has a set of devices, and these devices have a set of properties. The iCity API expect the developer to request information by pairs of deviceID-property. Therefore, we need to store the devices for each city and the properties that we can request from them. In the future we could add statistics about the execution of the application for debugging

purposes.

The point is that Cassandra is an important piece of the puzzle but the data stored will not grow too much since the state of the application does not require a lot of data.

Storm and the API

As you can see in figure 3.1, there are dashed arrows between the Storm application and the API. These arrows are drawn this way because they are TCP sockets between Storm and the API.

A Storm application consists of a topology of spouts and bolts. The spouts fetch the data continuously. In our case this data comes from the iCity platform. The output of the Storm application is given by the bolts in the end of the topology. This output is given continuously, because of the nature of the Storm application. Therefore, we need sockets to pass the results from the Storm application to the API layer.

But, how can the Storm application know the address and port of the sockets? This information is given by the API. In short, when the API has a new request, it creates a new TCP socket, and the address and port of this new socket will be passed to the Storm application. You can see this in figure 3.1 represented as the rightmost dashed arrow. You can think of this as subscriptions:

- The API opens a new socket for each request and subscribes it in the Storm application.
- The action of “subscribe” is done through a socket that the Storm application is listening to.
- It is up to the API layer to keep the streaming model. This means that the API can stream the data received from the socket to the end developer or it can decide to close the connection after a certain period of time. This means that this API is, by design, able to offer both “traditional” endpoints or streaming endpoints. We will see more about this in section 3.4.

3.2 The core infrastructure

In this section and in the following sections 3.3 and 3.4, I will be talking about the Storm application. I have named the Storm application “Snacker”. That is why all the packages name start by “snacker”.

SBT

The Storm application is built with the SBT¹ toolchain. This toolchain is the *de facto* standard for Scala software and it is similar to Java’s Maven or Ant. The configuration for SBT resides inside the “project/Build.scala” file. This file does the following:

1. It sets up basic information about this project.
2. It tells SBT which version of Scala has to be used.
3. It defines the different modules of this application.
4. It tells SBT the dependencies for each module.

SBT is the best toolchain that I have found to build Scala projects, and this choice has affected on how the code is structured.

The main function and the core package

The core infrastructure is divided into two Scala packages:

1. The package **com.mssola.snacker**. This package contains the main function and it is located in the “src” directory. This main function is responsible for:
 - Initializing the application.
 - Loading the services.
 - Executing the services.
2. The package **com.mssola.snacker.core**. This package contains a set of classes, objects and traits that will be used by the services. It is located in the “snacker-core” directory. We can think of this package as the main *library* for the services that are built on top of Storm.

The most important thing about the `com.mssola.snacker.core` package is the **BaseComponent** object. This object is expected to be subclassed by each of the services to be loaded. It is important because this object is responsible for initializing each service and setting it up.

¹<http://www.scala-sbt.org/>

3.3 Air Quality Sensors

London

London is one of the cities that are available in the iCity platform. The city of London has three services integrated in iCity:

1. **Air Quality Sensors (AQS)**. This service has sensors deployed all across the city of London. These sensors measure the air quality of the city. More specifically, these sensors send data about the levels of: NO_2 , SO, O_3 , PM10 and PM25.
2. **TFL Journey Planner**. This service gives access to developers to the TFL API.
3. **Alert ME**. This platform is a Smart Home platform.

The second and the third points are not really interesting in our case because we need a source that is public and that sends data constantly. Therefore, for this project I have picked the AQS platform.

Design and implementation

Even if the AQS platform is the most complete service from London, it has some major drawbacks:

- The **update rate** is low. Apparently the sensors from this platform send data every hour. Nothing more, nothing less. This is quite frankly, a poor update rate.
- There are lots of sensors that are not working. I have found that a lot of the deployed sensors are not working at all.

Because of my points above, I have decided that I will not provide a streaming API around this platform. Instead, I have built a minimal API around this platform that only wraps up the API to provide some extra information.

The Storm application implements support for this platform through the Scala package: **com.mssola.snacker.aqs**. This package is located in the “snacker-aqs” directory. The topology of this package is the following:

- The **AqsSpout** is the only spout for this platform. This spout fetches the data from iCity. After fetching it, this spout cleans and sends the data to the AqsBolt.

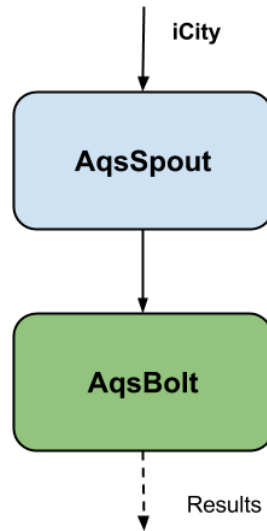


Figure 3.2: AQS Topology

- The **AqsBolt** is the only bolt for this platform. This bolt receives the data from the **AqsSpout**. After this, it performs some computations and sends the results through a socket.

As I have previously said, this platform will not have a streaming API. This means that when the API layer gets the data from the socket, it will close this socket instantly. Therefore, the lifecycle of an HTTP request follows these steps:

1. The API layer receives an HTTP request that points to the AQS service.
2. The API layer subscribes to the AQS service for this request.
3. The AQS module computes the results and sends them through the socket.
4. The API layer receives the data and unsubscribes this request from AQS.
5. The API layer generates a response with the given data.

API endpoints

The API for this service is minimal: I have implemented only one endpoint. This is because the AQS service is really simple, and there are not a lot of options here. This is the endpoint for this service:

`/aqs/{id}/interval`

The **id** parameter has two possible values: “all” or a device id. The possibility of getting the value for all the devices in a single request is huge step forward in comparison of the iCity API. Besides the “id” parameter, the developer has to include the following parameters in the request:

- The **from** and the **to** parameters. These parameters are UNIX timestamps containing the interval of time.
- The **property** parameter. Possible values are: “no2”, “so”, “o3”, “pm10” and “pm25”.

Calling this endpoint will result in a JSON response containing the values for the specified properties and devices.

3.4 Barcelona Sensors Platform

Barcelona

Barcelona is one of the cities that are available in the iCity platform. The city of Barcelona has invested heavily in the Smart Cities trend. Barcelona has opened a lot of static data in the OpenDataBCN initiative. Apart from that, Barcelona has three platforms integrated with the iCity platform:

1. The **Barcelona Sensor Platform (BSP)**. This platform allows any developer to access the data sent by sensors that are distributed around the city. This platform includes these kind of sensors:
 - Environmental sensors (temperature, NO_2 , CO_2 , noise, etc.).
 - Sustainability (level of capacity of the container waste).
 - Traffic management (parking sensors).
 - Walkers flows (number of pedestrian).
 - Irrigation control (ground humidity, wind rain, temperature).
2. **Smartcitizen Platform**. Smart Citizen is a platform to generate participatory processes of people in the cities.
3. **IRIS**: the complaints and suggestions system.

