KARELIA UNIVERSITY OF APPLIED SCIENCES
Degree Programme in Applied Computer Sciences

Jonas Lesy
Ruben Vervaeke

APPLYING INTERNET OF THINGS – SMART CITY

Thesis
June 2015

THESIS
June 2015
Degree Programme in Applied Computer Sciences

Tikkarinne 9
80200 JOENSUU
FINLAND
Tel. 358-13-260 600

Author(s)
Jonas Lesy & Ruben Vervaeke

Title
Applying Internet of Things – Smart City

Commissioned by
Karelia University of Applied Sciences

Abstract

This thesis displays the progression and result of the Final Project that was realized by Ruben Vervaeke and Jonas Lesy, during the second semester of 2014-2015. The project is comprised of finding a solution for Process Genius, a company that needed an easy way to collect data from various public service providers. From bus schedules to the opening/closing times of the city's bridges, they wanted all kinds of data to be gathered in a central place. The purpose of the project was, therefore, to build a data warehouse and to create a web service to access the required data.

The first part of this thesis will describe the context of the project and the operation procedures. It will have a more in-depth description of the purpose and outline of the project and explain how the project was handled. It also includes what had to be done first, the planning stage of the project and some preliminary steps that were necessary to start the development.

This part is followed by the required theory to be able to understand what had to be done. This part introduces all of the important concepts upon which the system is built.

After that, the implementation is described, which consists of the steps taken to build the data warehouse and web services and how they interact with each other. The complete system is explained and the core components are discussed in detail.

This thesis ends with the results and discussion of the project. That part will tell to what extent the goals of the project were reached, which difficulties where encountered and what possible future actions might be.

Language	Pages 125
English	Appendices 3
	Pages of Appendices 7

Keywords

Internet of Things, Hadoop, Big Data, Data warehouse

CONTENTS

LIST OF IMAGES

LIST OF TABLES

APPENDICES

ABBREVIATIONS

If any abbreviations are mentioned throughout this thesis, the explanation can be found in this list.

BLL	Business Logic Layer, a layer, often implemented in programs that make connection to a database, which prevents invalid data operations.
CDH	Cloudera Hadoop, Cloudera's open source distribution which includes Apache Hadoop.
CRLF	Carriage Return Line Feed, a term defined to refer to the end of a line. It is often used when transmitting messages so that the system knows the end of the message is reached.
DAO	Data Access Object, a layer implemented in applications to make connection with a database, often used for retrieving data out of a database.
DWH	Data Warehouse, a complete system built to immediately answer requests for data without having to overload the original sources of the data. It is most commonly used for analytical purposes.
ETL	Extract, Transformation and Load, the part of a data warehouse which collects and unites data from source files to make it usable for analysis.
HDFS	Hadoop Distributed File System, the file system used by Hadoop to store its databases and files on.
HTML	HyperText Markup Language, the standard language developed to create web pages.
HTTP	HyperText Transfer Protocol, the protocol defined to provide communication between a web client and a webserver.
IDE	Integrated Development Environment, a software application used to develop different applications.
JAR	Java Archive, is a standard data compression and archiving format used for files written in the Java programming language.
JDK	Java Development Kit, a software package needed by developers to program in the Java language.
JSON	JavaScript Object Notation, a standardised format of defining data objects with their attributes. JSON files are easy to read by humans and often used to transmit data.

JVM	Java Virtual Machine, an environment for executing Java bytecode. For example, compiled Java code runs on a JVM.
RDBMS	Relational Database Management System, a database management system used to manage relational databases.
RPC	Remote Procedure Call, the technology that allows an application to execute code on another machine without having to know the code written for that application.
UI	User Interface, the interface that makes interaction between the user and the system possible.
URL	Uniform Resource Locator, a structured name that refers to a piece of data. It can for example be used to locate a website or local storage.
XML	Extensible Markup Language, a standard created to create formal and structured files which store data like for example configuration settings. This presentation is human-readable and machine-readable.
YARN	Yet Another Resource Negotiator, the new version of MapReduce in Hadoop’s framework. It is a programming model for executing jobs.

1 INTRODUCTION

This thesis is written by Ruben Vervaeke and Jonas Lesy, two Belgian students, and is the result of our Final Project carried out during our Erasmus exchange. This Final Project accounts for 22 credits and is the final step to getting a degree in Applied Computer Sciences.

This document is formatted according to the instructions of the thesis committee at Karelia University of Applied Sciences. The guidelines by them were followed during the entire process of this thesis. To make it easier to comprehend the technical aspects of this thesis, a different font type was used to indicate class names, attributes, code sample and commands. This report is the result of about twelve weeks of working on the final project.

The aim of this thesis is to inform the reader on the development and details of the project. It is written in such a way that everyone with some minor experience on the subject will be able to understand it completely. If there are any technical terms, abbreviations or jargon, they will be explained in a way that anyone with basic IT knowledge will be able to comprehend the text.

To start off, we would like to thank Mr. P. Laitinen for giving us the opportunity to work on this project and for providing us with the interesting subject we had to work with. If it was not for him, there would not have been any project and this document would not have been written. We would also want to thank Mr. J. Ranta for monitoring our project and taking the time to organize meetings together with Mr. Laitinen.

Next to that, we would like to thank everyone at Process Genius for letting us help them to find a solution for their problem. They have always been friendly and provided us with all of the necessary information to continue with the development of this project.

At last, we want to thank our family and friends, who have supported us during the progress of this project. Motivation is one of the keywords necessary to achieve goals.

2 ACTION PLAN

2.1 Project background

2.1.1 Organization

This project was conducted in co-operation with Process Genius, a company settled in Joensuu. The company started in 2011 and specializes in cutting edge 3D online services. This means that they provide 3D models especially made for industrial process plants and the sales organizations that supply them. These models are very useful because Process Genius can display all of the necessary and important data on them. For example, when a power plant in a company is down or malfunctioning, the cause can be seen immediately on the 3D model. This means fixing the problem or investigating malfunctions is more efficient and quicker. The company provides the complete solution by conducting research in the customer's power plants, and then provide the 3D model for it and all of the important data. User-friendly experience is important for them and so is productivity.

2.1.2 Mission and vision

The founder's passion is to combine their know-how on scientific topics with methods to develop next generation tools. These tools are created to optimize user experience and to boost sales. They have a wide global partner network that gives them a good stance in the competitive market. Next to that, they possess a highly skilled team to develop their tools. They excel in graphical design, industrial knowledge and web application development. In short, they have everything they need to deliver high quality products to their customers.

2.2 Problem description

The employees at Process Genius develop a complete solution for industrial and technological companies. They have developed the idea to deploy their project and technology to help the citizens of Joensuu. At the moment, data from many different public services is not accessible.

To give an example, there is no easy way to check the bus schedule. The city of Joensuu does already have a website for this but it is very unclear and inefficient. We have also tested this website and we can agree with this statement, the website is not user-friendly and the schedules are hard to read.

Next to the bus data, Process Genius also wants to display other data such as when the bridges go up and where the snow ploughing machines are. This will all be presented on their 3D model from the Joensuu city.

The idea is that a user can, for example, just click on a bus stop on the map and see when a specific bus will pass there. It is supposed to be an all-in-one solution again, similar to what Process Genius usually delivers.

2.3 Goals and project outline

As Process Genius stated, they want to have access to the data so they can use it to display information on their 3D model. This is where we, as a project team, came in. Our task was to provide them with the data, so they can access it whenever they want. In other words, our task was to set up some sort of a local storage which can be accessed by them. We chose to set up a data warehouse for this solution.

It can be asked why a usual database was not chosen to store the data. The answer is rather simple. First, there will be need to save many types of data, e.g. information on buses and bridges. This makes saving all of that data into just one database not that easy, especially when the data will be processed and accessed later on.

The second important reason behind this approach was the fact that a usual database is not meant to perform data analysis on. Since another project team will be working on analysing and investigating this data, a data warehouse is a much better solution for them, too.

Important aspects of this project are co-operation and a future-proof solution. The latter is mentioned for the following reason. It is impossible to add every useful feature and data considering the timespan we had for this project. If someone else continues to develop our project (probably Process Genius, or maybe other students), they must be able to easily integrate those other new features. This project is built upon sustainability and we want to be sure that it can be used and modified easily. All of this means that we had the following tasks:

- define which data to be used and translate it into DB-models
- setting up Hadoop, Cloudera and our database (HBase)
- write a MapReduce script to transform the data into desirable format
- write all other necessary scripts and programs (Data puller, DAO, BLL, …)
- write a web service so Process Genius can access the data.

2.4 Project approach

This part of the report describes how we approached the project. It is not a detailed description, but only includes what was done, how the project started and which tools were used. It will give an overview of how the project progressed and what we did in general.

To fulfil this project, we first talked with our thesis supervisor, Mr. Laitinen and he explained us in short what the task was. After that, we had a meeting at Process Genius. They told us what the project was about and what they wanted.

These were the preliminary steps in our project. After this, we could start brainstorming on how to approach this project. Next to that, we were told to use the SCRUM-tool Trello, which makes it possible to follow up our project easily. The active tasks were displayed on the tool's interface and there was clear a view on who was doing what.

The next thing we did was investigating Hadoop and Cloudera, which both are discussed and explained later in this report This took a long time because Hadoop was completely new for us and we had to perform research on all of its different components.

We first ran a minimal version of the setup on our own laptops and later on switched to servers provided by the school. This setup included a running version of Hadoop with Cloudera on top. At that point, the problem was that we didn't have enough memory to run it smoothly and it started lagging right away. This problem existed until we were able to use the servers on campus.

Later, we installed Cloudera on the servers. Using these powerful servers, we were no longer hindered by the RAM issue, which made us able to work more efficiently.

While we were still waiting for some example/test data, we started building a small testing setup on which a self-made CSV-file could be transformed into the desired format. This testing setup ran on the servers and used the MapReduce functionality and the Data Puller.

After we got sample data, we started building every part of our warehouse by one component at a time. Slowly but steadily, the whole setup came to life and everything was tested, part by part. During the last weeks, everything was put

together and the parts were merged into one big system which represents the data warehouse. This system now pulls, processes and saves data and makes it accessible to other users/companies via a web service.

3 INTERNET OF THINGS

At home, at work, in your car and even on the road, the Internet is simply everywhere. It's practically impossible to remove it out of our everyday life. We get up and check the latest Facebook updates on our smartphones, read the news on our tablet while having breakfast, arrive at work and check our e-mails, go home and search for a recipe to prepare our favourite meal to then end by going to bed and reading a book bought on an e-book store. The Internet is used every single day and for every purpose you can possibly imagine.

But next to those daily uses of the Internet by individuals, there are thousands, if not millions, of Internet-based solutions for all of our problems. But the Internet is not only used by individuals but also for creating business solutions. Many companies have their own issues, be they small or big, which can be solved by implementing the use of the Internet.

Now to sum up the meaning of the term the 'Internet of Things', it comes down to the fact that the Internet is used for and by way more objects than smartphones, laptops and such. It actually means that these types of devices, which are not always operated by humans, are outnumbered by other ones. In other terms, different objects become available throughout the Internet, these objects are also referred to as embedded systems. These objects will be able to communicate over the Web and even take autonomous decisions. To give a small example, the Internet of Things could make it possible to start your microwave by using an interface or application on your smartphone or tablet.

This technical development creates huge opportunities, solutions and innovations. Almost everything you can think of can be connected to each other for whatever purpose desired. When this development is aimed at businesses, lots of new technologies can be created and implemented. Sensors can detect malfunctions and display them in a central interface, water levels can be meas-

ured and monitored, everything is possible. All these solutions make the everyday workflow of a business easier and more efficient.

But with these new and big technologies, it's inevitable that there are some disadvantages too. To start, there's a lot of criticism concerning privacy issues and environmental pollution. It is obvious that people don't want everything to be monitored and stored. What's important about the 'Internet of Things' is that there has to be a balance between right and wrong, allowed and not allowed, ...

The kind of monitoring and measuring mentioned above is usually done on a regular basis, if not continuously. This causes the creation of huge amounts of data, which need to be stored, processed and analysed. This brings us to the next chapter of this thesis.

4 BIG DATA

Wherever you look, there are different kinds of information spread out everywhere. Brands, names, dates, … All these words and numbers are probably stored somewhere and are also processed. With information being everywhere and companies wanting to store all kinds of data, the term ‘Big Data’ came up. The amount of data that is stored grows exponentially. Think for example about Facebook, storing all of their users’ profiles, pictures, messages and much more. Next to usual websites, and following the previous chapter, the Internet of Things also creates big amounts of data.

There are different reasons for the existence of big data and storing those large amounts of information. Probably the biggest cause is the growing desire for analysis of this data. Companies want to know everything about their customers, employees and business associates. They want to follow up on their customers’ purchasing behaviour and for example send aimed sales promotions and such.

This eventually brings up the importance of marketing purposes. Not only can big data be used for marketing strategies, data is also often sold and bought between companies.

An important note is that the term ‘Big Data’ can’t be used for every kind of data. There are three core factors to big data. The American information technology firm, Gartner, has defined these to describe big data.

The **volume** of the data:

Off course, the quantity of the data is important when considering big data.

The **variety** of the data:

It is important that analysts know how to categorize the data. This makes it possible to use the data in an efficient way.

The **velocity** of the data:

This term refers to the speed in which the data is generated or in which it is generated and processed. The velocity must be calculated to find out if the system meets the requested speed.

When all these conditions are met, the data that is being processed can be called big data. But next to these, there are three other key components that are important for processing and analysing big data.

The **variability** of the data:

The data must all be consistent. Inconsistency can block and slow down the processing of the data.

The **veracity** of the data:

Veracity is also very important. The data that is processed must be reliable and originate from a good source. This greatly affects the results of analysis.

The **complexity** of the data:

When data comes from different sources and in large volumes, it needs to be connected and linked in order to make it useable for analysis.

Are regular databases big enough to handle and store these huge amounts of data? This question is answered and discussed in the next chapter.

5 DATA WAREHOUSES

As it is probably clear after reading the previous chapters, there is a lot of data processed and stored nowadays. In this chapter, we'll discuss the processing of the previously mentioned Big Data. If we bring up the term Big Data, then data warehouses can't be left out of the conversation.

A data warehouse can be seen as a complete setup that automates the processing and storage process of Big Data. It is commonly used for reporting and data analysis. This means that data warehouses are optimized for these tasks. What's also very important is that they do not only store the current data but also historical data, hence the optimization for analysis. A data warehouse, or DWH, can be used to track differences in sales, weather conditions, … , almost everything you can think of.

Another key feature about data warehouses is that they automatically gather source data. Connection is made between the DWH and source and from then on, data will be stored into the DWH periodically. This happens through the ETL part of the warehouse, Extraction Transformation and Load.

Extraction

The data is extracted from different sources (homogeneous[1] and heterogeneous[2]). The system will be able to extract from sources from any location.

Transformation

The data is transformed so it is stored into the proper, desired format. This format is defined in advance and optimized for future analytical purpose.

[1] Homogeneous data: data of the same sort, type and kind. Like a combination of text files.
[2] Heterogeneous data: data of different sorts. Like video clips, spreadsheets, sound fragments, ...

Load

The data is finally loaded into some kind of storage. This is actually a database and can be a data mart, a data warehouse or an operational data store.

What makes this system very performant is that it runs parallel. If there has already been a small part extracted, let's say a couple of files, then the system doesn't wait. It immediately starts to transform the data. When this has happened, even if not all extracted files are transformed yet, the system already starts to load. There's little to no time waste and if configured well, it is indeed a very efficient system.

When regarding the load part of ETL, you may notice that it is stated that the data yet has to be loaded into, for example, a data warehouse. This may be confusing since the ETL process itself is part of the data warehouse. This shows the fact that a data warehouse is an all in one solution. It can perform the complete process of retrieving and storing the data but the data can also be stored somewhere else. All of this is defined when the warehouse is being set up. The designer chooses which tasks are performed by the warehouse and in what way this happens.

"Why use a typical DWH database and not a regular relational database?", you may think. To start, we'd like to mention that a DWH actually has a database to store the data, it's just not a usual relational one. Now this possible misunderstanding is out of the way, let's answer the question. First we'll explain what a relational database is and how it works.

A relational database is based on the relational model of data. This means that there has to be some kind of hierarchy and/or a logic construction in the database. In a relational database, the data is split into one or more different tables, which then consist of rows and columns. Each row has a unique key, which makes it possible to link a row in another table by storing this unique key in it. This copied key is then called a foreign key, but we'll not go too much into depth considering database terminology. As it might be clear, a relational database has a very strict structure and rows/columns can in most cases not always be

deleted as desired. The structure must be maintained and all dependencies must be removed first before being able to delete a key value for example. Let's take a look at the example shown in figure 1.