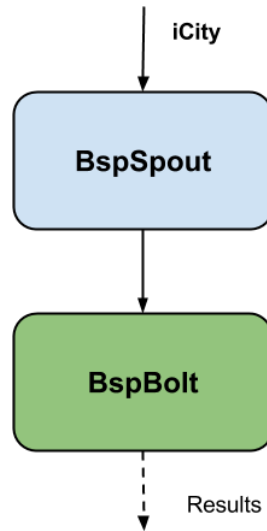


Figure 3.3: BSP Topology

From these three possibilities I have picked the BSP. This is because the Smartcitizen platform is in an experimental stage and the IRIS platform is too different to the London endpoint.

Storm

The Storm application implements the support for BSP in the Scala package: `com.mssola.snacker.b`. This package resides inside the “snacker-bsp” directory.

My original design consisted of one spout and five bolts (one for each kind of sensor). However, I later decided that this was not realistic for the following reasons:

1. The iCity API does not make a distinction between the type of data. Therefore, since there is not a real distinction, the data has to be treated equally, regardless of the type.
2. Conceptually, multiple bolts means that we are doing completely different things with the same raw data. For example, with the same data one bolt could store something on the database, and another bolt could compute something else. This is not the case.

Thus, the final design is fairly similar to the design of the London AQS service:

1. The **BspSpout** fetches the raw data from the iCity platform. After this, it cleans up the data and sends it to the BspBolt.
2. The **BspBolt** receives the data from the BspSpout and performs a set of computations. After this, it sends the results to the given socket.

Streaming API

The update rate of the sensors from the BSP is not very high, but at least it is better than London. This is why I have decided that BSP will have a streaming API. A streaming API means that a request has the following lifecycle:

1. The API layer receives an HTTP request that points to the BSP service.
2. The API layer subscribes to the BSP service for this request.
3. The BSP will continuously fetch and process data. The processed data will be sent through the socket.
4. The API layer receives data from the BSP service but does not unsubscribe the socket. That is, it will keep on receiving data indefinitely.
5. The API layer creates a response with “Transfer-Encoding: chunked”. This means that the TCP socket established between the client and the API layer will not fade away until one of both sides decides to close the connection. This will not happen from our end, so it is up to the client to close the connection.

From the client’s perspective, this means that he can “subscribe” to an endpoint and he will be getting updates in realtime without doing anything special. Let us remember that this was one of the major goals of this project: to be able to get updates of processed data on realtime.

One endpoint has been implemented for this purpose:

`/bsp/s/{id}`

The **id** parameter has two possible values: “all” or a device id. Therefore, a client has the possibility to subscribe to all the devices from Barcelona and he will be getting updates in realtime.

3.5 API

Introduction

In this section I will explain the design and implementation of the API layer. The API layer is not integrated in the Storm application. This is because the Storm application only transforms data into processed information, but it does not deal with the HTTP protocol.

Therefore, I have built a thin layer of software that deals with HTTP requests and responses. Moreover, it takes care of all the socket infrastructure. Overall, the API layer is really simple. It is so simple that it has been implemented in a single file. This file is located in the “api” directory and it is named “server.go”.

This piece of software has been written in Go. There are a lot of thing to say about the Go programming language, but I have already said them in section 2.3.5. The Go programming language is a great match for the API layer because:

1. Go puts special emphasis on **concurrency**. Concurrency is also a big deal in the Storm application. Therefore, it is a perfect match to use a language that deals with concurrency and parallelism in an elegant and powerful way.
2. Go is a **compiled** language. Therefore it is fast.
3. The **net/http** package is simple but really powerful. It abstracts away a lot of problems regarding HTTP requests and responses.

Step by step

When the client performs a request to our platform, the API layer receives it and evaluates whether this is a valid request or not. If it is a valid request, then a new **goroutine** is created. For simplicity, we can view goroutines as light threads.

This new goroutine will handle the request. Therefore, the whole layer does not block for one request, it just creates a new goroutine and keeps on listening for new requests.

This goroutine will open a new socket. This socket will be the one that will be used by the Storm application to send the results. After opening this new socket, this goroutine will send a message to the Storm application through a socket. This message contains: the address and port of the newly created socket and the parameters of the requests.

The Storm application will eventually send some output from the given socket. When this happens there are two possibilities:

1. The request was directed to the AQS platform. In this case, the goroutine will close the socket. After this, it will create and send a response that contains the processed data as given by the Storm application. When this is done, the goroutine closes the HTTP connection from the request and dies.
2. The request was directed to the BSP platform. In this case, the goroutine will not close the socket. Instead, it will be continuously receiving data from the Storm application. This data will be sent in chunks to the client. This is done by setting “chunked” to the “Transfer-Encoding” HTTP parameter in the response. This means that the communication will not stop until the client decides to stop it. When this happens, the goroutine will close the TCP connection between the goroutine and the Storm application and it will finally die.

The figure 3.4 describes the life cycle of a request. In this figure, the “3a” step represents the final step of a “standard” request. Note that the goroutine will close the connection and dies after the response has been sent. The 3b case describes a streaming request. The communication will not stop until the client says so. Therefore, the goroutine will be kept alive until the client decides to close it.

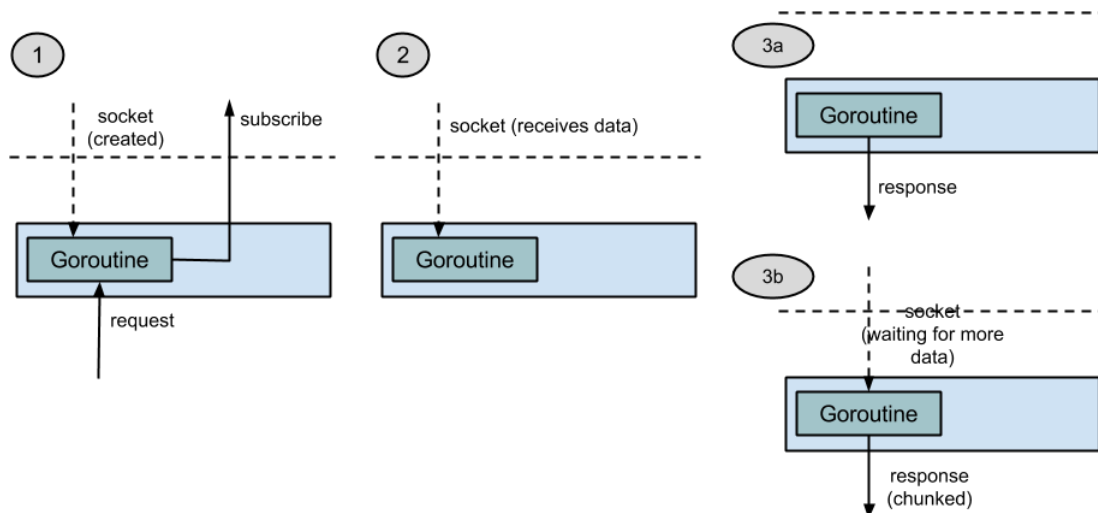


Figure 3.4: The life cycle of a Request

Chapter 4

The Cluster

4.1 Requirements

The first thing to do is to determine the requirements of the whole platform. This is needed to design a cluster that is able to successfully run this platform. In order to do this, I am measuring the requirements of each component in a normal execution. Moreover, I am also making measure on how these requirements vary depending on the situation.