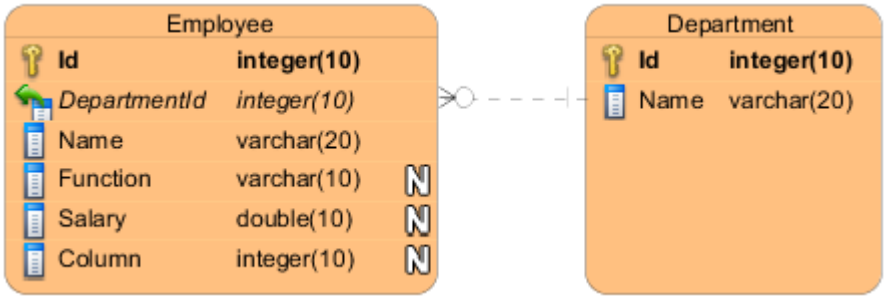


Figure 1 - Relational database model example

We can see two tables, usually there are way more but this is for the sake of simplicity. Each employee works in a department and each department can have multiple employees working in it. That's our relation right there. We can also see the foreign key, which was described earlier, in the `Employee` table. The employee gets the `DepartmentId` foreign key from the `Id` out of the `Department` table. In other words, an employee is linked to a department by the use of this foreign key. This in turn, means that a department can't be removed as long as there are employees working in it. Let's say we want to remove the department `Sales` out of our database, for any reason whatsoever. To be able to do this, we have two options. The first one is to remove every `Employee` that has the sales `DepartmentId` as foreign key, which would be highly inefficient. The other option is to change this `DepartmentId` foreign key to another department's key or leave it null, which can be seen as empty. It's actually not empty but as mentioned before, we won't go too deep into database theory.

Now that it is clear what's typical about a relational database, let's explain why it shouldn't be used in our case and what some of the relevant disadvantages and advantages are. To start off, a relational database is normalized. To explain this, it means that redundant data is removed and there is a strong hierarchy in the database. The advantage of this is that it saves up storage space. A relational database is optimized for write operations, it just is, there's no way around it. It is built to add and/or change data. On the other hand, a data warehouse is

built for fast reading operations and achieves high performance when executing analytical queries.

As stated earlier in this thesis, a data warehouse also saves historical data, hence the optimization for analysis purposes. To save all of this historical data into a relational database would be pretty much impossible. Actually it would be feasible but it wouldn't make any sense. A relational database is not built for fast, performant analytical queries, so it would be absolutely useless to do this.

Since the data we store will be mostly used to read out and display on an interface, good reading performance is highly recommended. Next to that, analysis is also an important feature for our setup, because a lot of analytical queries will be performed. This makes a data warehouse a better choice for our solution.

To make a final statement, relational databases are surely not replaceable. But a data warehouse came up for other purposes, to execute other tasks. The importance is to choose the right solution for the right case, either solution has its own advantages. Figure 2 provides a closer look on how a data warehouse might look like.

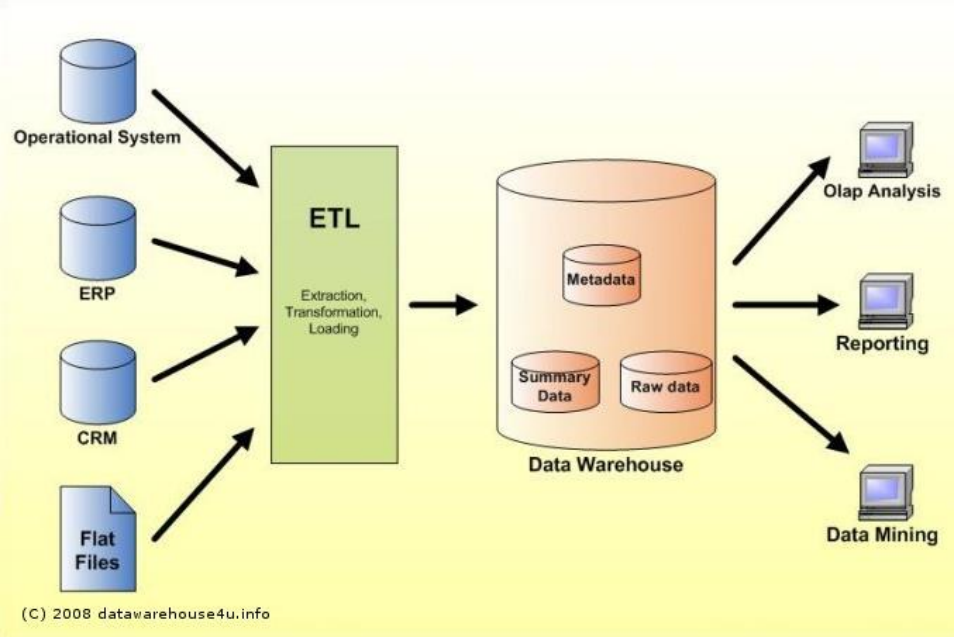


Figure 2 - Data warehouse overview[3]

[3] Datawarehouse4u. 2009. Data Warehouse.

The figure shows that a data warehouse gathers all kinds of data from different sources by the use of ETL, as described earlier. It then stores the metadata, summary data and raw data in its database and provides it for different analytical purposes. ETL can in fact be seen as part of the DWH because it is optimized for this data warehouse.

6 HADOOP

6.1 Introduction

Big data is becoming more and more a hot topic. Having data available and the resources to process this data can create impressive opportunities in the business world, like we discussed before. The amount of publicly available data grows every year and organizations no longer have to rely on their own data. But, with this advantage, it becomes harder to pull this data and use it in a satisfactory manner.

The capacity of hard drives has increased and the price per GB is at its lowest point in years. So data storage has become more affordable, which is a good thing. Increasing capacity is something that was relatively simple to obtain, but unfortunately a mechanical hard drive is still a mechanical hard drive, and mechanical parts have their limitations. It's the access speeds of hard drives that haven't increased over the years. Although solid state drives are becoming an interesting option because of their access speeds. The negative part of this is that they are still very expensive at the rate of 1 euro/GB in the year 2013 (and that's for consumer grade hardware).

A good solution for this has been around for quite some time now. The configuration of reading/writing from/to multiple hard drives at once, or in parallel. Unfortunately this approach comes with some possible problems. The first is hardware failure, adding more and more hard drives to a configuration has an increased chance of one of these drives to fail. A solution for this problem is to replicate data so it exists multiple times on the server. This can be a form of RAID. A disadvantage however of this solution is the presence of redundant data. A second problem is when using analytical tools to perform analysis on data. When storing this data in multiple places, it's challenging to implement a system that handles analysis on all of this data.

Based on these problems and developments the Hadoop framework was created. Hadoop is a framework that uses MapReduce algorithms and HDFS file system, to overcome the problems mentioned above. The Mapping part can be seen as the transformation component in ETL. The Reducing part overcomes the specific problem of analysis on big data. In this topic we will explain what Hadoop is, how it's designed and what makes it tick.

6.2 Components

Hadoop is a framework, it's not just an application you can run on a machine. It's a collection of components someone can use to satisfy his/her needs. The two biggest and most important components are MapReduce and the distributed file system HDFS (Hadoop Distributed File System). But there are other projects that were created for the Hadoop framework as well. Hadoop is now part of the Apache Software Foundation and Apache has created several other tools for the framework. The greatest thing about this framework is that it's completely open source. This means that Hadoop and all of its tools, extensions and components are free to use.

This is a list of the most important components of the Hadoop framework. The key components for our project are MapReduce, HDFS, HBase and ZooKeeper.

Common
The common part of the framework provides all of the required tools for the HDFS and general I/O. (serialization, persistent data structures, Java RPC).

Avro
A system for efficient, cross-language RPC and persistent data storage.

MapReduce
A distributed data processing model and execution environment that runs on large clusters of commodity hardware.

HDFS

A distributed file system that runs on commodity hardware in large clusters.

Pig

An execution environment and data flow language for exploring very large datasets.

Hive

A data manager in HDFS that performs queries on the data using a query language based on SQL.

HBase

A column-oriented database that uses HDFS for storage. It supports both batch-style operations using MapReduce and point queries (random reads).

ZooKeeper

Manages connections between nodes and provides a security layer.

Sqoop

A tool for efficient bulk transfer of data between structured data stores (such as relational databases) and HDFS.

Oozie

A service for running and scheduling workflows of Hadoop jobs.

6.3 MapReduce

MapReduce is a programming model for data processing. It is used by Hadoop and is one of the two parts that provide Hadoop its strength. These MapReduce programs can be written in programming languages like Java, Ruby, Python and C++. Its model is designed to work inherently parallel. All following code examples will be written in Java, because the authors of this thesis know Java quite well and the Hadoop framework itself is written in Java.

MapReduce contains two words, Map and Reduce. Both enable the processing of data to happen in parallel execution. Each phase works with key-value pairs as input and output. These can be of any type of data provided by the programmer. To be more concrete, the programmer needs to provide two functions: a map function and a reduce function. The map function converts raw data in usable data. The reduce function processes this usable data in any manner desired by the developer.

The data

To help understand everything, we will use a practical example. Let's start with some raw data we retrieved from the city of Joensuu. We received some excel files with information about detection sensors on crossroads in the city centre. Each excel file represents a crossroad and each worksheet within the file represents a sensor. To simplify this example, we converted one worksheet into a CSV format. Below, the example can be found.

```
Hour;2015-03-02;2015-03-03;2015-03-04;2015-03-05;2015-03-
06;2015-03-07;2015-03-08
0-1;25;18;25;37;21;66;49
1-2;20;19;14;18;8;35;40
2-3;7;11;6;12;6;40;46
3-4;19;5;5;19;12;38;53
4-5;14;9;7;21;13;34;44
5-6;49;36;43;38;42;20;15
6-7;160;177;169;173;156;42;41
…
23-24;18;21;28;26;62;81;32
```

Each row in the file represents one hour of a day. Each column represents a day in a week and the whole file represents the data of one week. The values themselves indicate how many times the detection sensor has been triggered. In our case, vehicles driving over the sensor will trigger these.

Now we need a MapReduce algorithm that we can apply on this data. Let's say we want to calculate on what day of the week the amount of vehicles passing by was the largest.

Map and Reduce

So the first step in our MapReduce application is to map this raw data to some usable data. Remember that all input/output used by the Hadoop MapReduce functions are defined as key/value pairs, because default mappers and reducers read data in line by line. So when we read for example the third line of our file, we need to have a reference to the date that the value corresponds to, therefore in our Mapper class we keep a static property containing all the dates.

The input key in our Mapper doesn't really matter, we define it as a `LongWritable` datatype which corresponds to the size in bytes of the line that the Mapper reads in. For the input value we define the type `Text`, because we expect a series of characters. For example the first 3 lines the Mapper will read in looks like this:

```
(0,Hour;2015-03-02;2015-03-03;2015-03-04;2015-03-05;2015-03-
06;2015-03-07;2015-03-08)
(83,0-1;25;18;25;37;21;66;49)
(109,1-2;20;19;14;18;8;35;40)
```

The input key is 0 for the first line because 0 bytes have been read by the Mapper, or in other words the first value in the input value will be the 0th byte in the file. The input key for the second line is 83 because the first line was 83 bytes in size.

Now we need the Mapper to define an output key and value. If we make the output key an array of the dates, we can define the output value as an array of sensor readings. This way the reducer can assign a value to the correct date. So both output key and value are arrays of values. Below, you can see the first 3 outputs of the Mapper.

```
([2015-03-02,2015-03-03,2015-03-04,2015-03-05,2015-03-06,2015-
03-07,2015-03-08], [25,18,25,37,21,66,49])
([2015-03-02,2015-03-03,2015-03-04,2015-03-05,2015-03-06,2015-
03-07,2015-03-08], [20,19,14,18,8,35,40])
([2015-03-02,2015-03-03,2015-03-04,2015-03-05,2015-03-06,2015-
03-07,2015-03-08], [7,11,6,12,6,40,46])
```

All output of the Mapper is processed by the MapReduce framework before it is sent to the reduce function. This process consists of sorting and grouping the key-value pairs by key. After this processing, which is called shuffle, our input for the reducer will look like this:

```
([2015-03-02,2015-03-03,2015-03-04,2015-03-05,2015-03-06,2015-
03-07,2015-03-08], [[25,18,25,37,21,66,49],
[20,19,14,18,8,35,40], [7,11,6,12,6,40,46], … ])
```

This looks a bit complex, but basically the framework creates an array of all the map output values where the key is the same. In our case the key is the equal for all values. So we will have 1 input for the reducer, containing the array of dates as the input key and an array of integer arrays that represent the values.

Then the reducer processes the input, in our case it does calculates the summation of sensor readings for each day and then calculates the maximum value for a particular day. The output for the reducer is very simple. We define a `Text` output key that represents the date and `IntWritable` output value that represents the number of times the sensor has been triggered:

```
(2015-03-06, 4075)
```

On figure 3, all of the taken steps can be seen. The previously explained output can be found in the white boxes but what's most important is to get an overview of how the whole system works.

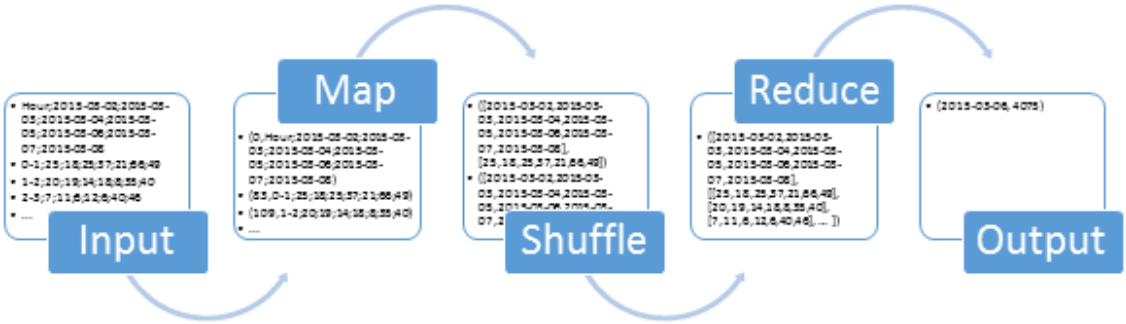


Figure 3 - MapReduce workflow

In Java code

We took a look on how the process works of a MapReduce algorithm, now we need to express this process in code. In total we need 3 classes, a mapper, a reducer and a main class to run the job (import statements are left out in these examples to safe some space).

The first is the Mapper that we can create by subclassing the `Mapper` class and defining four type parameters that specify the datatype of input and output, keys and values. This class defines an abstract method `map()` where we can define our transformation of the data.

```
public class MaxVehiclesMapper extends Mapper<LongWritable, Text,
    StringArrayWritable, IntegerArrayWritable> {

    private static String[] dates = new String[7];

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String[] values = value.toString().split(";");

        if (values[0].equals("Hour")) {
            configureDates(values);
            return;
        }

        IntWritable[] readings = new IntWritable[7];
        try {
            for (int i = 0; i < readings.length; i++) {
                readings[i] = new IntWritable(Integer.parseInt(
                    values[i + 1]));
            }
        } catch (NumberFormatException ex) {
            System.err.println("Exception: " + ex.getMessage());
        }
        context.write(new StringArrayWritable(dates),
            new IntegerArrayWritable(readings));
    }

    private void configureDates(String[] values) {
        for (int i = 0; i < dates.length; i++) {
            dates[i] = values[i + 1];
        }
    }
}
```

We know our csv file has a heading with all the dates of the readings, so we can write a check to see if we read a line that is the header. If so, we use a private method `configureDates()` to write the dates to a static string array called `dates`. If we are not on the heading, the method continues and loops over all the values in the line separated by a semicolon symbol. In the for-loop we parse the value to an `IntWritable` (which is the datatype used by Hadoop for integer values) and store it in an array of `IntWritables`. When done, we write our string array to a `StringArrayWritable` (which is a custom datatype defined in the project that represents an array of strings) for the key output and and an `IntegerArrayWritable` for the value ouput.

The Reducer looks a lot like the Mapper in terms of class and method prototypes. We can create a Reducer by subclassing `Reducer` and writing the type parameters which again define the input key and value, and output key and value. The `Reducer` defines an abstract method `reduce()` that is used to define a calculation on the input key/value. The result of the calculation is written to the output key/value. In our case we calculate the maximum value in an array of Integers.

```
public class MaxVehiclesReducer extends Reducer<StringArrayWritable,
    IntegerArrayWritable, Text, IntWritable> {

    @Override
    protected void reduce(StringArrayWritable key,
        Iterable<IntegerArrayWritable> values, Context context)
        throws IOException, InterruptedException {

        int[] readingsPerDay = new int[7];
        Iterator it = values.iterator();

        while (it.hasNext()) {
            IntegerArrayWritable iaw =
                (IntegerArrayWritable) it.next();
            for (int i = 0; i < iaw.get().length; i++) {
                readingsPerDay[i] +=
                    ((IntWritable) iaw.get()[i]).get();
            }
        }

        int maxValue= Integer.MIN_VALUE;
        String date = "";

        for (int i = 0; i < readingsPerDay.length; i++) {
            if (readingsPerDay[i] > maxValue) {
```

```
                maxValue = readingsPerDay[i];
                date = ((Text) key.get()[i]).toString();
            }
        }

        context.write(new Text(date), new IntWritable(maxValue));
    }
}
```

In the first part of the `reduce()` method we make a summation of all the readings per day. Remember that we receive an input value of a collection of `IntegerArrayWritable` objects. This is why the first part looks a bit funky. We iterate over the collection of `IntegerArrayWritables` which was created by the Hadoop shuffle algorithm after mapping was completed. Then we iterate over the `IntegerArrayWritable` to retrieve all the readings of every hour/day. We add the reading for every hour to the `readingsPerDay` array so we get a summation of all the readings per day.