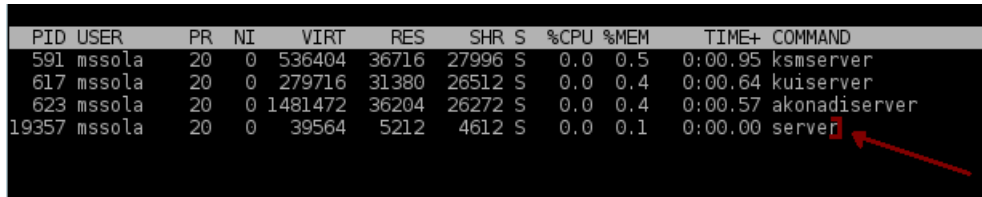
We will say that the platform is executing normally when all the services are working at full speed and the other services are all up and running.

API

The API layer is easy to track. It is a process called “server”. Therefore, to see what is this process consuming I perform the following command:

```
top -p 'pgrep server | tr "\n" "," | sed 's,$'
```

With this command I have the following results:



PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
591	mssola	20	0	536404	36716	27996	S	0.0	0.5	0:00.95	kmsserver
617	mssola	20	0	279716	31380	26512	S	0.0	0.4	0:00.64	kuiserver
623	mssola	20	0	1481472	36204	26272	S	0.0	0.4	0:00.57	akonadiserver
19357	mssola	20	0	39564	5212	4612	S	0.0	0.1	0:00.00	server

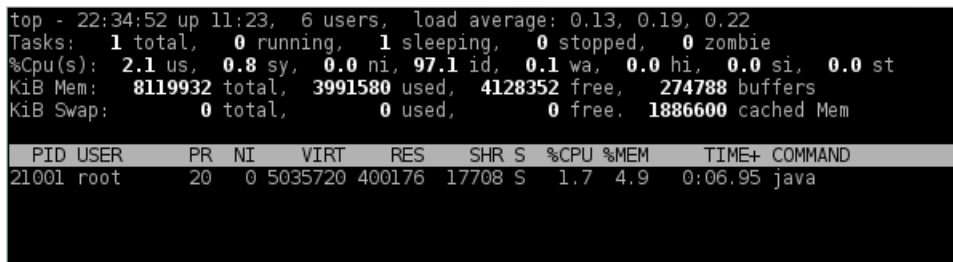
Figure 4.1: Running top for the API

Therefore, it is consuming approximately 4.6 MB of shared memory, which is almost nothing. This proves that the API layer is extremely thin. There are peaks of CPU usage when there are a lot of requests coming. However, these peaks are in the order of 2% and 3%. This is because the API layer spends most of the time waiting for requests or waiting for data from the Storm application. The peaks are most probably the creation of goroutines.

Cassandra

In my system Cassandra is a bit harder to track than the API layer. This is because I start it as a daemon with systemd. In this case, the name of the process of Cassandra is just “java”, which is not very helpful.

Anyways, if I perform the same command as in the API layer, I get the following results:



```
top - 22:34:52 up 11:23, 6 users, load average: 0.13, 0.19, 0.22
Tasks: 1 total, 0 running, 1 sleeping, 0 stopped, 0 zombie
%Cpu(s): 2.1 us, 0.8 sy, 0.0 ni, 97.1 id, 0.1 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 8119932 total, 3991580 used, 4128352 free, 274788 buffers
KiB Swap: 0 total, 0 used, 0 free, 1886600 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
21001	root	20	0	5035720	400176	17708	S	1.7	4.9	0:06.95	java

Figure 4.2: Running top for Cassandra

As you can see, Cassandra is quite heavy. In a normal execution Cassandra is consuming approximately 400 MB of memory (where 17.3 MB is shared). This is really heavy. Moreover, the CPU usage is around values of 1.5% and 2%. When Cassandra is being heavily used (for example, when initializing the platform), the CPU usage raises to values like 30%. This is not a problem because Cassandra is rarely heavily used. Therefore, Cassandra will usually have values like 1.5% during its execution.

Cassandra is used in this application to store the state of the platform. Therefore, I have to measure the disk usage too. In order to do this, I have performed the following command:

```
du -ch /var/lib/cassandra/
```

With this I get the following results:

```
mssola:~ $ du -ch /var/lib/cassandra/
commitlog/  data/          saved_caches/
mssola:~ $ du -ch /var/lib/cassandra/data/
104K  /var/lib/cassandra/data/snacker/devices/snapshots/1400678578961-devices
108K  /var/lib/cassandra/data/snacker/devices/snapshots
492K  /var/lib/cassandra/data/snacker/devices
100K  /var/lib/cassandra/data/snacker/properties/snapshots/1400678589614-properties
104K  /var/lib/cassandra/data/snacker/properties/snapshots
108K  /var/lib/cassandra/data/snacker/properties
604K  /var/lib/cassandra/data/snacker
4.0K  /var/lib/cassandra/data/system/peer_events
4.0K  /var/lib/cassandra/data/system/peers
40K   /var/lib/cassandra/data/system/schema_columnfamilies
4.0K  /var/lib/cassandra/data/system/hints
4.0K  /var/lib/cassandra/data/system/batchlog
4.0K  /var/lib/cassandra/data/system/NodeIdInfo
36K   /var/lib/cassandra/data/system/compaction_history
100K  /var/lib/cassandra/data/system/sstable_activity
4.0K  /var/lib/cassandra/data/system/schema_triggers
4.0K  /var/lib/cassandra/data/system/paxos
68K   /var/lib/cassandra/data/system/IndexInfo
108K  /var/lib/cassandra/data/system/local
4.0K  /var/lib/cassandra/data/system/compactions_in_progress
36K   /var/lib/cassandra/data/system/schema_keyspaces
44K   /var/lib/cassandra/data/system/schema_columns
4.0K  /var/lib/cassandra/data/system/range_xfers
472K  /var/lib/cassandra/data/system
4.0K  /var/lib/cassandra/data/system_traces/sessions
4.0K  /var/lib/cassandra/data/system_traces/events
12K   /var/lib/cassandra/data/system_traces
1.1M  /var/lib/cassandra/data/
1.1M  total
mssola:~ $ █
```

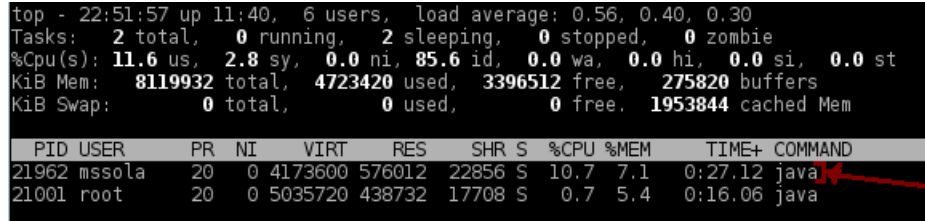
Figure 4.3: Disk usage of Cassandra

Cassandra is using a total of 1.1 MB of disk. Moreover, only 604 KB are actually the data from the database of this application. This is almost nothing, so disk usage is not an issue.

Storm

Finally, let us measure the Storm application. Sadly, a Storm application does not set a process name. Therefore, the name of the process will be “java”.

Anyways, we can perform the same “top” command as in the other components to get the CPU usage and the memory consumption. This is what I got after running the “top” command:



```

top - 22:51:57 up 11:40, 6 users, load average: 0.56, 0.40, 0.30
Tasks: 2 total, 0 running, 2 sleeping, 0 stopped, 0 zombie
%Cpu(s): 11.6 us, 2.8 sy, 0.0 ni, 85.6 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem: 8119932 total, 4723420 used, 3396512 free, 275820 buffers
KiB Swap: 0 total, 0 used, 0 free, 1953844 cached Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM     TIME+ COMMAND
 21962 mssola    20   0 4173600 576012 22856 S   10.7   7.1   0:27.12 java
 21001 root       20   0 5035720 438732 17708 S    0.7   5.4   0:16.06 java

```

Figure 4.4: Running top for the Storm application

As you can see, now I am running the whole application, and there are two java processes. Cassandra is started by systemd, so the user of Cassandra is root. Therefore, the Storm application is the one owned by my user “mssola”.

As expected, the Storm process is heavier than Cassandra. In this case, the Storm application is consuming around 580 MB of memory (where 22 MB are shared). To be honest I expected to see more shared memory across Java applications, but this is not happening in this application. Besides that, I expected the Storm application consuming even more memory, and this is not the case. Regarding CPU usage, the Storm application is around 10%, but these numbers are not steady. The maximum value that I have recorded is 30%. This leads to another unexpected fact: Storm is more CPU-bound that I had anticipated.

Total

After analyzing each component, we have already enough information to compute the requirements of the application. The application requires:

- **Memory:** it requires around 900 MB of memory. Since these numbers will be growing during the execution of the application, I would set 1.5 GB of memory as a requirement. If I add the memory that the Operating System and the other programs use, the requirement raises up to 1.8 GB more or less. Therefore, a computer running this application should have at least 2 GB of memory.
- **CPU:** the CPU usage is higher than I first expected. However, the CPU usage never goes higher than 30%. It usually fluctuates around 10% of CPU usage. My CPU is an Intel(R) Core(TM) i7-4650U with 4 cores and a frequency of 1.70 GHz. Therefore, any modern CPU can run this application with no problems.

- **Disk Storage:** this application makes almost no use of disk. Therefore, I would suggest to not set any requirements regarding disk for this application.

4.2 Pushing the limits

In the section 4.1, I have calculated the minimum requirements for this application. This requirements corresponded to a “normal execution”. Now, what if the application is not running “normally”, but rather intensively ?

First of all, the disk usage will not grow unless we add more cities to the system. the other cities do not have a lot of devices. Therefore, these numbers will not grow too much. I predict that it will never reach 1 GB. Therefore, to me disk usage is not a problem for this application.

The memory will certainly grow over time, but not too much. I have measured that in a normal execution, after some minutes, the Storm application consumes 5 MB more and it stays there. Therefore, even if the memory consumption grows a bit more on execution, it is not too alarming.

One thing that I have realized is that the “normal execution” is IO-bound. That is, the vast majority of time is waiting for some I/O operation. The most notable I/O operations in this platform are:

1. **HTTP requests.**

This I/O operation is certainly the most important one. The Storm application basically consists of making a huge amount of HTTP requests and then process the fetched data. The processing of this data does not require a lot of time, but it is CPU intensive. Since the CPU usage is fairly low for the Storm application, I think that it is fair to assume that the application spends a lot of time waiting for HTTP responses.

2. **Cassandra.**

This is not done very often, but it matters. Saving and restoring the state of the platform requires a handful of I/O operations to be performed.

3. **Temporary files.**

Storm creates on the background a lot of temporary files and temporary directories. These temporary files are basically logs.

This has lead me to the conclusion that the “normal execution” does not put this application into a test when it comes to CPU usage. To prove this, I have done a benchmark.

This benchmark consists of:

1. The **snacker-benchmark** mock service. This is a service that I am adding to the application. This service does not do anything interesting: it just requests data from a *mock* server and it prints this data.
2. The **benchmark** mock server. This is a very simple application. It is written in Go and has the endpoint that the snacker-benchmark service will call. This endpoint just returns a random number.

The flow of this benchmark is:

1. Storm will initialize and load the snacker-benchmark service.
2. The snacker-benchmark service will perform an HTTP request to the mock server.
3. The mock server receives the HTTP request and sends a response with a random number.
4. The snacker-benchmark service will “process” this random number and append it to a log file. This logging is done to check that the benchmark is working.

So, why am I doing this? The mock server is faking a server of the iCity platform. Since this mock server is running locally, we greatly cut down the time on HTTP requests. This service does not store anything into Cassandra. Therefore, the only I/O operations being performed are for logging purposes. Consequently, with this benchmark the application should be CPU-bound instead of IO-bound. We can check this by just running the benchmark for a while.

When a couple of minutes have passed, I perform the “top” command as seen in section 4.1. This time I have got the following results:

```
top - 23:46:38 up 12:35, 7 users, load average: 3.08, 2.92, 1.73
Tasks:  2 total,   0 running,   2 sleeping,   0 stopped,   0 zombie
%Cpu(s): 45.7 us, 18.6 sy,  0.1 ni, 33.2 id,  0.0 wa,  0.0 hi,  2.3 si,  0.0 st
KiB Mem:  8119932 total, 5146860 used, 2973072 free, 279472 buffers
KiB Swap:   0 total,   0 used,   0 free, 2162116 cached Mem