The next part is to calculate the maximum value in the `readingsPerDay` array. This part is pretty straightforward as we keep a `maxValue` Integer to store the maximum value in and a string to store the corresponding date in. Once we are done with our analysis we can write our results to `context` by providing a new `Text` for the key and a new `IntWritable` for the maximum value.

The third and last piece of code we need is called a Driver class in Hadoop terms. It defines a Hadoop job that can be run by defining a main method. Inside the main method we create a new instance of Job. On this Job instance we can set various properties to define the classes, input and output types needed for the job. We also set the input path (for the source file) and the output path (for storing the result), by using the main method's string parameter.

```
public class MaxVehicles {

    public static void main(String[] args) throws Exception {

        if (args.length != 2) {
            System.err.println("Usage: MaxVechicles <input path>
                <out-put path>");
            System.exit(-1);
        }
```

```
        Job job = new Job();
        job.setJarByClass(MaxVehicles.class);
        job.setJobName("Maximum vehicles for day of the week");
        FileInputFormat.setInputPaths(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(MaxVehiclesMapper.class);
        job.setReducerClass(MaxVehiclesReducer.class);
        job.setOutputKeyClass(StringArrayWritable.class);
        job.setOutputValueClass(IntegerArrayWritable.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Normally when a job is run on a Hadoop cluster, it is packaged into a JAR file. This way Hadoop can provide the algorithm to all nodes that execute parts of the job. We can call the `setJarByClass()` method to define a class that Hadoop can search for in JAR files when it wants to execute a job.

We can provide multiple input paths for the job, if we want more than one input file for the MapReduce algorithm. We can define one output path to indicate where the result of the algorithm should be written. Note that all these paths are relative paths to the root directory of the Hadoop HDFS file system. It's important to note that the output path mustn't exist, otherwise the framework will throw an exception.

The last properties we can set are the Mapper and Reducer classes that the job should use. Together with the output types for the Mapper, all the necessary properties are set. Eventually we can call `waitForCompletion()` method to execute the job and wait for it to finish.

Running the program

We wrote our MapReduce application, now it's time to run the application. Before we do this, we need to make sure Hadoop is running properly on our system (how to do this is mentioned in chapter 10.2 Cloudera installation). In our case we are running the application on Ubuntu where Hadoop is configured in pseudo-distributed mode. There are 3 steps we need to perform to successfully execute the program:

1. Copy the data file onto the Hadoop distributed file system. We need to make sure that the resource is available on the HDFS file system. We can use the shell command `-copyFromLocal` from the Hadoop `fs` component. We need to specify the source path and the destination path. In our case Hadoop is configured in pseudo-distributed mode, so our destination path is `hdfs://localhost`. When using this command, the copy will be placed on the `/user/hadoop` path of the HDFS file system.

```
% hadoop fs -copyFromLocal /tmp/input/data.csv
hdfs://localhost/input/data.csv
```

2. Next we need to tell Hadoop that it can find the JAR file that we created when we built the program. We do this by exporting the name of the JAR file onto the hadoop classpath.

```
% export HADOOP_CLASSPATH=MaxVehicles-1.0.jar
```

3. The last step is executing the application via the `hadoop` command. We specify the class that Hadoop needs to run by typing its package name, followed by the class name. We defined the input path as the first string argument of the main method and the second as the output path. We wrote our main method so that the program runs only when there are 2 arguments provided. So we define our relative input path where the input file is located and define a non-existing output path.

```
% hadoop fi.karelia.maxvehicles.MaxVehicles
/user/hadoop/input/data.csv ouput
```

When we execute the last command, if everything went well, you should get some terminal output (shown in Appendix 1) which displays a lot of useful information about the job. We will talk about this whole workflow in chapter 6.3.2 The process of a MapReduce Job run.

After the job has completed execution, we can check the output directory on the HDFS file system. There we find a file named `part-r-00000.txt`, which is

the output from our reducer in our program. We have one reducer in this application, so there is only one output file. If multiple reducers would be defined we would get a `part-r-xxxxx` file per reducer. The content of this file can be seen on figure 4.

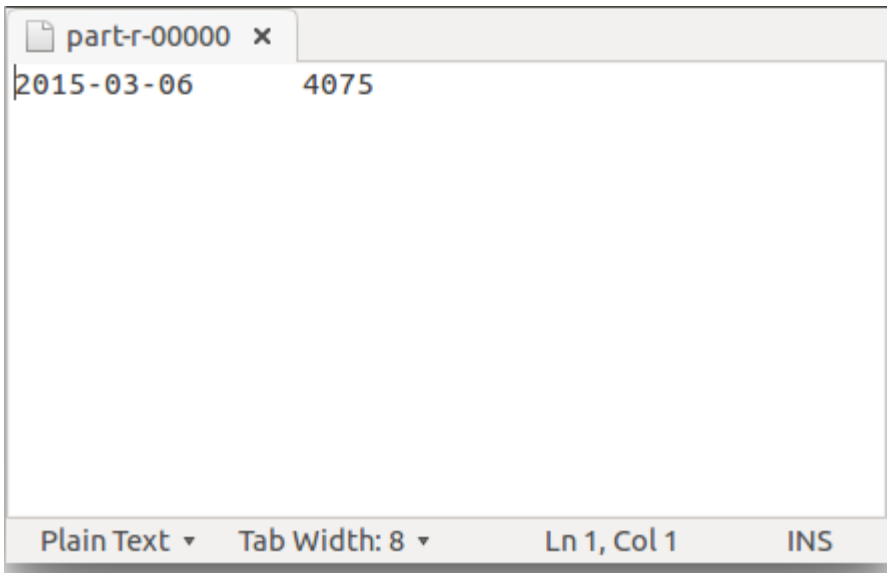


Figure 4 - Reducejob result

This file contains the date on which the highest number of traffic is detected.

6.3.1 The basics

This presentation of how the MapReduce system works is at the highest level possible. So let's take a further look at what happens under the hood.

In terms of software components, a MapReduce job is a unit of work, which the client wants to be carried out. The requirements for this job are input data, the MapReduce program and configuration settings. Hadoop executes the job by splitting the job into tasks. There are two types of tasks: map tasks and reduce tasks.

In terms of hardware components, there are two types of nodes (physical machines) that control the job execution process. There is one jobtracker and one or multiple tasktrackers. The jobtracker manages all the jobs on the system via scheduling tasks to the tasktrackers. Tasktrackers execute the tasks of the job

and report back to the jobtracker about the task's progress, which in turn keeps track of the overall progress of a job. This proves that the system can work parallel and very efficient.

Data flow

When input is given to a specific MapReduce job, the framework divides the input into fixed-size pieces called input splits. Hadoop then creates a map task for each of the input splits, because of this, if the input splits are on different machines, the map tasks can be executed in parallel.

When a job has many input splits, it takes less time to process each split than when the splits are smaller in number. So processing the splits in parallel would take less time to finish the tasks. It can, however, happen that the input splits are so small that the overhead of managing the input splits would take a larger execution impact on the system rather than the task itself. That's why the Hadoop file system has a default value for the block size, which is 128 MB (at the time of writing this thesis). This value can be set via properties in the `hdfs-default.xml` configuration file for the entire cluster or specified when each file is created.

```
<property>
    <name>dfs.blocksize</name>
    <value>134217728</value>
</property>
```

It's important to note that the jobtracker doesn't necessarily schedule tasks to nodes where the input data is already stored on the HDFS file system. It's of course highly desired that this would be the case, otherwise this would come at the cost of sacrificing a lot of network bandwidth. That's why Hadoop has a built-in data locality optimization system to overcome this problem. The principle is simple, the basic rule is to assign tasks to the nodes that cause the least amount of network traffic. There are three possible scheduling assignments, either the data block is stored onto the same node the task is scheduled on, either the data block is stored onto a node in the same rack the task is scheduled on, or the data block is stored onto a node in another rack the task is scheduled on. The different types of assignment can be seen on figure 5.

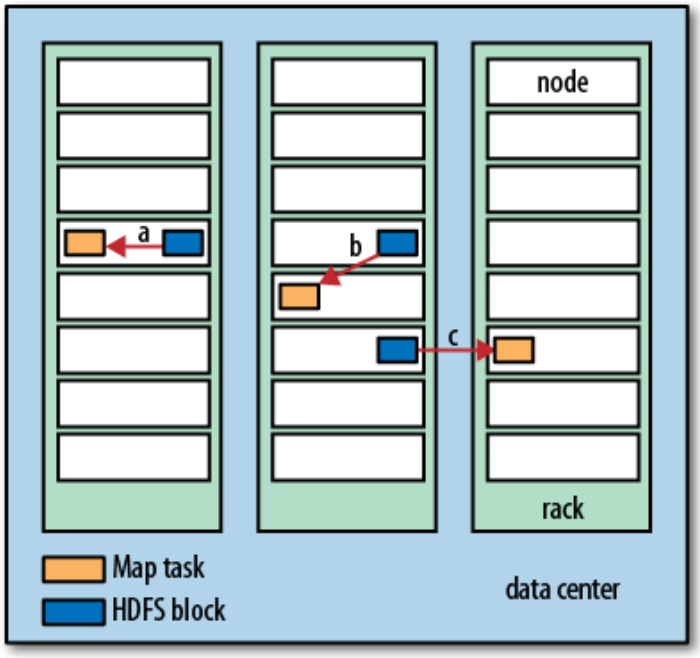


Figure 5 - Possible job scheduling assignments[4]

Earlier we saw that the map task has an input and an output. The input is stored on the HDFS file system. But the output of a map task is not stored onto the HDFS file system, instead it is placed on the local file system of the node it was executed on. The reduce task uses the output (stored locally) of the map task for its input and processes it to a specific output. In other words the map's output is intermediate output, it just needs to be stored until the reduce job is finished and then it's no longer required. Later on we will talk about the HDFS file system and what its benefits are compared to a local file system. But one big feature is replication of data to provide backup in case of hardware failures. So storing this intermediate data onto the HDFS file system would create a massive amount of overhead.

Unfortunately reduce tasks can't benefit of the data localization optimization feature. Therefore we will take a look at how Hadoop transfers the intermediate output of the map tasks to the input of the reduce tasks. We have two possible situations that we will discuss. One is where all the map outputs are transferred to one reduce input. The other is where all the map outputs are transferred to multiple reduce inputs. The number of reduce tasks can be set independently for a given job.

[4] White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

Map outputs to one reduce input

When the map tasks are finished processing, output is written to the local file system. The output must be transferred to the node where the reduce task is running. Once the data is transferred, the data is merged so it can be used as input for the reduce function. The data flow is illustrated in figure 6. The light blue boxes represent individual nodes, the thick red arrows show data transfer across the network.

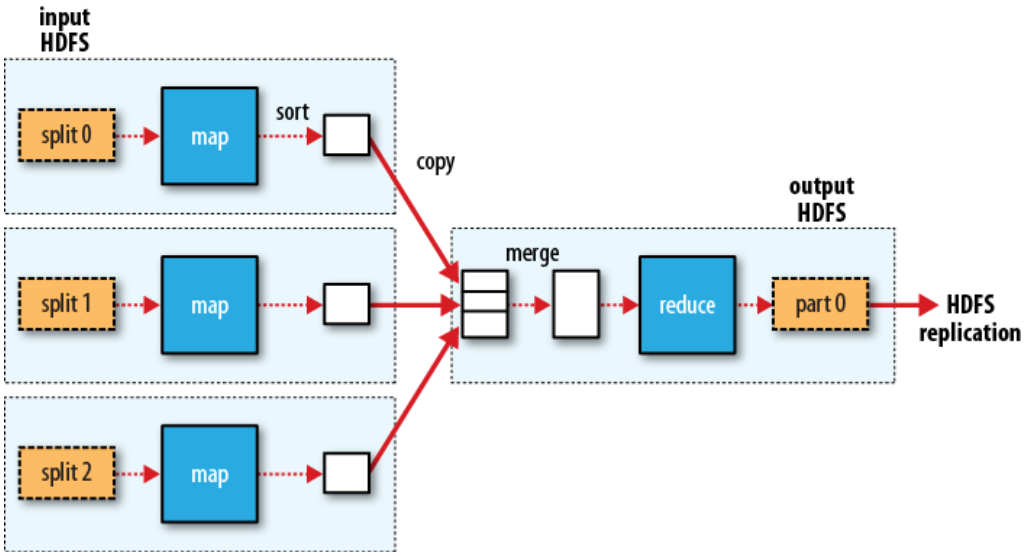


Figure 6 - Map outputs to one reduce input workflow[5]

Map outputs to multiple reduce inputs

The second possibility is that there are multiple reducers defined. We would want the reducers to receive an equal amount of input, so the workload is well balanced. Therefore each mapper divides its output into partitions. When we have two reducers, each mapper will create two partitions containing the output data. There can be many key-value pairs in each partition, but the records for a specific key-value are only stored in one partition. This means that the output data is defined once over two partitions and not copied. The way this partitioning is done can be defined through a user-defined partitioning function or the user can accept the default partitioning system.

In figure 7 we can see that the mappers partition the data through the sort function. Then each partition is transferred across the network to the reducers.

[5] White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

Each reducer receives one partition from each mapper. Then the data flow is the same as with one reducer, the inputs are merged so they can be used by the reduce function.

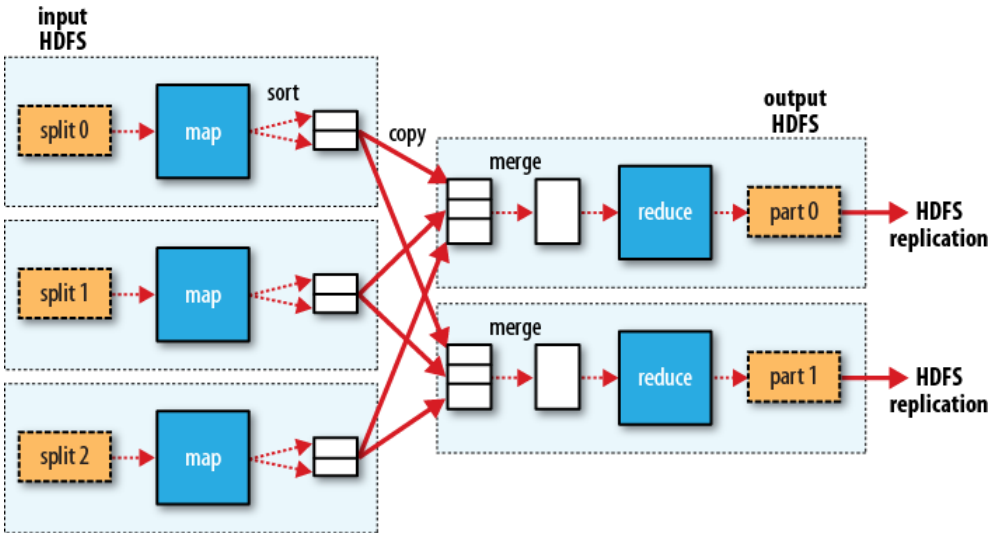


Figure 7 - Map outputs to multiple reduce inputs workflow[6]

Combiner functions

We already mentioned the data localization optimization that is automatically applied by Hadoop to minimize data traffic across the network. But there is another way of minimizing the traffic. The user can specify a combiner function that runs on a mapper's output, the output of the combiner function is then used for the input of the reduce function. One very important note however is that the user needs to decide whether he/she can use a combiner function for its reduce tasks. Let's use an example to make this clear.

Suppose we have some data from the Belgian universities. The data was processed by two maps that were written to retrieve the highest student score (in percent) from each university. Following output was created by the mappers.

mapper output 1

```
(Vives, 92.3)
(Howest, 94.7)
(VUB, 91.8)
```

[6] White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

mapper output 2

```
(KULeuven, 95.1)
(HoGent, 93.5)
```

We would now write a reduce function that calculates which university has assigned the highest score to a student. So after the reducer has merged all the mappers' outputs, the reducer's input would look something like this (it doesn't need to look exactly like this, how these key-value pairs are defined is completely up to the user).

reducer input

```
([Vives, Howest, VUB, KULeuven, HoGent], [92.3, 94.7, 91.8,
95.1, 93.5])
```

And after the reducer processed the data, which means calculating what the highest score was that a school assigned to a student, would be as follows.

reducer output

```
(KULeuven, 95.1)
```

Now because the reducer function is looking for a maximum value in a list of double values, we can use a combiner function on each mapper's output that uses the reducer function to reduce the output's data size. After applying the combiner function the mappers' outputs would look like this.

mapper output 1

```
(Howest, 94.7)
```

mapper output 2

```
(KULeuven, 95.1)
```

There is less data to be transferred across the network to the reducer (think about the benefits when we would use gigabytes of data). So, in short, the combiner function will run twice for each mapper's output, but it will reduce the amount of data transferred. This provides increased efficiency and a reduction of network traffic.

With the combiner function applied, the reducer's input now would look like this.

reducer input

```
([Howest, KULeuven], [94.7, 95.1])
```

And after processing the data in the reduce function we would get the exact same result as before:

reducer output

```
(KULeuven, 95.1)
```

And this is the important thing to note that you can use a combiner function in this example, because we are calculating a maximum value. But think about what would happen if we would calculate the average of these highest scores. Without a combiner function we would calculate the result based on the values below.