  PID USER      PR  NI   VIRT   RES   SHR S  %CPU  %MEM    TIME+  COMMAND
 24705 mssola    20   0 4196216 810704 22940 S 124.9 10.0   9:56.35 java
 21001 root       20   0 5035728 474012 17756 S   0.3  5.8   0:33.50 java
```

Figure 4.5: Running top in the benchmark

In the figure above we can see how the Storm and Cassandra processes are running. The Cassandra process is the one owned by root. If we take a look at it, we can see that the Cassandra runs in the same way than in a “normal execution”. That is expected. However, what is going on with the Storm process? According to top, the CPU usage is above 100%. This confirms that with this benchmark the Storm application is CPU-bound.

Now, how is it possible that the Storm application is consuming more than 100% of the CPU ? The CPU of my laptop has four cores. With this into account, it is relevant to see the following picture:



Figure 4.6: Load distribution across CPU cores

The problem with “top” is that it is not precise on the CPU usage since it is not taking into account that my laptop has four cores. The figure above was taken at the same time as the figure that displays a CPU usage of 124.9% for the Storm process.

This figure gives us very positive news: it scales on multiple cores. Instead of using the 100% on one core and making a more irregular use of the other cores (thus, making a bad use of the cores available); it distributes the load evenly across cores. That is, Storm does a clever job at distributing the load of the application across multiple cores. Therefore, I conclude that this application scales with the number of cores.

This conclusion leads me to a far more interesting statement: it is not about the brute force of the CPU, it is about the cores this CPU has. One consequence of this is that the computers that run this application do not have to have expensive and performant CPUs. Instead, this application requires CPUs with as many cores as possible, even if these cores are not the most advanced.

4.3 An ideal cluster

In the section 4.1 we have seen the minimal requirements of this application, and in the section 4.2 we have seen the limits of this application. Now it is time to think about an ideal cluster.

I am not going to give a detailed list of specifications or a specific hardware. Instead, in this section I am giving some tips and recommendations about how the components should be so the cluster is optimal.

CPU

As we have seen in the section 4.2, the most relevant factor of the CPU is not raw performance, but the number of CPUs. Therefore, in an ideal cluster, each CPU has at least four cores. Luckily, nowadays it is not hard to find CPUs with at least 4 cores, and they are not expensive.

The number of cores is the most important factor, but if we have this covered, then we can go for CPUs with more brute force per core. This depends on the money that we are considering to spend.

Memory

Memory is the most important factor for this application because it consumes a lot of it. The application alone consumes at least around 1 GB of memory. With the Operating Systems and the other programs running, I have estimated that the minimum requirement is 2 GB of memory.

Moreover, in the section 4.2 I have concluded that memory will not grow too much during execution. This is why I do not think that more than 4 GB of memory per machine is needed.

Furthermore, in an ideal cluster, I think that the most relevant property of the memory is not the capacity, but the speed. This application does a lot of accesses to memory. It is very important that this accesses are fast. This is why I would prefer a fast memory of 2 GB, rather than a slow memory of 4 GB. My tip here is to spend money on faster memory rather than to have more capacity.

Last but not least, it is also important to consider using the hard disk as an auxiliar memory in extreme situations. We can take advantage that this application does not consume a lot of space in disk, to create a big swap space.

Disk

This application does not need a lot of disk capacity. Therefore, we can save money here by picking disks that are the lowest capacity. However, even if disk is not used a lot, this application does use it (for example, to write logs), and ideally this accesses should be as fast as possible.

Therefore, I conclude that disks have to be fast and with low capacity. For this reason I believe that SSD hard disks are an ideal match for a cluster running this application. There are SSD hard disks with capacities such as 64 GB and 128 GB. Both capacities are fine.

Connections

As we have seen in the section 4.2, this application is IO-bound. However this application is CPU-bound if you cut down the I/O between HTTP requests. Therefore, Internet connection is a major bottleneck.

For this reason, I think that the designer of an ideal cluster should not be shy when spending money on fast cables, since this is going to be a decisive factor.

Nodes

This application runs smoothly in a single machine. However, if it keeps on receiving and receiving requests, it might come the day that this machine is overwhelmed. If this happens, we should consider having multiple nodes in the cluster for this application.

The number of nodes running is important because this application takes concurrency and parallelism really seriously. This application scales with the number of nodes that are running.

Chapter 5

The Development of the Project

5.1 The Schedule

5.1.1 Initial planning

I divided the load of work of my project into four clear stages. The first one was dedicated to study the feasibility of the project and to do the initial planning of the project. This was done through a course called Gestió de Projectes (GEP). This course includes the following stages:

- Scope of the project.
- Project planning.
- Budget and sustainability.
- Preliminary presentation.
- Bibliography.
- List of conditions.
- Oral presentation and delivery of the final document.

Project analysis and design

After taking the GEP course, I moved on and started the stage of “Project analysis and design”. The main goal of this stage was to draw a clear picture of the project and analyze all the goals of the project.

Therefore, this stage was made out of two sub stages. The first one, the analysis of the project. In this sub stage I devoted myself to define all the goals, requirements, features and use cases of my platform. The other sub stage consisted in design of the platform itself.

Project iterations

1. Development of the core software infrastructure

In this iteration I focused on the core infrastructure that has to hold the whole platform. This was executed only in the software front.

2. Providing services

The next iteration consisted on building the services on top of the base infrastructure that had been created in the previous stage.

3. Designing the cluster

At this point, we had the software ready to be deploy in production. So in this step I focused on writing down the specifications of an ideal cluster.

4. Concluding the development

In the last iteration I concluded the development of this project: running tests, final checks of the code, etc.

Final stage

The final stage consisted on closing the project. The main points of this stage were:

- Write the documentation of the platform.
- Write the Final report (this document).
- Give the lecture of my thesis (June 27, 2014).

Gantt chart

This initial planning was summarized in the following Gantt chart:

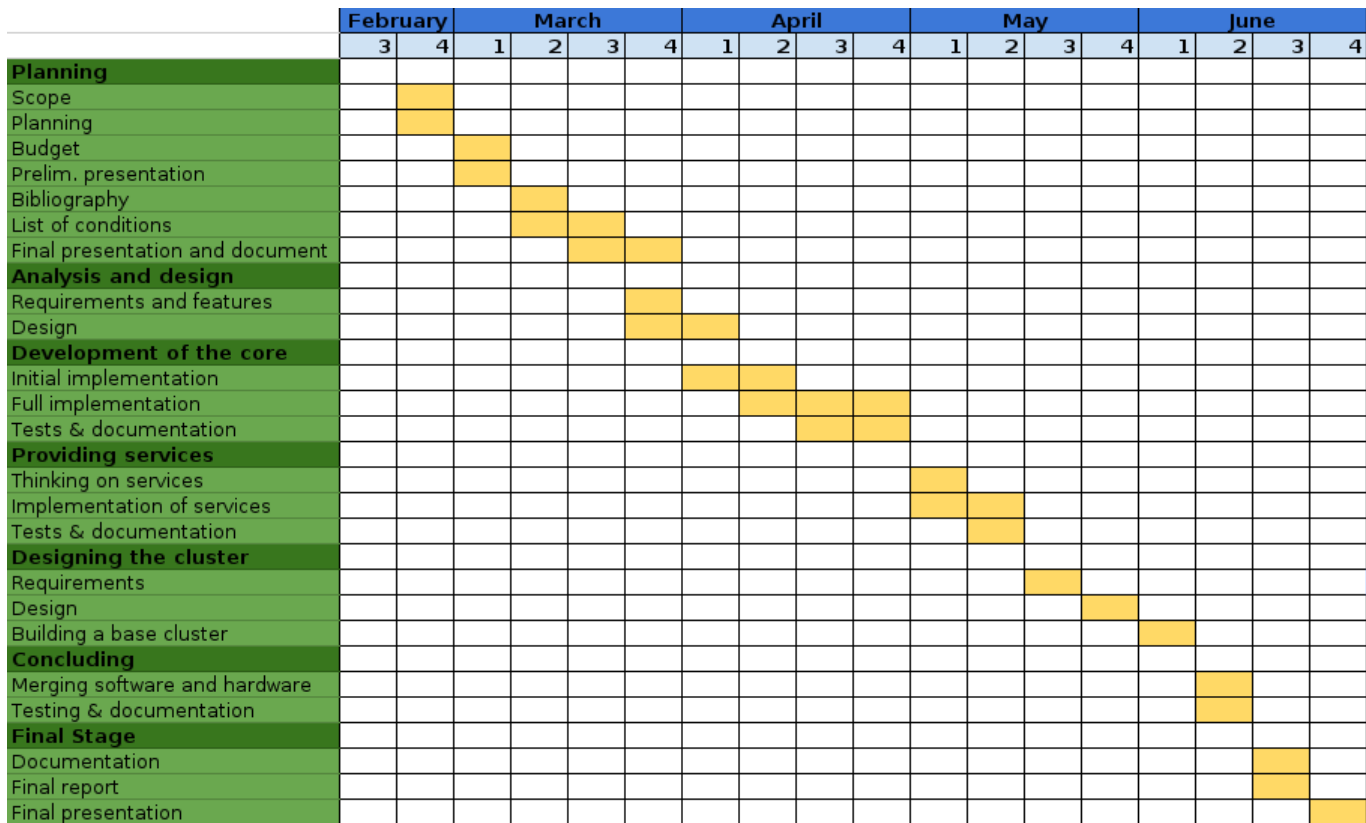


Figure 5.1: Gantt chart

One thing to note is that I computed my work load in weeks instead of computing it with hours. This might be odd, but it proved to be clear and simple. In my final report of the GEP course I stated that the total amount of hours per day was not fixed. However, I changed this policy to a much saner approach:

- On **weekdays** I dedicated 2 hours per day. This is because I had a day to day job and I also took more courses at university. This meant that I could not spend too much time during weekdays. Approximately, though, I dedicated 2 hours per day.
- On the **weekend** I dedicated approximately 8 hours to this project.

As you can see, I have been quite flexible regarding hours dedicated per day. Anyways, in total I have spent 17 weeks developing this project. This means that in total I have spent approximately:

$$17 \text{ weeks} \times (5 \text{ weekdays} \times 2 \text{ hours} + 8 \text{ hours on the weekend}) = 306 \text{ hours}$$

5.1.2 Changes in the midterm evaluation

There have not been any major change to my initial schedule. I have to admit that for personal reasons I started the project a couple of weeks later than expected. This means that by my midterm evaluation (“Fita de seguiment”), my final schedule shifted a bit from my original one. This did not result into many problems because:

- In my original Gantt I kept the “Analysis and design” and the “Development of the core” phases separately. I merged these two phases into one. This turned out to be fine.
- The “Providing services” phase was shorter than expected.

5.1.3 Final changes

Apart from my changes on the schedule by the time of my midterm evaluation, I did not have to do anymore changes in my schedule. The changes on the schedule did not affect negatively in the development of this project.

5.1.4 Conclusions

As you have seen, my initial schedule was surprisingly on point. I only had to do some minor changes on the schedule by the times of the midterm evaluation and that was it.

Even though I followed the schedule successfully, during the development of this project I felt that I was working too much during the stages “Development of the core” and “Providing services”. I think that one of the weakest points of my initial schedule was that I decided to spend too much time on stages like “Planning” and “Analysis and design”. I probably could have shortened this two stages, so I had more time on other stages that required more hard work.

5.2 The Budget

5.2.1 Estimated Salary

The first thing to consider when computing the budget for a project is to establish the wage for the people that will work on this project. In this project I am alone, therefore I just have to compute my salary for the given period of time that I have worked in this project.

So, what should I earn ? We have multiple possibilities when trying to establish a wage for a job:

1. Spain has a minimum wage, so it is an option.
2. We can pick the average salary for a Computer Programmer.
3. We can pick the average salary for a Big data developer.