```
([avg1, avg2, avg3, avg4, avg5], [92.3, 94.7, 91.8, 95.1, 93.5])
```

Score average = 93.48

Now if we apply a combiner function to reduce the network traffic, we would calculate the result based on these values:

```
([avg1, avg2], [92.9, 94.3])
```

Score average = 93.6

As can be concluded a different result is calculated depending on using a combiner function. So the designer of the MapReduce has to think carefully about using a combiner function. The requirement for using one is that the output of the reducer would be the same when not using combiner function.

6.3.2 The process of a MapReduce Job run

In this part of the MapReduce chapter we will take a look at how the Hadoop framework processes a job. How does Hadoop distribute the tasks to the nodes? How do the nodes get the data to be processed? Or even the program to run? These questions will be answered in this chapter.

Development of the Hadoop framework has been going on for quite a while now and since version 2.0 there is a new implementation of the MapReduce component. It is called YARN and was developed by a group at Yahoo!. This update of the MapReduce algorithm was necessary because the first version was hitting scalability bottlenecks for very large clusters.

In the classic MapReduce you have one jobtracker process that has two main functions: jobscheduling and task progress monitoring. But in YARN this jobtracker and all its related tasks are split up into separate entities. It defines these two roles into two independent daemons[7]: a resource manager and an application manager. The biggest difference with the classic MapReduce is that each job has a dedicated application master.

Because YARN is the new standard for the MapReduce component in Hadoop, and in our practical application we use YARN too, we will discuss YARN and not the classic MapReduce.

MapReduce on YARN contains 5 components:

- A client node that triggers the job execution, or in other words submits a job.
- A resource manager, which coordinates allocation of resources on the cluster.
- Node managers, which launch and monitor compute containers on machines in the cluster.

[7] Daemon: A program that runs as a background process without user interaction.

- A MapReduce application master, which coordinates the tasks running for the submitted job. Both the application master and the MapReduce tasks are run inside containers, which are scheduled by the resource manager and managed by node managers.
- The Hadoop distributed file system, which is used for sharing job files between machines.

Let's start discussing the workflow of when a MapReduce job is submitted for execution. This workflow is shown on figure 8.

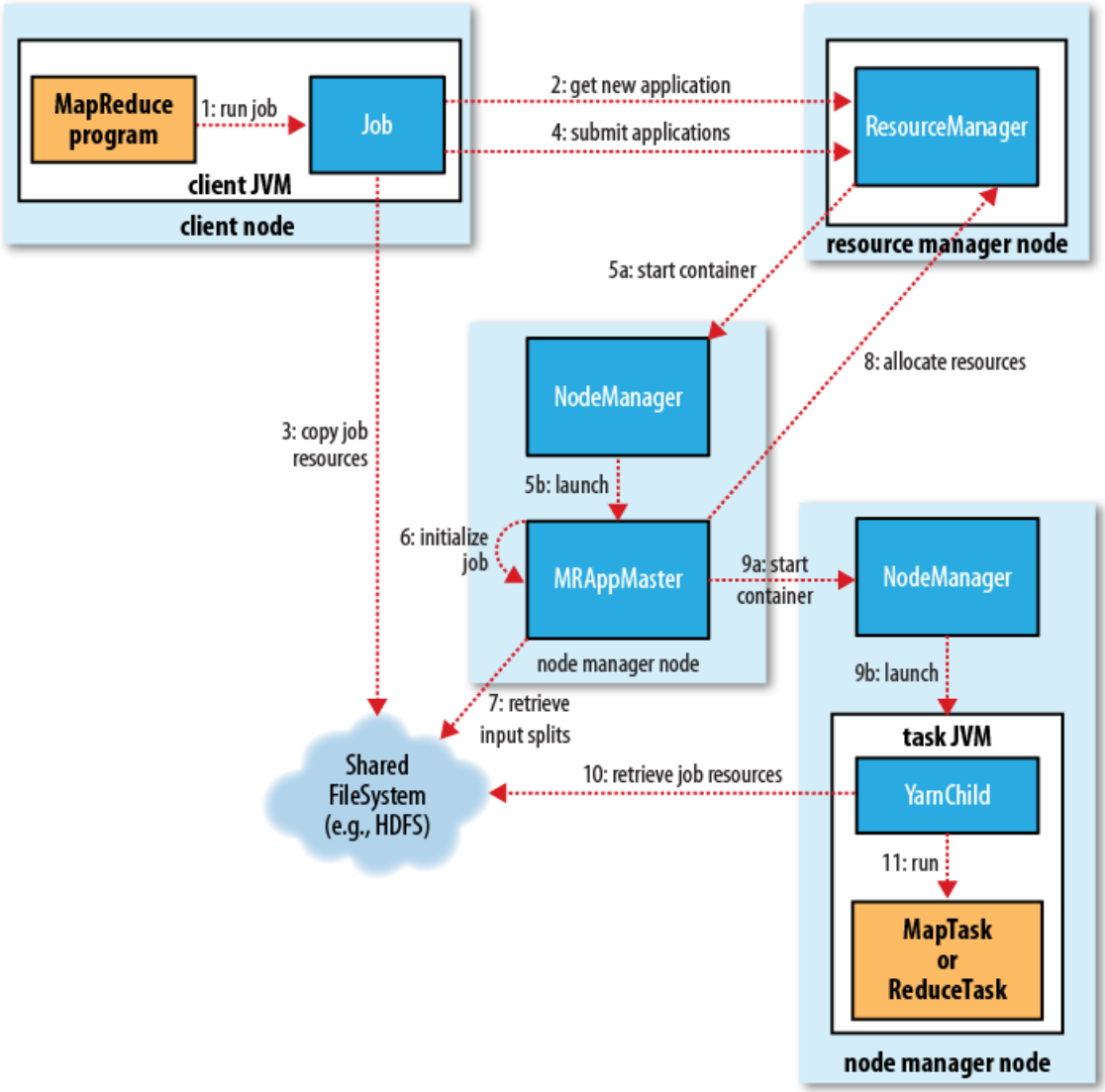


Figure 8 - MapReduce job execution workflow[8]

[8] White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

Job submission

The first part in the process involves the submission of a MapReduce job. We can do this by calling `submit()` or `waitForCompletion()` on the `Job` object in the client node (1). Submitting the job results in a call to the resource manager to retrieve a unique application ID (2). The job client then checks the output specification that is configured on the `Job` object. With this information it calculates the number of input splits for the job and copies job resources (job JAR, configuration and input split information) to the Hadoop distributed file system (3). The last step in the job submission process is to call `submitApplication()` on the resource manager (4).

Job initialization

After the resource manager receives the call to its `submitApplication()`, it hands off the request to the scheduler. The scheduler allocates a container and the resource manager launches the application master's process under the node manager (5a, 5b). The application master is a Java application whose main class is defined as `MRAppMaster`. This class initializes jobs by creating a number of bookkeeping objects that are used to track the job's progress (6). After this, `MRAppMaster` retrieves all the job's input splits that were calculated in the client and stored on HDFS (7). Using these input splits, it creates a `Map task` object for each split and a number of reduce task objects that are defined by the `mapreduce.job.reduces` property in the `mapred-default.xml` configuration file.

```
<property>
    <name>mapreduce.job.reduces</name>
    <value>1</value>
</property>
```

The next step is to determine how to execute the tasks related to the job. The way in which this is done differs a lot from the classic MapReduce. In classic MapReduce the tasks are always running on multiple tasktrackers (if there is more than one tasks), but in MapReduce on YARN, the `MRAppMaster` determines whether the job is very small and could possibly run the job in its own

JVM[9]. Sometimes this is done because the overhead of metadata and allocation of resources on other machines would be greater than running the tasks sequentially on one single node. When this situation occurs, the job is said to be run as an uber task.

The last step in the job initialization process is calling the `setup()` method on the `Job` object to create the job's output directory.

Task assignment

After all these steps the application master can start requesting all the necessary resources for the job's map and reduce tasks from the resource manager (8). All these requests provide information about each map task's data locality, which is necessary to know where the input split reside on the HDFS file system. Using this information the scheduler can make decisions about assigning tasks to the nodes. It always attempts to place tasks on data-local nodes (which are the nodes where the input splits reside on), and if this is not possible, the scheduler will try to assign the task to a rack-local node.

Task execution

When the resource manager's scheduler has assigned a task, the application master can start the container by contacting the node manager (9a, 9b). In preparation of task execution, the node manager localizes the necessary resources for the task, including the job configuration information and JAR file (10). The task is then executed by a Java application whose main class is `YarnChild` (11). The execution of the map and reduce tasks happen in dedicated JVMs, for each task a new JVM is created.

Progress and status updates

The application master keeps track of all the tasks that are running for the job. The tasks report their status and progress back to the application master every three seconds. The application master can make a combined progress status of all the running tasks to give an overview of the complete job progress. The workflow of these status updates can be seen in figure 9.

[9] JVM, Java Virtual Machine: this is a single work unit that executes Java applications.

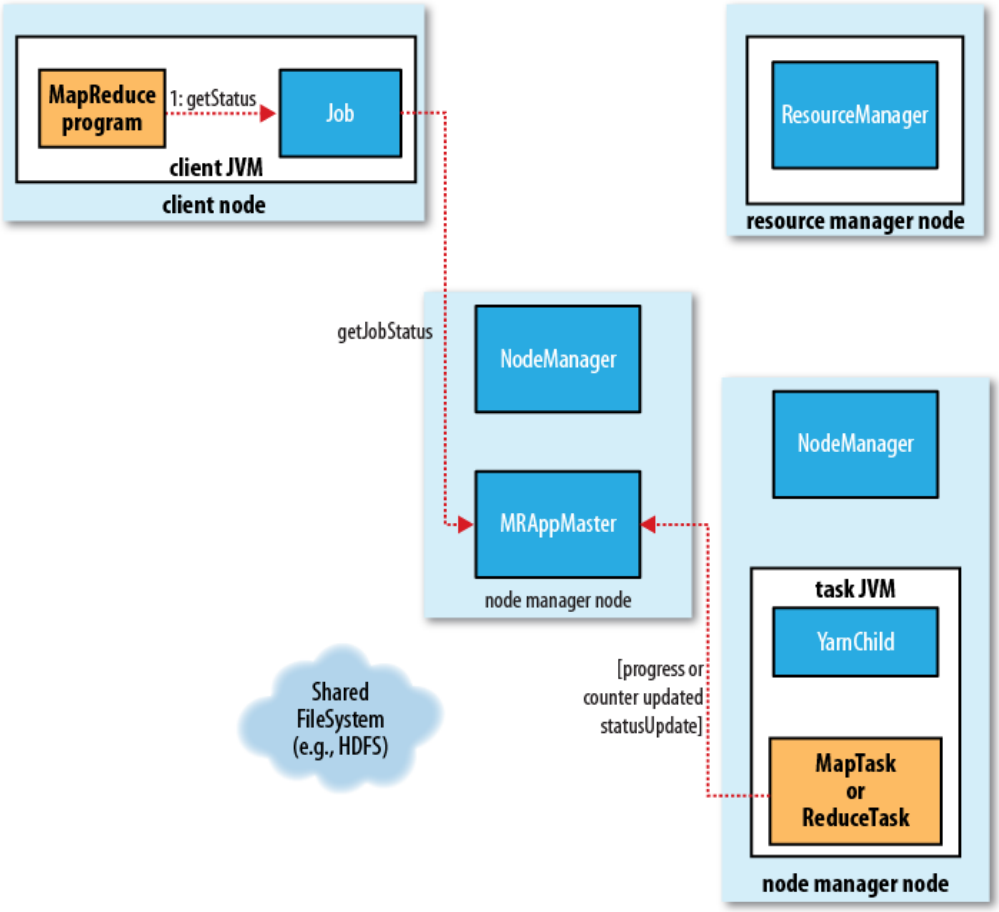


Figure 9 - MapReduce status check workflow[10]

The client node polls the application master every second to receive progress updates for the job. When running a job from the command line, you can see the progress in the shell's output. You can also check the progress of jobs via the web interfaces for the application master. You can first check the resource manager's web UI via connecting to `http://hostname:8088` (which is the default port for the resource manager) and from there you can check all running jobs and redirect to the application master web UI for the job you want to check the progress and status for.

Job completion

The client polls the status every second to the application master, but it also polls every five seconds to check whether the job has completed or not. It does this by calling the `waitForCompletion()` method on the `Job` object.

When the job is complete, the application master and task containers clean up their working state and the `OutputCommiters'` job `cleanup()` method is

[10] White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

called. The job history server then stores all the information about the finished job.

6.3.3 Failures in YARN

Because of all the different components in the YARN MapReduce, we need to consider the possibility of failure for each component. In the following section we will describe which components could fail and what the consequences or solutions are.

Task failure

This is a failure on the highest level possible. Runtime exceptions and sudden exits of the JVM are propagated back to the application master, which marks the task attempt as failed. When the application master registers a hanging task through the absence of a ping over the open channel, the task attempt is also marked as failed.

The maximum number of attempts for a task can be set through the property `mapreduce.map.maxattempts` in the `mapred-default.xml` configuration file. A job will be marked as failed if the maximum percent of map tasks in the `mapreduce.map.failures.maxpercent` property is exceeded.

```
<property>
    <name>mapreduce.map.maxattempts</name>
    <value>4</value>
</property>
<property>
    <name>mapreduce.map.maxattempts</name>
    <value>4</value>
</property>
```

Application master failure

The failure of a job's application master is more critical. Just like tasks, the applications in YARN attempt to execute multiple times in case of failure. By default the application is marked as failed when it fails once, but this property can be set through the property `yarn.resourcemanager.am.max-retries` in the `yarn-default.xml` configuration file.

```
<property>
    <name>yarn.resourcemanager.am.max-retries</name>
    <value>1</value>
</property>
```

The application master sends heartbeats to the resource manager to indicate that it is still running. In the event of a failure, the resource manager will detect this absence of heartbeats and will start a new instance of the application master running in a new container. When this happens the client no longer has knowledge of the application master because a new instance was created, so it needs to locate the new instance. It does this during job initialization, by requesting the resource manager for its new address.

Node manager failure

If a node manager fails, it will stop sending heartbeats to the resource manager, which will remove the node from the resource manager's pool of available nodes. Any task or application master running on the failed node manager will be recovered using the mechanisms described in the above two sections.

Resource manager failure

Failing of the resource manager is the biggest failure that could happen on YARN. Without the resource manager neither jobs nor task containers can be launched. After a failure, an administrator starts a new resource manager instance. It can recover its previous state due to the checkpoint mechanism that saves the resource manager's state to persistent storage. This storage specification can be set via the `yarn.resourcemanager.store.class` property in the `yarn-default.xml` configuration file.

```
<property>
    <name>yarn.resourcemanager.store.class</name>
    <vaue>org.apache.hadoop.yarn.server.resourcemanager.
        recovery.FileSystemRMStateStore</value>
</property>
```

6.4 The Hadoop Distributed File System

The Hadoop framework uses its own designed distributed file system called HDFS. In this topic we will discuss how the Hadoop file system is designed and how it works.

6.4.1 Design

Let's start with a general statement of the HDFS file system: "HDFS is a file system designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware"[11]. The file system is created for storing large files. The streaming data access pattern means that it's build with a write-once, read-many pattern in mind. It's more important to read the whole dataset fast, rather than reading the first record fast. Hadoop can be deployed on any number of machines, and these machines can be of any type, there is no need for high-end server machines, because HDFS contains a built-in fault-tolerance mechanism.

Together with advantages, there will almost always be disadvantages as well. The Hadoop file system is not ideal for low-latency data access or storing small files. But there are several components of the framework that try to overcome these disadvantages. In our project for example, we need a database that provides real-time access. Fortunately the framework provides the HBase module that enhances the application to use with the HDFS file system.

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, MSST2010, May 2010.

6.4.2 Concepts

Namenodes and datanodes

In classic MapReduce we had two big components: jobtrackers and tasktrackers. Well, in HDFS we also have two components: namenodes and datanodes. The namenode always needs to be present, because it's the master node and the datanodes are slaves (or workers). The namenode stores a complete file system tree on its file system. It does this by creating two files: a namespace image and an edit log. In this tree, the namenode stores all the locations of the blocks of all datanodes. Figure 10 gives an overview of this.

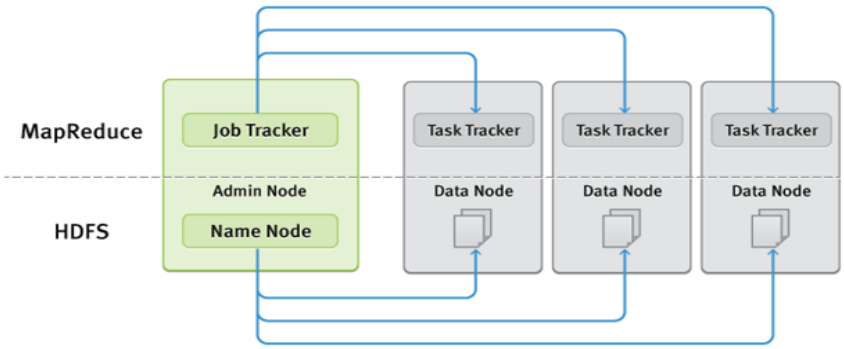


Figure 10 - MapReduce name- and datanodes[12]

The namenode is the most important node, because the storage of the location of data in the file system tree guarantees the retrieval of data on the datanodes. Without it, there would be no way of knowing how to reconstruct the blocks on the datanodes to retrieve usable data. So if the namenode fails, all data is lost. Fortunately Hadoop has provided a way to overcome this huge risk of data loss. One is to make backups of the persistent state of the file system's metadata and write this to multiple file systems. Another is to add a secondary namenode to the cluster setup. The main task of the secondary namenode is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large. It stores a copy of the primary namenode, so in the event that the primary namenode fails, no data is lost.

[12] Greenplum. Hadoop Components. NDM.

HDFS Federation

The namenode stores the location of every file and block in the file system tree. When the namenode is operational, this namespace image is loaded into memory. So when a client requests some data, the access time is lower because the location's metadata is already in the namenode's memory. Considering this fact, how much memory is required for the namenode? We could have millions of files in our cluster. HDFS Federation is a concept introduced in version 2.x of Hadoop. It allows adding multiple namenodes to an existing configuration, so that a portion of the file system namespace could be assigned to each namenode.

This portion that a namenode manages is called a namespace volume and it contains the metadata for the namespace and a block pool containing the blocks for the files in the namespace. Because each namenode manages its own namespace volume, namenodes are completely independent of one another. This means that, when a namenode fails, it does not directly affect other namenodes.

HDFS high-availability

We just covered what happens in case a namenode fails. Because of the replication of metadata and using a secondary namenode the recovery of data location information is guaranteed. But in case of a namenode failure, how long does it take to recover this information? The namenode is still a single point of failure. A new primary namenode must be started with one of the file system's metadata replicas. This new state of cluster setup means that the namenode is in safe mode. Then the administrator must configure the datanodes to use the new namenode. After that, the namenode goes through the process of loading its namespace image into the memory and replacing its edit log. On large clusters this process of starting a new namenode can take up to 30 minutes or more. Version 2.x of Hadoop addresses this problem by adding support for HDFS high-availability. The basic idea behind this solution is the implementation of two namenodes in an active-standby configuration. This way, when a namenode fails, the standby namenodes can take over almost without any interruption or time loss.

6.4.3 Data flow

We already took a look at how map tasks and reduce tasks relate to each other in the previous chapter. Now we will take a look at how this data between nodes is transferred in the HDFS file system. We will look at two cases, how a file is read and how a file is written on the HDFS file system.

Anatomy of reading a file

Executing a client's request to read a file on the HDFS file system requires several steps to be performed. Figure 11 shows these steps and under the image, the steps will be explained.

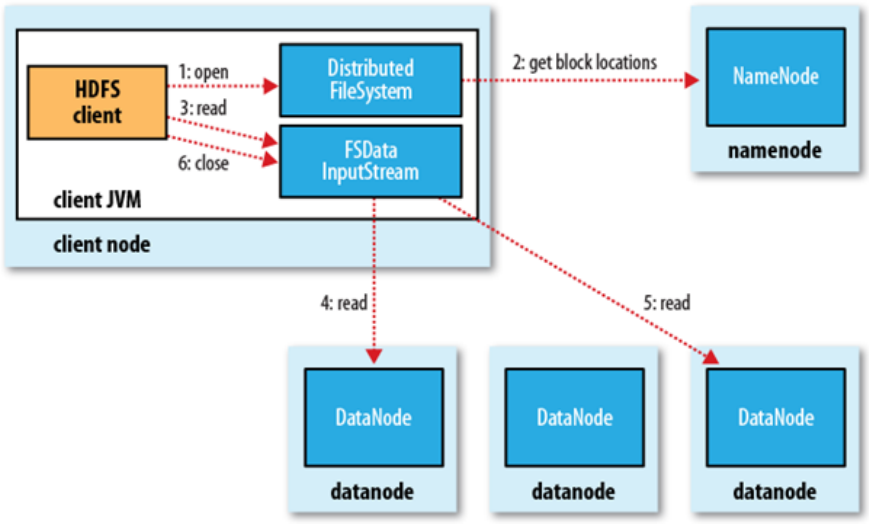


Figure 11 - HDFS file read workflow[13]

The client starts by calling `open()` on the `FileSystem` object (1). In our case it's the distributed file system HDFS so the method will be executed on the derived class instance `DistributedFileSystem`. The `DistributedFileSystem` in turn calls the namenode to retrieve the locations of the blocks for the requested file (2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. After the `DistributedFileSystem` received all addresses it returns a `FSDataInputStream` object to the client. The client can then use this object to read the data from.

[13] White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

The client calls the `read()` method on the `FSDataInputStream` object, which in turn connects to the closest datanode for the first block in the file (3). The `read()` method is repeated until the whole block is read and transferred back to the client, then `FSDataInputStream` closes the connection with the datanode (4). After reading a whole block, `FSDataInputStream` calls the `read()` method for the next block of the file (5) (which in this example is on another datanode). When all the blocks for the file are read, the client closes the connection by calling `close()` on the `FSDataInputStream` object (6).

We can see from this example that it is the client that connects directly to the datanodes, and not through the namenode. This is an important aspect of the design of HDFS. Transferring data this way causes the network traffic to be more spread out, it's more load-balanced. The only role of the namenode here is that it provides the block locations on the datanodes, which (as previously mentioned) are stored in memory, so this happens very fast.

Anatomy of writing a file

In this example we will cover the steps necessary to create a new file, write some data to it and close the file. In this setup the number of replications is set to 3 (default). Which means the HDFS file system stores 3 copies of each file to create a backup in case of failure. You can modify this property in the `hdfs-default.xml` configuration file.

```
<property>
    <name>dfs.replication</name>
    <value>3</value>
</property>
```

Again, first an image (figure 12) is displayed to give a view on how the writing works, then every step will be explained.

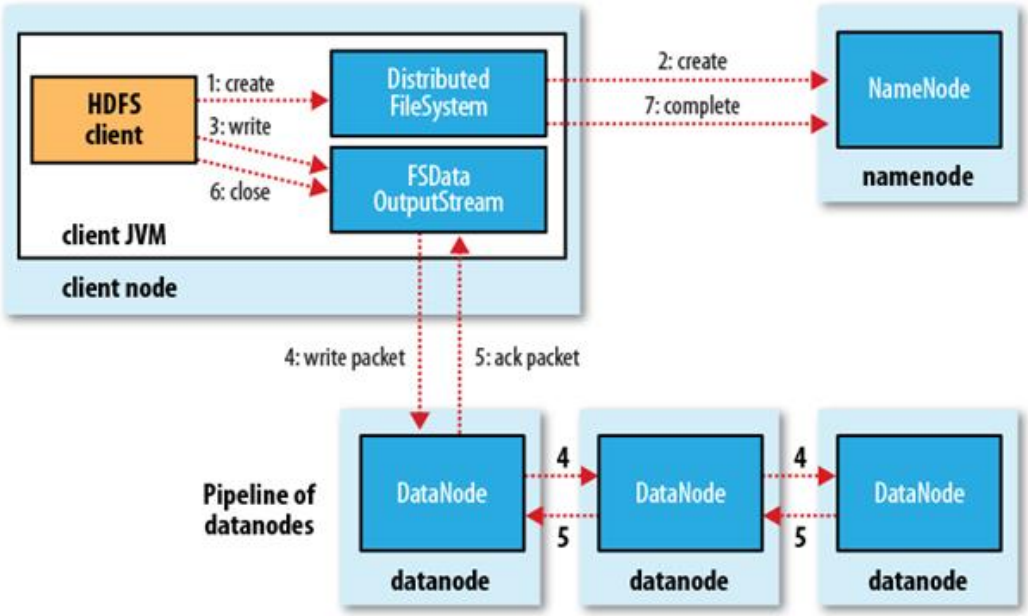


Figure 12 - HDFS file write workflow[14]

The client can create a file by calling the `create()` method on the `DistributedFileSystem` object (1). The namenode needs to know the location of the data on the HDFS file system, so the `DistributedFileSystem` calls `create()` to create a new file in the file system's namespace (2). Once the namenode is done, the `DistributedFileSystem` returns an object of `FSDataOutputStream` to the client for writing to the file.

The client can then call the `write()` method on the `FSDataOutputStream`, which in turn splits the data into packets that are written to an internal queue (3). This mechanism is called a producer-consumer principle. The queue is consumed by the `DataStreamer`, which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store replicas in. This list of datanodes forms a pipeline for writing. Then the `DataStreamer` starts writing the packets to the first datanode in the pipeline (4). After this, it's the task of the datanode to write the same packet to the next datanode in the pipeline. When the last packet is written onto the datanode's HDFS filesystem, the datanode sends an `ACK` (acknowledge) response back to the previous datanodes (5). When the first datanode is reached, it sends all `ACK` responses back to the `DFSOutputStream` where it checks if the number of `ACK` responses are correct.

[14] White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

When the client is finished writing data, it closes the connection by calling `close()` on the `FSDataOutputStream` object (6). Once all remaining packets are flushed to the datanode pipeline, the client calls the `complete()` method to indicate that the file is written on the datanodes (7).

6.5 Hadoop input and output

6.5.1 Serialization

Because the Hadoop framework is built upon a distributed file system, the transmission of data between different nodes in a cluster should be as fast and reliable as possible. For this transmission between nodes, Hadoop uses remote procedure calls or RPCs. This protocol uses serialization and deserialization to convert messages into binary streams. Serialization is the process of converting objects into byte streams. Deserialization is the opposite of serialization.

There are 4 requirements to the design of the Hadoop remote procedure call serialization format:

- Compact: the messages sent by the RPC should be as small as possible to reduce network traffic.
- Fast: the whole communication in the distributed file system relies on this protocol, so it is of vital importance that there is as little performance overhead as possible.
- Extensible: the protocol should be designed with future updates and changes in mind. Newer versions of the protocol should be backwards compatible.
- Interoperable: Hadoop is designed to work on a variety of machines, with a variety of operating systems. The protocol should be designed in a way that it supports all types of machines.

Hadoop has created its own serialization format that is called `Writables`. They apply to the first 3 requirements mentioned above, but because Hadoop is written in Java, the `Writables` format lacks the interoperability requirement. Let's take a look at how Hadoop implemented this format.

The Writable interface

To define all different datatypes for this format, Hadoop created the `Writable` interface that defines an interface for creating `Writable` classes. It's these classes that can be used for input and output values and keys for type parameters of MapReduce methods. It just has two methods for writing its state to a `DataOutput` binary stream and for reading its state from a `DataInput` binary stream.

```
public interface Writable {
    void write(DataOutput out) throws IOException;
    void readFields(DataInput in) throws IOException;
}
```

Let's take a look at a small example on how the serialization process works. `ShortWritable` is an implementation of the `Writable` interface. It represents a short value which is normally two bytes in size. First we create a new `ShortWritable`.

```
ShortWritable shortValue = new ShortWritable(7);
```

Now we can write a helper method to check the serialized form of the `ShortWritable`. We wrap a `ByteArrayOutputStream` in a `DataOutputStream` to retrieve the bytes in serialized stream.

```
public static byte[] serialize(Writable value) throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dataOut = new DataOutputStream(out);
    value.write(dataOut);
    dataOut.close();
    return out.toByteArray();
}
```

Now we can check this byte size with a test method.

```
byte[] bytes = serialize(shortValue);
System.out.println("Byte size of ShortWritable: " + bytes.length);
```

This method will print the following result.

```
Byte size of ShortWritable: 2
```

Writable classes

Hadoop has created a number of `Writable` classes that conforms the `Writable` interface. The most common `Writable` implementations are displayed in table 1.

Table 1 - Common Writable implementations

Java primitive datatype	Writable implementation	Serialised
Boolean	BooleanWritable	1
Byte	ByteWritable	1
Short	ShortWritable	2
Int	IntWritable	4
Float	FloatWritable	4
Long	LongWritable	8
Double	DoubleWritable	8

All these wrapper[15] classes provide a `get()` and `set()` method for retrieving and storing the value.

But the Hadoop library provides some more implementations of the `Writable` interface. The most important are `Text` which is a `Writable` implementation for the equivalent of the string datatype, `BytesWritable` which is a wrapper for an array of binary data and `NullWritable` which has a zero-length serialization. The latter can be used in situations where you don't need a key or value in a MapReduce function.

[15] A wrapper class is a class that wraps a primitive data type into a custom object. This is used to work with primitive datatypes in other ways and to provide extra functionality e.g. datatype conversions.

7 HBASE

Relational Database Management Systems (RDBMS) are widely used for a different variety of solutions. But as mentioned before, they do not fit for all solutions. This is where HBase comes in. Since the setup has to process a big amount of data, a regular RDBMS is not suited for it. HBase is written in the Java programming language. This means every workstation that runs HBase needs to have Java set up.

The biggest differences between HBase and a usual RDBMS can be found in HBase's architecture. The RDBMS does not support scaling but the HBase database does by splitting all rows into regions. These regions are then hosted by exactly one datanode. This means that the load is divided and adding an extra node means that an extra region (with its rows) can be added.

HBase does not store data in column-oriented groups like a typical RDBMS does. It does store data in a column-oriented format on disk but not like the traditional columnar databases. These traditional ones excel at providing real-time access to data, HBase provides key-based access to a specific cell of data or a sequential range of cells.

Another key difference is that HBase does not use usual query solutions like SQL or NoSQL. Instead, queries are performed in the form of commands. These can be instructed through the Java API or through the command shell.

Considering scalability, there's also a big difference between RDBMSs and the HBase database. When regarding RDBMSs, adding extra storage or performing table-wide queries is far from efficient. Analytical databases, like the HBase database, however, can store thousands of terabytes and perform huge queries that scan lots of records or entire tables.

The following chapters will describe what's typical for the HBase architecture and how it works.

7.1 Data Model

The data model differs from the one used in a traditional RDBMS. In this chapter, all the core parts of this model will be explained.

Table

A database always consists of tables, these are hierarchy-wise the highest collection of data. These tables consist of columns and rows.

Rows

A row in HBase is somewhat different from the one in a RDBMS. It consists of a row key and one or more columns with their values. This represents a key-value map in which the row key is the key and different values are attached to this key.

What's also different is that rows are sorted by row key in alphabetical order instead of numerical order. Table 2 shows the effect of this difference.

Table 2 - RDBMS versus HBase key sorting mechanism

RDBMS	HBase
Row-1	Row-1
Row-2	Row-10
Row-10	Row-2

As can be concluded from this table, the sorting order will have a big impact on how complete rows will be sorted in a database. This has to be taken into account when designing the row key. This creates the opportunity for very useful row key design patterns. One that is used a lot is the use of the website domain name. If you, for example, store data from Google (com.google.www and com.google.docs) and Bing (com.bing.www), the data from Google will be grouped together instead of being sorted by the first letter of the domain name.

This row key pattern results in the following sorting order:

- com.google.www
- com.google.docs
- com.bing.www

Instead of:

- docs.google.com
- www.bing.com
- www.google.com

This kind of sorting makes the table optimized for the scans, which will be explained later. Related rows can be stored near each other which results in boosted performance and efficiency. What should be taken into account is that HBase stores rows onto a region based on its row key. When a row key is poorly designed the hotspotting phenomenon could emerge. Let's recap that the Hadoop and HBase system runs on different nodes (distributed) and that the rows can be saved on different nodes. Hotspotting means that a big amount of traffic is directed at a specific node (or a few nodes of the same cluster). Due to this amount of traffic, the machine that is responsible for hosting this region gets a lot of requests for reads, writes and other operations. This, in turn, causes performance issues and could even lead to region unavailability.

To prevent this, decent design of row keys is a must. It is logical that rows that need to be together should be together, in other words, in the same region. But it is also important to spread data over different regions across the cluster.

When monotonically increasing row key this hotspotting will happen because all clients will send requests to the same region and then go to the next one and do the same. So avoiding incremental sequences (1, 2 ,3 …) or timestamps is a good thing to do.

There is no universal 'correct' row key, it depends on the kind of data that is needed to be stored and if a timestamp is linked to a row key or not.

Columns

A column in HBase also looks different, it consists of a column family and a column qualifier. These are separated by the use of a ‘:’ character (colon). There can be as many columns as desired in a database, even up to millions. Columns also don’t have type or length boundaries on their values.

Column family

Columns are grouped into column families. This is done to build semantical boundaries between the data and makes it possible to perform specific actions on them like compressing them. All of the columns in a similar column family are stored in the same file, called an HFile. Opposed to columns, when using column families, it is advised to keep the number of families as low as possible.

Column qualifier

To provide the index for a specific part of the data, a column qualifier is added to a column family. These qualifiers may be altered after creation and may be totally different between rows. An example of a column family and qualifier can be seen below.