The best option, in my opinion, is the second one. But since I also assumed the role of the manager, I will assign myself the average salary of a computer programmer in the USA, which is a bit higher than in Spain. According to USNews¹, the average salary for a computer programmer in the USA was 74,280\$ annually in 2012. I believe that this quantity is fair. In euros this is 54,860.49€.

I am computing my salary by just multiplying the number of hours with the €/hour rate. This salary does not include taxes. Here it goes:

$$\begin{aligned} 54,860.49\text{€ annually} &\Rightarrow 63.5 \text{ €/hour (18 hours/week as explained in section ??)} \\ 306 \text{ hours} \times 63.5\text{€} &\Rightarrow 19,431.0\text{€ in total} \end{aligned}$$

5.2.2 Software

Usually there are some costs associated with the software: licenses, patents, etc. In my case the following are costs associated with software:

- I have used the Archlinux² Linux distribution as my Operating System. This Linux distribution is free of costs.

¹<http://money.usnews.com/careers/best-jobs/web-developer/salary>

²<https://www.archlinux.org/>

- All the libraries and frameworks that I have used to build the platform and the services around it are open source and free of costs. This includes, for example, the following: the Storm framework, the Scala programming language, the Go programming language.
- I have developed the application with the KDevelop³ Integrated Development Environment (IDE). I have also used other editor such as Kate and KWrite. All these editors are open source and free of costs.
- I have written this memoir and all the delivered reports with L^AT_EX⁴ and the Kile⁵ editor. Both tools are open source and free of costs.
- I have used Git as the source code version manager. Git is also open source and free of costs.

Therefore, I conclude that I have **no** associated costs with the software that I have used. Everything that I have used is open source and free of costs.

5.2.3 Hardware

I have only used my laptop to go through this Thesis. I would not usually consider it as a cost, but it is an established practice when computing the costs of a project. This is to cover any incident that can happen to my laptop when developing this project. Lucky for me, my laptop has not suffered any damage. Anyways, we have to compute it.

My laptop is an Apple Macbook Air from late 2013 (with an extra of memory). I paid around 1400€ for this laptop. Since I have a warranty for 2 years, I will divide these 1400€ between 24 months. Therefore:

$$\begin{aligned} 1400\text{€} / 24 \text{ months} &= 116.66\text{€/month} \\ 116.66\text{€} \times 4 \text{ months} &= 466.64\text{€ in total.} \end{aligned}$$

This project has been developed and tested entirely in my local machine. Therefore, we do not have to compute hours in a cluster either.

³<http://kdevelop.org/>

⁴<http://www.latex-project.org/>

⁵<http://kile.sourceforge.net/>

5.2.4 Conclusions

My main conclusion is that all the costs of this project are tied to the people that are working on it. In this case, it is just me. Therefore, the costs of this platform can only be computed as the time that I have spent working on it.

Specifically, the costs are my salary and the amortized cost of my laptop. Note though that this result does not include taxes. With everything being computed, I conclude that the budget for this project is:

$$19,431.0\text{€} + 466.64\text{€} = \mathbf{19,897.64\text{€ in total.}}$$

Finally, it is time to decide if 19,897.64€ is a fair quantity. After doing some research I conclude that it is not too much. In fact, is a fairly low price for this kind of application.

Let's keep in mind that the salary that I have chosen is the salary of a "regular" programmer. Programmers from the field of Big data usually have higher salaries⁶. Therefore, I think that to be more realistic the costs should be higher.

⁶<http://readwrite.com/2013/09/24/big-data-apple-driving-industrys-biggest-salaries>

Chapter 6

Social and environmental impact

6.1 Environmental impact

In the same way that this platform can bring a lot of goodness in the social front, it certainly comes with a cost. In my case the cost is an environmental impact that cannot be understated.

This platform has to run in a cluster. A cluster, by itself, presents a lot of environmental issues that have to be dealt. If we do not do that, and we do not care about the environmental impact, it might happen that the cluster that executes this platform might be really aggressive towards the environment.

Another way to see it is on the bills. If, for example, this cluster does not make a proper use of electricity, bills will be higher.

Regardless how you see it, taking care of the environmental impact is a vital aspect of the implementation of this platform. The components that are more damaging to the environment are the following:

- Power supply.
- Maintaining a cooling system.
- The implied environmental costs of building the cluster.

All of these can be reduced by using the minimum amount of cluster time as possible. This means to run the software in “batches” or with a very low latency. However, this is not possible at all if the cluster has a lot of requests, and that is to be expected.

So, the only way to reduce the environmental impact is to build the cluster with components that consume as least as possible. The components integrated in the cluster have to have strong policies regarding environmental issues. Therefore, in this project there is not much that I can do to reduce the environmental impact: it only depends on the components of the cluster.

6.2 Social impact

Another thing that we might want to consider is how can this project impact upon society. In this regard we have to deal with the following topics:

- How can this project impact the economy.
- How can this project change how people interact with the city.

These two topics can be answered with: it depends. It depends on how other developers take advantage of this platform. It depends on how this project is implemented in public institutions, if it is implemented at all. It depends on how this project is further developed. Ideally speaking, it can have a big impact because:

1. If more developers join this platform or extend it, it can lead with the development of apps that can be used by regular users. This can improve both the perception and the experience of a user towards the city.
2. If the first point happens, it is very likely that people can earn money from the apps being developed.
3. The government can use the given information to improve some services or make them more efficient.

So ideally speaking there are a lot of opportunities around this platform. These opportunities would most likely have a very positive impact upon the city and the people in it.

6.3 Laws and Regulation

The platform that has been built during my Bachelor Degree Thesis does not violate any European or International law. From the very beginning it has been stated that

this platform fetches all the data from the *iCity* platform. This platform is an European effort that has been created according to the laws of the European Union. Therefore, the platform that I have built in my Thesis is also bound by the laws of the European Union.

For this same reason, I conclude that this platform is not infringing in any way the laws and the regulations from the European Union. Moreover, all the infrastructures integrated in the *iCity* platform, have their own policies and rules. This does not affect this project because all the infrastructures that I have chosen are free to use and with no limitations whatsoever.

Chapter 7

Conclusions

We can debate a lot of things in Computer Science but, in the end, the only thing that matters is if we have accomplished our goals. Looking back at the very beginning of this memoir we can read the main goal for this project:

“The goal of this project is to build a base platform that is able to generate rich information about a set of cities in real time.”

So, have we accomplished this goal? I think so. Looking back we can see that during this project I have accomplished the following milestones:

1. I have created a base platform that processes data in streaming. The base platform can be extended to fetch data from other sources, it can be extended to handle the data in a different way, etc. This platform is extensible and rock solid.
2. I have implemented a couple of services that prove that this platform works. The two services that have been developed in this project are quite minimalistic, but they both prove that the base platform is working and that a lot of things can be achieved with this software.
3. I have explored the limits of this platform, and I have explained them along the way. This way I have been capable to determine the requirements of the platform. Moreover, with these tests I have been able to give some hints on how a cluster can be built to run this platform.

Therefore, I believe that all the expectations for this project have been met, and I could not feel more confident about it.

Appendices

Appendix A

Storm: A quick introduction

It will be hard to review the code of the platform if the reader does not have any clue about Storm, and how Storm applications work. This is why I have written this appendix. It is probably not the most complete tutorial on Storm, but I believe that it gives the reader a quick and easy to follow introduction to Storm.

For this reason, I have added a directory called “storm-example” in the root of the project. In this directory there is a file called BasicTopology.scala. This file contains the easiest example that I could think of about Storm: counting words. In this example we have one spout and two bolts. The spout sends random sentences from a pool of sentences. The output of this spout is sent to the first bolt (called **SplitSentence**). This bolt will split the sentence into words and send the words to the last bolt. The last bolt (called **WordCount**) keeps track of the words that it has tracked so far. When to WordCount bolt receives a word, this word is printed to the standard output, followed by a number (representing how many times this word has been counted). Therefore, this is the **topology** of this example:

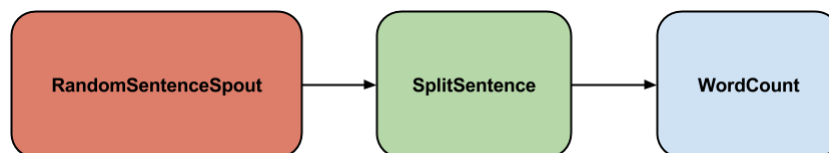


Figure A.1: The topology of the Storm example

Let’s dig into the code. First of all we will take a look at the RandomSentenceSpout. The code for this spout is shown by the figure A.2.