```
Column family: content
Column qualifier1: content:html
Column qualifier2: content:pdf
```

Timestamp

Each value in an HBase database gets a timestamp. This timestamp identifies a specific version of a value. The standard syntax of the timestamp is conform the time of writing the data on the region server. This timestamp value can be altered as desired.

Cell

As with a regular RDBMS, a cell is a collection of different parts of the data model. With HBase, a cell is represented by the combination of a row, a column family and a column qualifier. As mentioned before, this cell contains a value with its timestamp.

Datatypes

In a traditional RDBMS, datatypes are very important considering the way the database will process the data. In an HBase database, this is much easier.

Row keys, column qualifiers and values are processed as bytes. It doesn't matter what their actual content is, they will always be handled as bytes. It is possible though to add datatypes but this is pure for the user's benefit, the system won't use or even notice them. This means that whatever the user wants can be used as row key for example. It can be a combination of numbers, letters or both. Table names and Column Family names must be human readable and thus printable characters. The timestamp attached to values is a long value and represents the time in milliseconds.

Visual representation and example

Figure 13 gives a visual image of how an HBase table may look.

	Column Family - Personal		Column Family - Office	
Row key	Name	Residencephone	Phone	Address
0001	Bob	777-5-1111 777-5-1234	777-44-25	100 Birdlane
0002	Mary	777-2-2222	777-44-26	100 Birdlane
0003	John	555-1-1111	777-44-27	100 Birdlane
0004	Carl	454-1-5555	666-21-20	101 Birdlane

Figure 13 - HBase table example (myTable)

The first thing to notice is that the rows ordered alphabetically, as mentioned before. Next to that, every value has a corresponding timestamp (here it is not shown to keep the image simple and clear). Two column families can also be seen, `Personal` and `Office`, each with their own columns. These families are the same for each row. This makes searching algorithms way more efficient. If you look for a personal phone number for example, only the `Personal` column family will be processed in this process. Considering column qualifiers, `Name` and `Residencephone` are two column qualifiers for the `Personal` column family.

Another thing that can be noticed is that there are two different `Residencephone` values for Bob (row key `0001`). Since a timestamp is added to

each value, his old and his new phone number can be found. Every value in the table above has `Timestamp1` as timestamp, except Bob's second phone number, this value has `Timestamp2` as timestamp. If no timestamp is given in the searching process, the newest value will be given, being his new phone number in this case.

Here you can see some possible requests for this example and their results.

```
0001 = {
    Personal : {
        Name : {Timestamp1 : Bob},
        Residence Phone : {Timestamp2 : 777-5-1234} },
    Office : {
        Phone : {Timestamp1: 777-44-25},
        Address : {Timestamp1: 100 Birdlane} }
    }
0001, Personal = {
    Name : {Timestamp1 : Bob},
    Residence Phone : {Timestamp2: 777-5-1234}
    }
0001, Personal:Residence Phone = {
    { Timestamp1: 777-5-1111 },
    {Timestamp2: 777-111-1234 }
    }
0001, Personal:Residence Phone, Timestamp1 = { 777-5-1111 }
0001, Personal:Residence Phone, Timestamp2 = { 777-5-1234 }
```

7.2 Data operations

There are four operations which can be performed on HBase's data model, `Get`, `Put`, `Scan` and `Delete`.

Get

A `Get` performed on a table returns the attributes for a chosen row. This operation is performed by using the `get()` method on the `Table` class that is part of the HBase core library. The following command will give all of the data of the first row of the table in figure 13.

```
% get 'myTable', '0001'
```

Put

The `Put` is used either to add a new row to an existing table (thus, with a new key) or update an existing row (with an existing key). This following command is an example of this operation.

```
% put 'myTable', '0001', 'Office:Phone', '777-44-24'
```

This command will add a new phone number to the `Phone` qualifier of the `Office` column family of first row (`0001`). This phone number will be added with the timestamp on the region server when the `Put` is performed. If an existing value needs to be updated, then the timestamp of this value is also provided in the `Put` command.

Scan

A `Scan` can be performed to get all of the data in a specific table at once or to iterate over multiple rows for desired attributes. The following command will give each row in the `myTable` table.

```
% scan 'myTable'
```

Delete

To delete a row out of a table the `Delete` command is used. The following command will delete the second row, and all of its according values, out of the `myTable` table.

```
% delete 'myTable', '0002'
```

These are the four main transactions that can be performed considering data retrieval and data manipulation. Next to those four, there are some other minor commands used with the HBase database.

`Disable` is used to disable a table. If one wants to delete a table or change its settings, the table needs to be disabled first.
`Enable` is used to enable a table again.
`Drop` is used to delete a complete table.

7.3 HBase Schemas

A database is built upon a schema which defines the tables, columns, primary keys and everything else that defines how a database works.

To make changes to a column family, the table must be disabled first according to the command mentioned above. After disabling the table, the `Alter` command can be used to change its name and such. Below, an example can be seen of adding a new and deleting an existing column family.

Add a `Foreign` column family to the `myTable` table:

```
alter 'myTable', {NAME => 'Foreign'}
```

Delete this column family again:

```
alter 'myTable', {NAME => 'Foreign', METHOD => 'delete'}
```

8 CLOUDERA

This chapter will explain what Cloudera is, why we used it and some of its basic functionalities will be discussed.

Cloudera is an American company and creates open source software based on Apache Hadoop. They deliver software packs that automatically install and configure Hadoop and all of the desired framework components with their own Cloudera Manager on top.

There are different solutions available but only the Cloudera Express package will be described here since that was the one used during this project.
As stated on Cloudera's website: "Cloudera Express provides robust cluster management capabilities like automated deployment, centralized administration, monitoring and diagnostic tools."[16]

In other words, it will make it possible to perform all of the necessary tasks to control and manage a Hadoop cluster. When using this system in a complete business it should be mentioned that Cloudera Enterprise would be a much better solution than Cloudera Express.

When Cloudera is installed for the first time, a cluster will be set up (named and configured) and the data- and namenodes will be selected and added to the cluster. After that, all of the services are assigned to the different nodes, here the choice is made which server will be doing what and how the workload will be divided. This is a very important step and balancing the workload equally is crucial.

[16] Cloudera. 2015. Cloudera Express.

The wizard will take you through every step necessary for installing and configuring Cloudera. This set up is easy and every step is well explained. Basic knowledge on Hadoop is required since specific terms are used by Cloudera. After going through the full configuration, the services are distributed and ready to start the work.

A global view on the Cloudera Manager can be seen in figure 14.

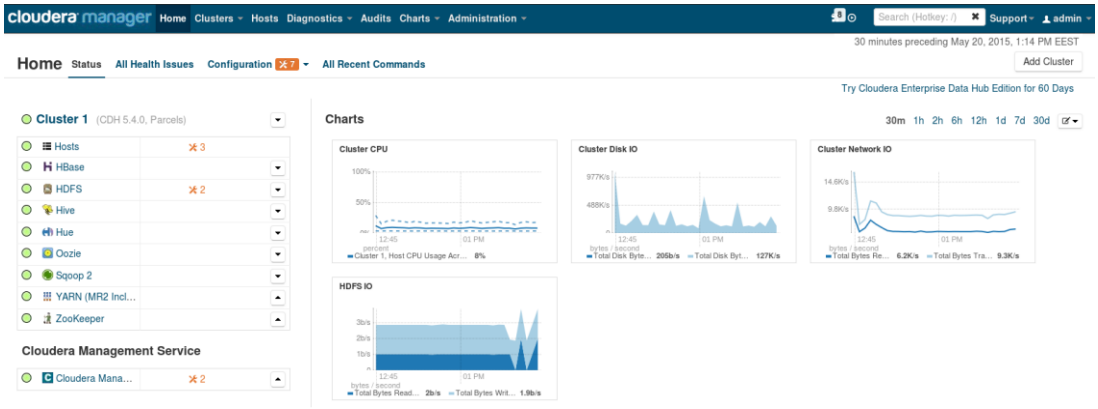


Figure 14 - Cloudera Manager overview

As should be clear from the picture, everything can be monitored and configured using this manager interface. As mentioned before in chapter 6, Hadoop is a framework and consists out of different components, these can be seen on the left side of the page. If something is wrong with one of the components, the according circle in this table will turn red. From there, you'll also be able to start and stop the different components and see which error messages a component has created.

As mentioned before, the main benefit of Hadoop is its high scalability which is reflected in the Cloudera Manager by making it easy to add nodes and clusters. It's really straightforward to then configure these nodes and then add them to a cluster. There is a clear overview of health and configuration issues and even log files are kept and are visible within the Cloudera Manager. Also considering history, monitoring is optimized. As shown on figure 15, the system keeps track of everything that goes wrong or causes problems, with the correct timestamp.

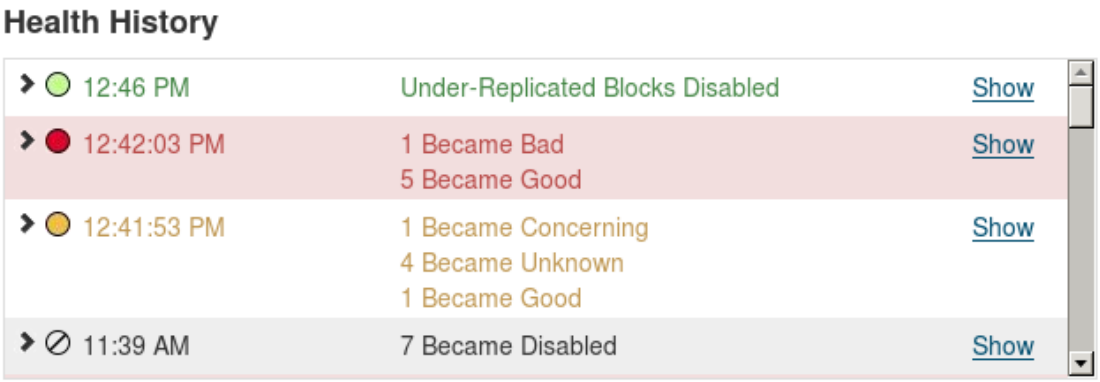


Figure 15 - Cloudera Health History

If we now take a closer look at the HDFS component of the Hadoop set up, every desired statistic and configuration can be seen. These are also very nicely presented on charts on the right side of the page. Below is an example of the storage capacity of the file system.

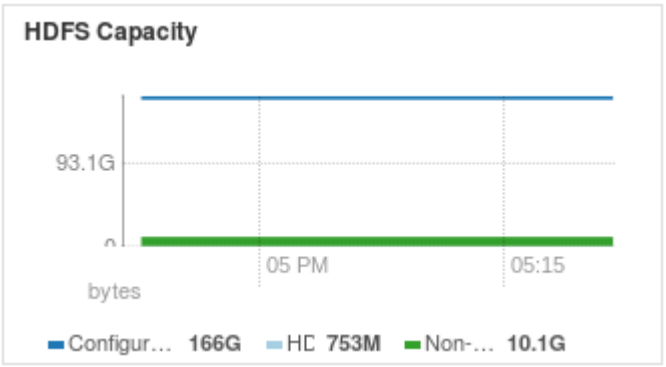


Figure 16 - Cloudera statistics example

If the green and the light-blue bars would nearly reach the darker blue bar, then extra storage capacity should be added. This again proves how easy it is to monitor every part of the system and keep an eye on things that could possibly go wrong.

9 DEVELOPING TOOLS

9.1 Hardware

The whole system is running on a blade server located in the Karelia University of Applied Sciences, Wärtsilä campus. On top of this blade server, three virtual machines were installed. One of them is the master machine, which divides the tasks and such and the two others are slaves which perform tasks. The master is provided with 16GB of RAM memory a 100GB hard drive disk and has 4 cores. The slaves have 12GB of RAM each and also 4 cores.

9.2 Software

In this chapter, all of the used software during the project will be listed and explained. The biggest parts will be the software we used to connect to the hardware setup and the software we used to develop.

Node management

Since the network setup was created by two other students, they made the decisions considering creating a connection with the machines.

To start off, we had to connect to the gateway. The two students, mentioned previously, provided the systems with open source remote desktop connection software named X2Go. This makes it possible to connect to the gateway from wherever you want, it is accessible from any location. What we had to do is install the client version of the software on our own laptops and run it. After configuring the gateway on the X2Go Client (entering the correct IP, desired port, …) we could create a new session. For this, we entered the account details provided by the two students and clicked ‘Ok’ to connect to the gateway. The X2Go Client interface can be seen on figure 17.

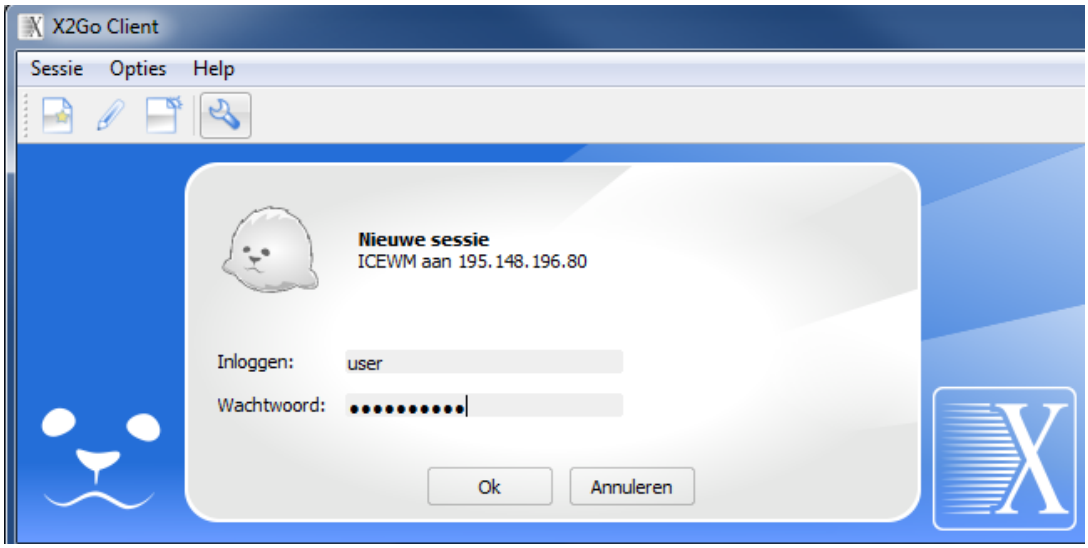


Figure 17 - X2Go gateway connection

After a successful connection with the gateway has been established, the desktop of the gateway will be opened. From this point, another X2Go Client has to be launched to create further connection. This step can be seen on figure 18.

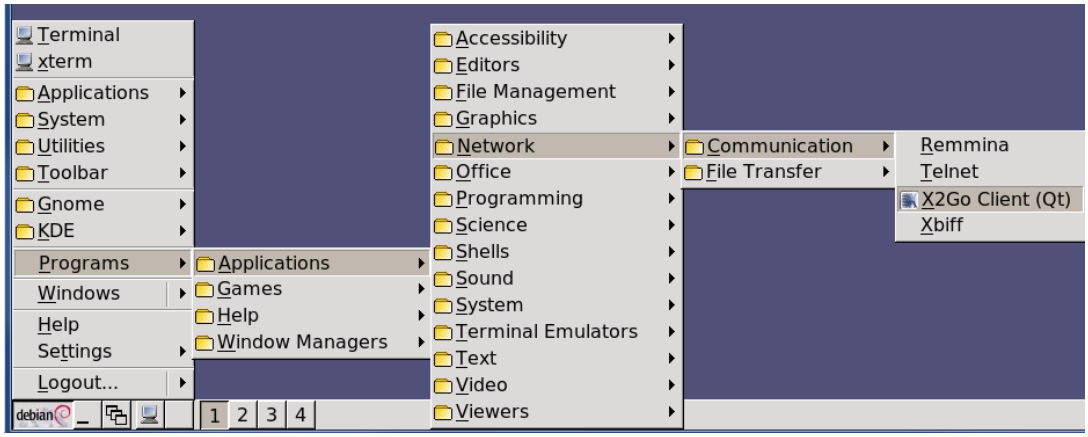


Figure 18 - X2Go connecting to virtual servers

This will result in all of the desired virtual machines showing up and being ready for connection. These connections are established in exactly the same way as with the gateway. This new X2Go screen can be seen on figure 19.

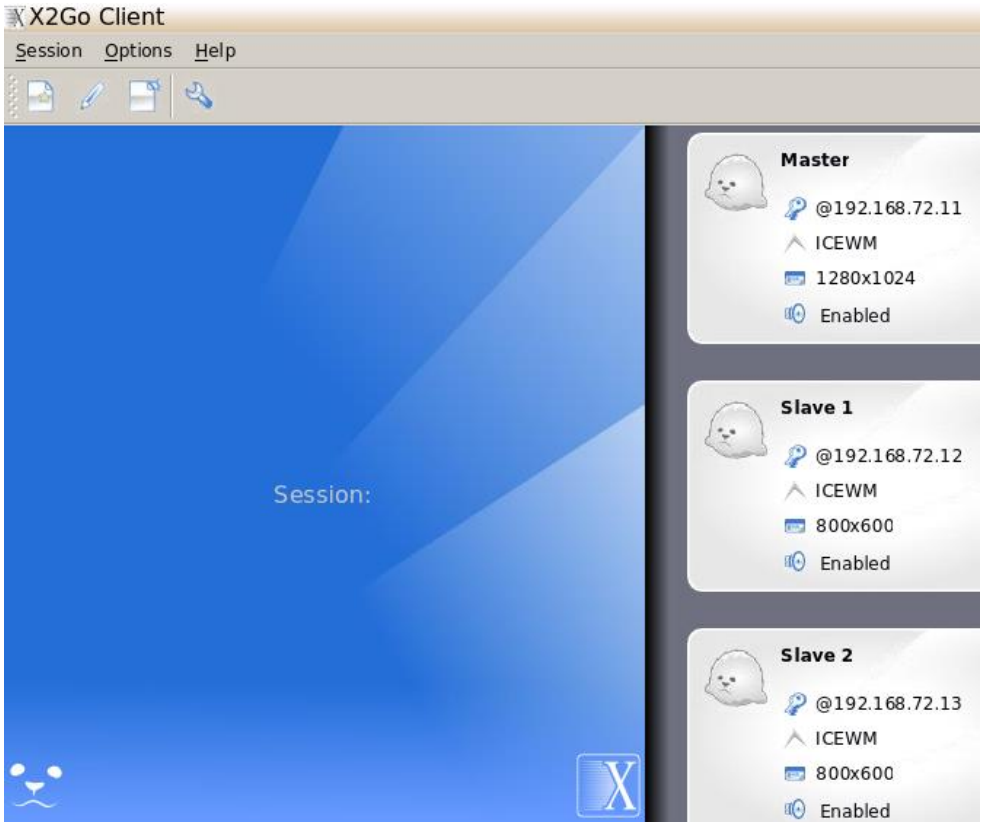


Figure 19 - X2Go machine overview

One can definitely conclude that it is a very simple but yet very secure way to connect to the desired machines. If one doesn't have access to the gateway, then he/she won't get even close to the machines where the complete warehouse set up is configured on.

Developing the Data Warehouse

After obtaining access to the machines where the system runs on, the development itself started. For this, we had to install and set up some different software packages.

The NetBeans program was used to develop all necessary classes for the warehouse. For our application development we used the NetBeans IDE from Oracle. This IDE is open source and thus completely free of charge. It's easy to use and our team is quite familiar with it. For an easy deployment of our application, we decided to install the IDE on our Ubuntu test machines. This way, we can run a MapReduce application immediately without having to copy the JAR file every time to the Hadoop machine.

Because we need to be able to provide our data to clients, we will need a web server. In NetBeans you have various web servers to choose from. We choose the Glassfish server from Oracle, because it can easily integrate with the Jersey framework which we will use for our REST web service. So it's important to note that we are not creating a regular Java application, but rather a web application. This approach has one important consequence, all MapReduce methods we write, will have to be called from within our application, and not from the command shell via the `hadoop` command. So no JAR file is generated when building our application, it is simply deployed on our web container.

Our project will need a variety of external Java libraries. The most important one being the `hadoop-core` library. To make library management easier in our project, we used the Maven management and comprehension plugin. This plugin makes it very easy to manage all your external libraries through the use of a `pom.xml` file, where all the dependencies are declared.