```
25 class RandomSentenceSpout extends BaseRichSpout {
26   var _collector: SpoutOutputCollector = _
27   val _sentences = List("the cow jumped over the moon",
28                         "an apple a day keeps the doctor away",
29                         "four score and seven years ago",
30                         "snow white and the seven dwarfs",
31                         "i am at two with nature")
32
33   def open(conf: Map[_, _], context: TopologyContext, collector: SpoutOutputCollector) = {
34     _collector = collector
35   }
36
37   def nextTuple() = {
38     Thread.sleep(100);
39     val sentence = _sentences(Random.nextInt(_sentences.length))
40     _collector.emit(new Values(sentence));
41   }
42
43   def declareOutputFields(declarer: OutputFieldsDeclarer) = {
44     declarer.declare(new Fields("word"))
45   }
46 }
```

Figure A.2: The code of the RandomSentenceSpout

A Spout in Storm has to inherit from the BaseRichSpout. After doing this, it has to implement three methods:

1. **open**: this method will be called each time that the bolt is opened. We can initialize our bolt here. In the most simple case, we just need to initialize our instance of the SpoutOutputCollector class.
2. **nextTuple**: this method will be called everytime that Storm needs a new value from this spout.
3. **declareOutputFields**. This method is the glue between this spout and the next bolt. It declares the number and the names of the fields that will be sent as the output. Note that the number and type of output values have to match the ones given by the “_collector.emit” call inside the nextTuple function.

Now let’s see the bolts. The first bolt is the one the splits the given sentences into words. The code for this bolt is shown in figure A.3.

As you can see from the SplitSentence’s code, bolts are even simpler than spouts. In this case, this bolt is a subclass of the class **BaseBasicBolt**. A bolt only has to implement two functions:

1. **execute**: this function will be called when this bolt has received some input. In this case the input is a sentence. The execute function will emit each word to the next bolt.

```

48  /**
49   * The SplitSentence class has the RandomSentenceSpout for the input
50   * and the WordCount bolt for the output. It receives a sentence and
51   * emits the words of this sentence.
52   */
53  class SplitSentence extends BaseBasicBolt {
54  def execute(t: Tuple, collector: BasicOutputCollector) = {
55      t.getString(0).split(" ").foreach {
56          word => collector.emit(new Values(word))
57      }
58  }
59
60  def declareOutputFields(declarer: OutputFieldsDeclarer) = {
61      declarer.declare(new Fields("word"))
62  }
63  }

```

Figure A.3: The code of the SplitSentence bolt

```

65  /**
66   * The WordCount only has an input, the SplitSentence bolt. It receives
67   * a word and keeps track of how many times each word has been visited.
68   */
69  class WordCount extends BaseBasicBolt {
70      var counts = new HashMap[String, Integer]().withDefaultValue(0)
71
72  def execute(t: Tuple, collector: BasicOutputCollector) {
73      val word = t.getString(0)
74      counts(word) += 1
75      collector.emit(new Values(word, counts(word)))
76  }
77
78  def declareOutputFields(declarer: OutputFieldsDeclarer) = {
79      declarer.declare(new Fields("word", "count"));
80  }
81  }

```

Figure A.4: The code of the WordCount bolt

2. **declareOutputFields**: it is the same as we have seen from the RandomSentenceSpout.

The last bolt receives each word and has to count it. You can see the code of this bolt in the figure A.4.

As you can see, there is nothing special about the WordCount bolt. Also note that the `declareOutputFields` method has to be implemented even if there are no more bolts listening to the output of the WordCount bolt. This output will be logged into the standard output.

We have now the three classes ready. Now it is time to use them in the main function. The main function is shown in figure A.5.

As you can see, the first thing that we do in the main function is to create a new **TopologyBuilder**. With this builder we can form a topology. As I have already said, the

```
83  /**  
84   * The object that holds the main function. It defines the topology which  
85   * is as follows: RandomSentenceSpout -> SplitSentence -> WordCount. This  
86   * topology will run locally.  
87   */  
88  object BasicTopology {  
89  def main(args: Array[String]) = {  
90      val builder = new TopologyBuilder  
91      builder.setSpout("randsentence", new RandomSentenceSpout, 5)  
92      builder.setBolt("split", new SplitSentence, 8)  
93      .shuffleGrouping("randsentence")  
94      builder.setBolt("count", new WordCount, 12)  
95      .fieldsGrouping("split", new Fields("word"))  
96  
97      val conf = new Config  
98      conf.setDebug(true)  
99      conf.setMaxTaskParallelism(3)  
100  
101      val cluster = new LocalCluster  
102      cluster.submitTopology("word-count", conf, builder.createTopology)  
103      Thread.sleep(10000)  
104      cluster.shutdown  
105  }  
106 }
```

Figure A.5: The code of the main function

topology that we are going to setup is the same as figure A.1. Nonetheless, note that with these same spout and bolts we could perfectly have different topologies. For example: we could create a topology consisting of one spout, 2 SplitSentence bolts and 1 WordCount bolt. This would work just fine because the classes for spouts and bolts are totally unaware of the topology. This flexibility is one of the most valuable advantages of Storm over other alternatives.

After building the topology, we setup the number of task parallelism. After this, we will submit the newly created topology. We will run this topology for 10 seconds, and then we are shutting it down.

This is all the code that we need for this example. It is simple, straightforward and fast. Now it is time to run this application to see how it actually performs. To do this we will be using the SBT toolchain. As you can see, there is already a configuration file for SBT for this project. This file is named “build.sbt”. Take a look at some of the options.

In order to run this application we will simply perform the following commands:

```
$ sbt  
> run
```

If it is the first time that you run this application, it will automatically download and install all the dependencies for you. The “run” command compiles the project and then it runs the executable main function.

```
[info] 7895 [Thread-20-count] INFO backtype.storm.daemon.executor - Processing received message source: split:10, stream: default, id: {}, [and]
[info] 7896 [Thread-20-count] INFO backtype.storm.daemon.task - Emitting: count default [and, 53]
[info] 7896 [Thread-32-split] INFO backtype.storm.daemon.task - Emitting: split default [dwarfs]
[info] 7896 [Thread-18-count] INFO backtype.storm.daemon.task - Emitting: count default [the, 86]
[info] 7896 [Thread-20-count] INFO backtype.storm.daemon.executor - Processing received message source: split:10, stream: default, id: {}, [dwarfs]
[info] 7896 [Thread-20-count] INFO backtype.storm.daemon.task - Emitting: count default [dwarfs, 26]
[info] 7897 [Thread-16-count] INFO backtype.storm.daemon.executor - Processing received message source: split:10, stream: default, id: {}, [white]
[info] 7897 [Thread-16-count] INFO backtype.storm.daemon.task - Emitting: count default [white, 26]
[info] 7897 [Thread-16-count] INFO backtype.storm.daemon.executor - Processing received message source: split:10, stream: default, id: {}, [seven]
[info] 7897 [Thread-16-count] INFO backtype.storm.daemon.task - Emitting: count default [seven, 53]
```

Figure A.6: The output from the Storm example

If everything went according to plan, you should see an output like the one shown in figure A.6. I have highlighted in red some of the results.

Bibliography

- [1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.
- [2] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. <http://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/p35-lakshman.pdf>, 2010.
- [3] Anish Nair Leonardo Neumeyer, Bruce Robbins and Anand Kesari. S4: Distributed stream computing platform. <http://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/S4PaperV2.pdf>, 2010.
- [4] Martin Odersky and many others. An overview of the scala programming language. <http://www.scala-lang.org/docu/files/ScalaOverview.pdf>, 2006.

List of Acronyms

EU European Union.....	1
API Application programming interface.....	iv
JVM Java Virtual Machine.....	6
LHC Large Hadron Collider.....	10
BSD Berkeley Software Distribution	
JAR Java ARchive	
DBMS DataBase Management System.....	16
AQS Air Quality Sensors.....	23
BSP Barcelona Sensor Platform.....	25
GEP Gestió de Projectes.....	41
IDE Integrated Development Environment	46