```
<project … >
    <dependencies>
        <dependency>
            <groupId>org.apache.hadoop</groupId>
            <artifactId>hadoop-core</artifactId>
            <version>1.2.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.hbase</groupId>
            <artifactId>hbase-client</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
    ...
</project>
```

When building our project, Maven will automatically download and configure all defined dependencies in this `pom.xml` file. This has one major advantage, when working in a team via a version control system, other developers only have to download the `pom.xml` file, and they will be able to download all the correct libraries, without having to worry if their dependencies or versions are the same as the other developers IDE's.

Next to of all of this, we used GitHub to be able to develop together. GitHub is a version control system which provides a repository for storing any type of file. It

is most commonly used for programming purposes but can be used for any cooperation means.

GitHub can be managed from within the NetBeans application as it has all of the necessary features integrated. To start off, an account has to be created on the GitHub website. This step is followed by creating a repository, which is accessible through an HTTP link. Only one repository is needed per project as all developers use this same repository to work together. Then a connection is made between the NetBeans IDE and the GitHub repository. This is done by logging in with your credentials in the NetBeans application and providing the previously mentioned HTTP link. Then an initial push to the repository of the project has to be done which means that it will be accessible for all of the other developers. Now GitHub is set up for further development. The only thing they have to do is initiate a pull-request from within the NetBeans program to get the initial version of the project.

From then on, changes can be made to this project and, when the changes need to be pushed to the repository, commits have to be made. A commit is actually saving the changes on the local file system, this has to be done before being able to push these changes to the remote repository. Typical commit messages will accompany the commits, these are brief, but to the point, notes to tell the other team members what has been changed and done. These messages are very important because communication is a key part of developing a project in a team. It’s crucial to immediately be aware of what exactly has changed in the project since it may affect what you are doing yourself. The possibility to add commit messages is also integrated in the NetBeans application and makes the program have everything needed to use it for version control.

Project managing

During the development of the project, a project managing tool named Trello was used. This tool handles management in a SCRUM-like way. SCRUM is a definition for a development strategy in which different team members work towards different and equal goals. All of the tasks are defined and divided in the

service and it gives a global overview of what has to be done, what has already been accomplished and who is working on what.

The tool is basically a website which provides an interface optimized for its purpose. It uses some terminology to define different parts, these words will be explained by the use of figure 20, which gives an overview of the interface.

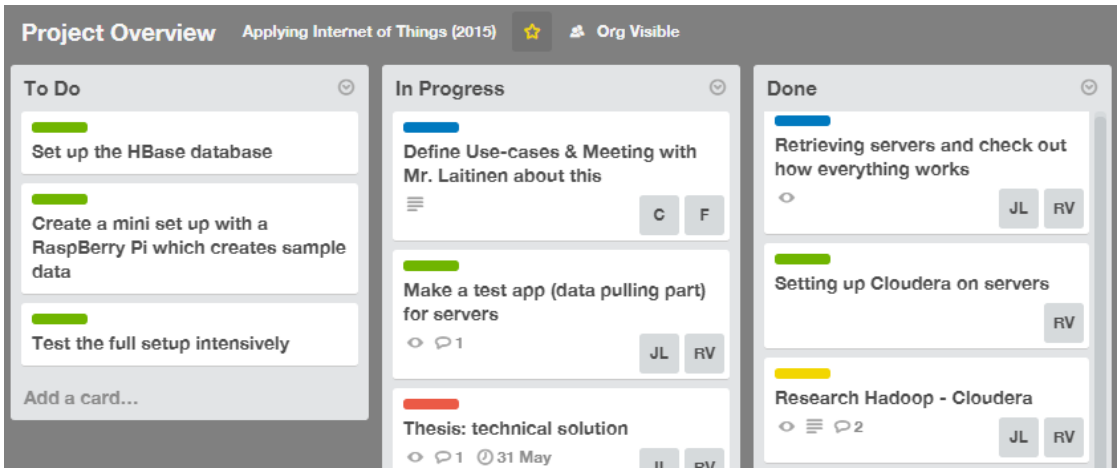


Figure 20 - Trello overview

A board is what you basically see on this picture. It holds all of the different tasks and all of the other data. It can actually be seen as a chalkboard where different things are jotted down on or a bulletin board where notes are attached to.

The board is filled with lists, these are actually divisions to keep everything clear. In our case we used ‘To Do’, ‘In Progress’ and ‘Done’ to keep track of the progress. This division is mostly used considering SCRUM development and is universally accepted as useful and efficient.

On these lists, cards can be found. A card represents a task and its title basically defines what it’s about. Next to that labels can be added to a task, these are the coloured bars on the picture. The function of a label is to define the purpose the task serves. In our project we used some self-defined colours for different tasks.

- Blue: general tasks, tasks which are no part of development but are important though.
- Green: development tasks
- Red: tasks considering the thesis
- Yellow: research tasks

These colour labels are used for organizing everything and to keep the board clear for everyone. A team member can be added to a task so everyone knows what he/she has to do. The tasks can easily be dragged from one list to another and comments and files can be attached to them too. This tool is free to use and provided us with all of the functionality needed to follow up the project in a productive manner.

10 PRACTICAL APPLICATION

In this chapter everything we have done to actually create our application will be described.

10.1 Network setup

On figure 21, the setup of the network and hardware we worked with can be seen.

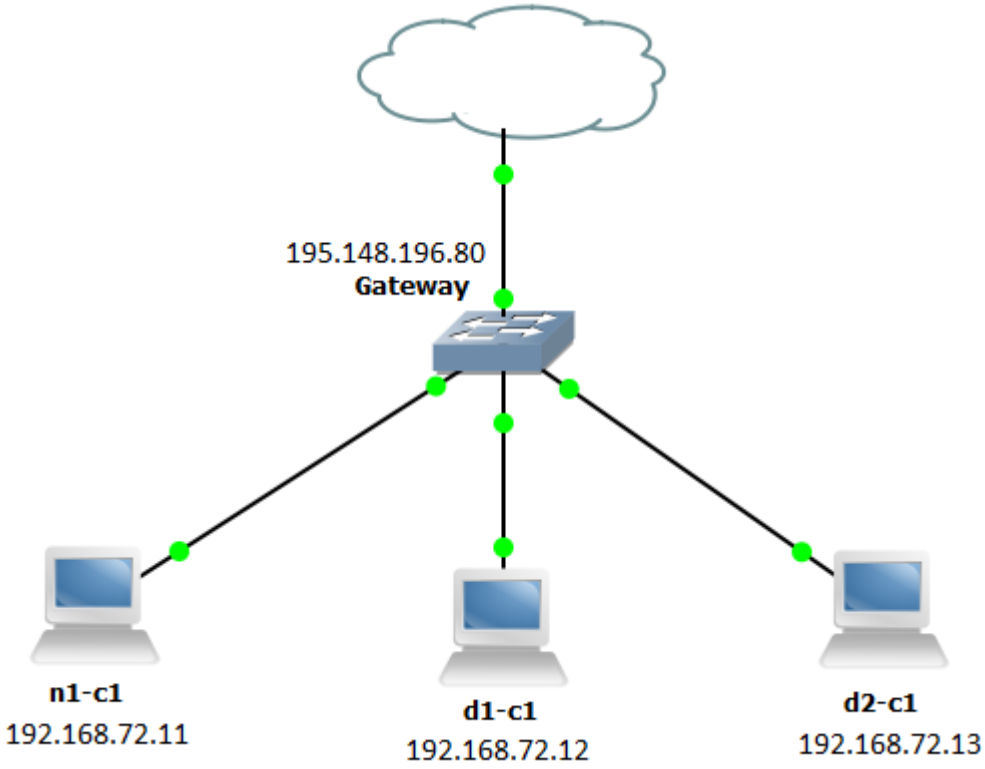


Figure 21 - Network setup

Let's start from the white cloud, this is where we were located considering the development. This cloud represents 'the Internet', we worked from the outside and connected to the gateway on the IP address `195.148.196.80`. After we connected to the gateway with the use of a username and password, we could connect to our working setup which consists of a master (namenode) and two slaves (datanodes). The gateway is used to protect our setup from undesired

external access. The host `n1-c1`, is the namenode of the setup and the other two are datanodes.

This setup was provided to us by two Finnish students (Tiainen Henri and Janne Puustinen) which have made a thesis on this in which they describe how they have set this up. We'd like to refer to their thesis for more information on this network setup, which is not published.

10.2 Cloudera installation

For our Hadoop deployment, we used Cloudera as it will simplify installation and configuration of Hadoop a lot. The core component of a Cloudera installation is the Cloudera Manager and to install this, 4 other components are needed and must be installed correctly on our system.

- Oracle JDK
- Cloudera Manager Server and Agent Packages
- Supporting database software
- CDH and managed service software

Cloudera offers three different installation paths to use for deploying Cloudera Manager. Each has its advantages and disadvantages. These different installation paths can be seen in table 3.

Table 3 - Cloudera installation paths

Installation path	Description
Automated installation by Cloudera Manager	This is the easiest and fasted way to deploy Cloudera on a cluster. Cloudera Manager automates the process of installing Oracle JDK, embedded PostgreSQL database, Cloudera Manager Server and Agent Packages, and managed service software. This path is only recommended for demonstrations purposes only, it's not suited for large scale solutions.

Manual installation using Cloudera Manager Packages	The user needs to install the Oracle JDK, Cloudera Manager Server and embedded PostgreSQL database packages on the Cloudera Manager Server host. The user has two options in this install, manual install or automated installation. This is the preferred method for installing Cloudera and is suitable for production environments as it is scalable.
Manual installation using Cloudera Manager tarballs[17]	This installation path is exactly the same as the previous, only instead of installing the components as packages they are installed as tarballs.

In our setup, we used the 'Manual installation using Cloudera Manager Packages' path. We needed a scalable, production-ready environment. This is actually not suitable for testing, but therefore we created multiple test environments as virtual machines that run on our consumer grade laptops. Note that all installations will happen on machines running Ubuntu 14.04.

Installation and configuration of databases

Cloudera Manager uses a number of databases and data stores to store information about configurations as well as history information about health checks and task progress. The easiest way is to install the embedded PostgreSQL database. You can however install a database of your own choice, but to ensure optimal configuration we decided to go with the embedded database. You can specify on what host you want the database to be installed, but because our cluster setup is so small we will install the database on the master host (the namenode). To install this PostgreSQL embedded database we can start by installing the package through the `apt-get` command in Ubuntu:

```
% sudo apt-get install cloudera-manager-server-db-2
```

[17] A tarball is an archive file to collect and compress various files for distribution or backup purposes.

After a successful installation we need to start this database using the following command:

```
% sudo service cloudera-scm-server-db start
```

Cloudera Manager Repository Strategy

In this step, we need to define a repository strategy to manage and install all the necessary components for Cloudera. We do this by defining a repository URL on the Ubuntu system and updating all installed packages on the system.

First we need to create a new `cloudera-manager.list` file inside the `/etc/apt/sources.list.d/` directory with the following content.

```
# Packages for Cloudera Manager, Version 5, on Ubuntu 14.04
x86_64
deb [arch=amd64]
http://archive.cloudera.com/cm5/ubuntu/trusty/amd64/cm trusty-
cm5 contrib
deb-src http://archive.cloudera.com/cm5/ubuntu/trusty/amd64/cm
trusty-cm5 contrib
```

Now we can update all packages on the system, and because we defined this new repository, we will retrieve all necessary packages for the Cloudera installation on our system.

Installation of Oracle JDK

Hadoop is developed in the Java programming language. Therefore Java needs to be correctly installed on all hosts. It is of crucial importance that all the versions of Java are the same on all machines. Otherwise unexpected behaviour could arise when executing MapReduce applications. Because of these possible consequences, developers also need to make sure that, when they create new MapReduce applications on other test setups, the JDK version has to be the same as well. We installed version `1.7.0_79` on all hosts and on our virtual test machines.

Cloudera Manager Server packages

This step involves the actual installation of Cloudera Manager Server on the master host. We need to make sure that we install these packages on a host

that has access to the database we installed earlier. Fortunately we deployed this database on the master node. For this we need to install two packages via the `apt-get` command.

```
% sudo apt-get install cloudera-manager-daemons cloudera-
manager-server
```

And actually at this point, all the necessary software for Cloudera is installed. However we still have two slave nodes that are not installed yet. In Cloudera terms, the Cloudera Manager Server is the master node (namenode), Cloudera Manager Agents are the slave nodes (datanodes). But these slave nodes need another package to be installed.

Cloudera Manager Agent packages

Next step is to install the required software on our slave nodes so they can be used as datanodes in our Hadoop cluster. In this case we can use the following `apt-get` command that we execute on every slave node.

```
% sudo apt-get install cloudera-manager-agent cloudera-manager-
daemons
```

At this point, we have all of the Cloudera packages installed on all our hosts. But how do the slaves know who is their master? We need to configure the Cloudera Manager Agents to point to the Cloudera Manager Server. We can do this by setting the IP address and Cloudera port of the master host. This is configured in the `/etc/cloudera-scm-agent/config.ini` file on every slave.

```
server_host=192.168.72.11
server_port=7182
```

Starting Cloudera

Now all of the necessary packages are installed onto the machines, we can try to start up the Cloudera Manager. It's important to first start the Cloudera Manager Server and then the Cloudera Manager Agents, as the agents will search for a Server when booting.

We start the Cloudera Manager Server (the namenode) via following command.

```
% sudo service cloudera-scm-server start
```

Then we can start our 2 Cloudera Manager Agents (the datanodes) via following command.

```
% sudo service cloudera-scm-agent start
```

Configuring Cloudera

Now the Cloudera Server and Agents are started, we can continue configuration through the Cloudera web interface. In a browser we can go to the hostname of our namenode followed by port `7180` which is the default port for the Cloudera web interface.

```
http://n1-c1.karelia.fi:7180
```

After this, the user goes through the setup process of configuring Cloudera, which is quite straightforward and therefore we will not discuss this in this paper expect for one aspect of the configuration which is the services in Cloudera.
In the ‘Add Services’ section of the configuration process the user has the option to select a variety of services to add to the setup:

- Core Hadoop (HDFS, YARN, ZooKeeper, Oozie, Hive, Hue and Sqoop
- Core with HBase
- Core with Impala
- Core with Search
- Core with Spark
- All services
- Custom services

In our case we chose the ‘Core with HBase’ option. Now following on this assignment of services, the user needs to define role instances to hosts. Every service contains a few roles (for example the HDFS services contains the

namenode and datanode roles) and this graphical interface makes it very easy to assign roles to specific hosts.

After the configuration setup is complete. You can view role instances for every host separately. On figure 22 an overview of the role instances that are running on our second datanode is shown.

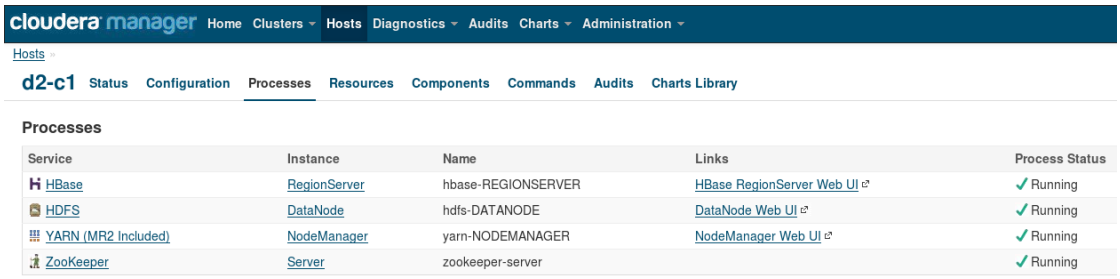


Figure 22 - Cloudera role instances

As we can see, the HBase RegionServer role is running on this node, as is the DataNode role instance of the HDFS service. Datanodes and regionservers should always be one and the same machine. Then we also have a YARN NodeManager running and a ZooKeeper Server. To run ZooKeeper efficiently on a cluster, there should be at least 3 nodes with the ZooKeeper role assigned to. In our cluster we only have three nodes, so every node has this role instance present.

Now we have a complete Hadoop cluster running with three nodes, ready for MapReduce applications to executed on.

10.3 Application

On figure 23, the full setup of the warehouse is shown. This gives an impression off how the whole system looks like and makes it easier to understand how it works. Below, we'll describe every part of it and try to point out any difficulties.

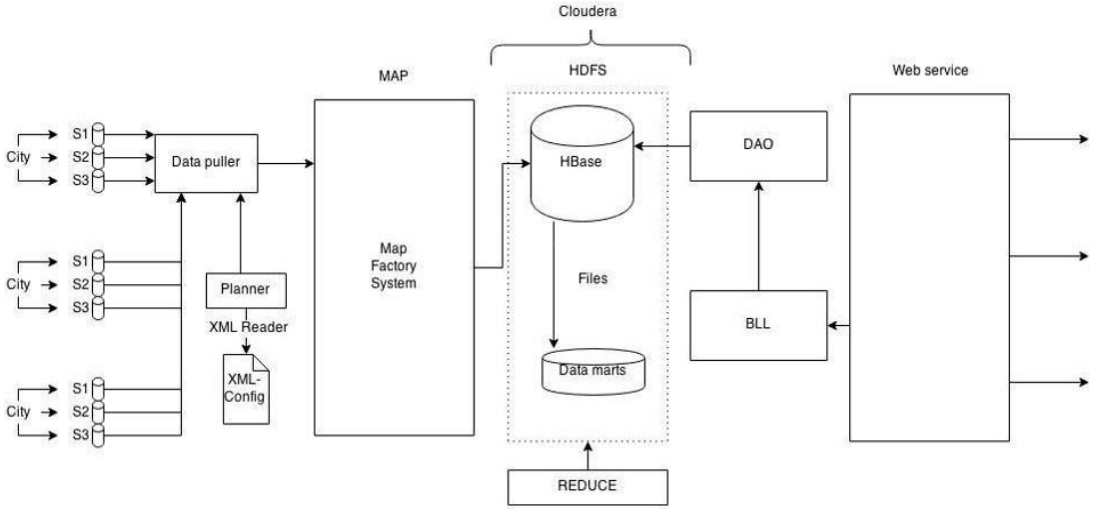


Figure 23 - Application architecture overview

The setup can be divided into three main parts. First the data is retrieved from any kind of source. After that, the data is transformed and stored into the database, which is HBase in our case. Finally access to this data is provided through a web service. As can be seen on the left side, different cities, each with their services, are presented. This scheme is a global visualisation of the implementation. As mentioned earlier, the project is built upon sustainability and can be expanded across different cities. We took this into consideration and thus implemented it this way. During the next chapters, the different parts will be described in detail and we'll explain how the complete system works.

10.3.1 Data retrieval

The first task of the system is actually retrieving the data. To do this, there are some different parts used in our setup. This chapter will discuss every component and explain how they work.

Configuration files

Since all kinds of data will be coming from different sources, we'll have to predefine these sources. This predefinition could for example include a database URL and login credentials, along with the frequency with which the data has to be pulled. Something important to mention is that it would be very unwise to hardcode these configuration details. If, for example, a database's URL changes, the code of the system would need altering which is highly unpractical.

Instead, these details are saved into configuration files that can easily be adapted to the needs of the user or the system. This is done in the XML language that uses tags to separate different configuration details. In our setup, there are two different kinds of configuration files. There is one main file, in our case called `cities.xml`, which holds an overview of every city that uses our system. To this city, another file is linked (e.g. `JoensuuServices.xml`) which holds the actual data necessary to connect to different sources.

This distinction is done to maintain good performance when data is being pulled. The specific services file for a city is the only file that will be processed. To give an example, let's assume that there are two cities, Helsinki and Joensuu. Now in a particular case, data from a service located in Joensuu needs to be pulled. When the services of Helsinki would also be processed, a lot of valuable time would have been lost. Instead, first the city is passed on, in this case Joensuu and then only the services in the `JoensuuServices.xml` file will be checked and not those in the `HelsinkiServices.xml` file. As should be clear by now, we handled the following syntax for a services file: `CityServices.xml` with `City` being replaced by the actual city name.

An example of the `cities.xml` file is shown below.

```
<?xml version="1.0" encoding="UTF-8”?>
<cities>
    <city>
        <name>Joensuu</name>
        <file>JoensuuServices.xml</file>
    </city>
    <city>
        <name>Helsinki</name>
        <file>HelsinkiServices.xml</file>
    </city>
</cities>
```

The first tag is an official tag to define the file as an XML file. Then root element (`cities`) is used to define all of the different cities. This element is followed by a repeated `city` element which defines every different city. This element can be repeated infinite times and holds the name of the specific city and the file in which the different services are located for this city.

The following file is an example of a services file, here created for a Joensuu example.

```
<?xml version="1.0" encoding="UTF-8”?>
<services>
    <service>
        <name>TrafficLights</name>
        <resources>
            <resource>
                <name>Lights</name>
                <datalocation>Lights.xsl</datalocation>
                <scheduling>
                    <type>interval</type>
                    <interval>604800000</interval>
                </scheduling>
            </resource>
            <resource>
                <name>Sensor</name>
                <datalocation>Sensor.xsl</datalocation>
                <scheduling>
                    <type>interval</type>
                    <interval>302400000</interval>
                </scheduling>
            </resource>
        </resources>
    </service>
</services>
```

Since this file is written in the same format as the previous one, there is also a universal XML tag at the top of the file, followed by a root element (`services`). This file will hold all the different services for a specific city, each with their own name and resources. These resources are in turn repetitive and contain the location of where the data resides, in this example they are just worksheets located on the local file system. These data locations would later be database URL’s and then necessary credentials could be added to obtain access to this data location. What’s also stored within these resources is their scheduling system. The scheduling defines how frequently the data should be pulled which differs for different kinds of data. This scheduling element has a type and an interval. We defined these types ourselves and table 4 will give a view on the different possibilities.

Table 4 - Scheduling types

Scheduling type	Description
Initial	This data only needs to be pulled once.
Interval	This data needs to be pulled after every specified timeframe.

First things to notice is that these scheduling types can easily be altered and expanded to the wish of the user. The second things to mention is that the interval measures are expressed in milliseconds. So the example of `604800000` in the interval element means that the data needs to be pulled every week. This value can be a couple of minutes, an hour, a day, a week, a month or even a year or more. The absolute maximum refreshing rate is the max value of a `Long` type in Java which is 2^{63} and represents 292 million years.

Data domain classes

Now that every configuration detail is saved, we need some way to virtually get this configuration into our system. In our project, this is done by creating classes that represent the different key parts of this connection system. These classes are placed in the `fi.karelia.publicservices.data.domain` package. On figure 24, the class diagram of these classes can be seen.

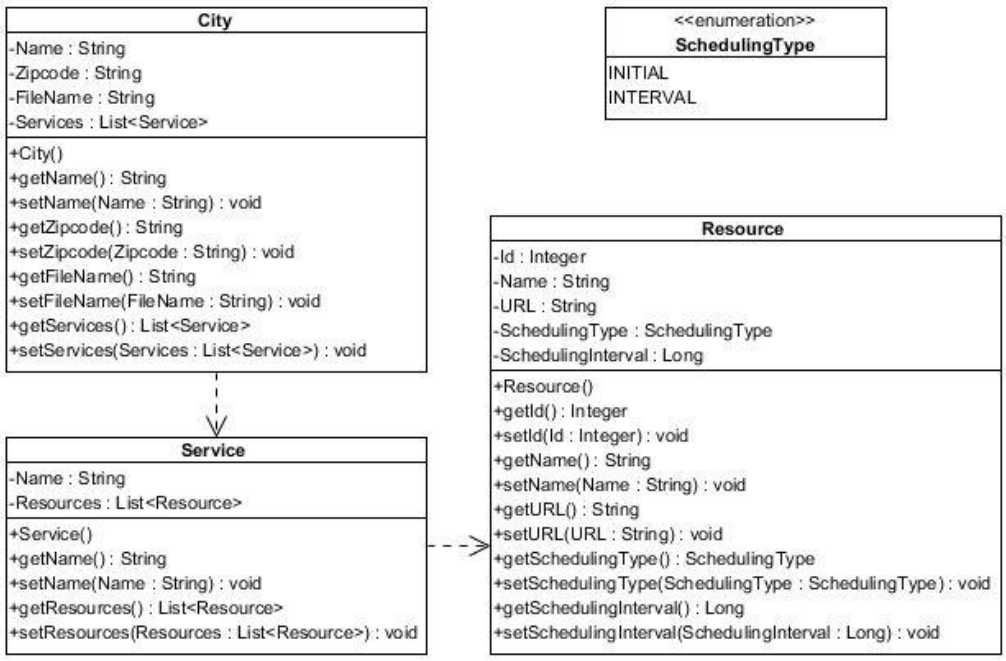


Figure 24 - Data domain class model

As can be seen on the image, one city has multiple services and a service can have multiple resources. Next to that, there are different scheduling types which are defined in an enumeration class.

XML Reader

Now we have the objects representing cities and their services/resources, we need a way to transform the data from the configuration files into these objects. This is done by a class called `XMLReader`. This class was written based on the java implementation for XML parsing[18].

What the `XMLReader` class basically does is converting the data provided in the XML files to actual Java objects which are ready to be used. The reader has two methods up to this point, other necessary methods may be added if the structure of the configuration changes. This would be easy to do but for now the system works completely using the following two methods.

The first method is the `getAllCities()` method that will check the `cities.xml` file and return a list of all of the city objects that have services files. What happens is that the `cities.xml` file is being read and every city element is checked for its name element which contains the name of the specific city. Next to this name element, a file element is attached to define the location of the services file linked to this city. At this point a new city object is created and the name and filename are set for this object. Then the second method is called which is the `addServicesToCity()` method.

What this second method basically does is configure the list of services, each with their appropriate resources, for a defined city. So the city object, to which the correct services will be linked, is given as parameter. What happens here is that the file, of which the name was attached to the city object in the first method, will be read out just like with the `cities.xml` file during the previous method. The XML file is being parsed and for every service element, a service object is created and for every resource element, a resource object is created. During these creations, the correct fields of the objects are configured according

[18] Parsing is the process of converting a string of symbols into the desired objects.

to the data in the tags. The resources are linked, through the use of a `List` object, to the correct service and the same happens for linking different services to a city object.

So this actually is a top-down approach, first a city is created and then everything for this city is constructed and configured appropriately. Every time a new city element in the `cities.xml` file is processed, a new city object is created and added to the list which will be returned at the end of the first method. After this method is completely executed, all of the necessary objects are created and residing in our application. As stated before, the `XMLReader` class is the link between the actual domain objects and how they should be configured according to the XML configuration files.

Data scheduler

Our data warehouse always needs to have the latest data available. In the previous section we explained how scheduling intervals can be defined on resources. Now we'll discuss the implementation of the class that pulls this data at periodic intervals.

This class has two important functions, which we need to explain with care. On the one side, we have the scheduling of resources that require updates defined by intervals. On the other hand, we need to make sure that when these XML files change (because of adding new resources for example) the schedule with the modified resources is updated. Table 5 gives an overview of what should happen if XML resource files are modified

Table 5 - Consequences of resource modification

Change	Action
Resource added	When a new resource is defined in the XML files, we need to check if there is an interval value defined for this resource. If there is, we need to schedule this resource for periodic updates in our `DataScheduler`. If there is no interval defined, we need to pull the data from the resource just once, like an initial pull.

Resource modified	When an existing resource is modified, we need to check all the fields within the resource element. Our scheduled resource object will need to match its properties with the new values in the XML file. But on top of that, we need to modify the existing schedule for this resource if the interval value changed or the interval type has changed.
Resource deleted	If a resource element is deleted in the XML files, we need to remove the resource from the scheduler.

Now these resources are defined at the lowest level in our XML files. One level higher we have the service element. Table 6 provides an overview of what happens when a service element is changed.

Table 6 - Consequences of service modification

Change	Action
Service added	When a service is added, but no resources as child of the service, nothing happens.
Service modified	Modifying a service can mean, a service's name is modified, or it means resources as children elements are modified. When the latter applies we need to perform the actions defined for when a resource is modified.
Service deleted	When a service is deleted, we need to delete all resources that are defined as children of this service element. This is the action taken for when a resource is deleted.

The highest level in our XML configuration files is the cities element. Like mentioned in the previous section, these cities are defined in the `cities.xml` file. Table 7 gives an overview of what happens in case of modifications in this file.

Table 7 - Consequences of city modification

Change	Action
City added	When a city is added, new services and resources defined in the new XML file should be added following the actions mentioned in service added and resource added.
City modified	Modifying a city can only mean two things. Either the city name was changed (the program will register this as a city deleted and added), or the name/location of the corresponding city file has changed. In either way, the old services and resources should be deleted and the new ones added again.
City deleted	When a city is deleted, all the scheduled resources for this city should be stopped and deleted following the action described in deleting a resource.

Now let's take a look at how this is done in Java code. We first created the `DataScheduler` class. And because this class takes care of all scheduling, we should make this class a singleton[19] to ensure no duplicate scheduled resources will be defined. We achieve this by defining one static `DataScheduler` object, creating a private constructor and defining a `getInstance()` method to retrieve the single instance of the `DataScheduler`:

```
public class DataScheduler {

    private static volatile DataScheduler dataScheduler = null;

    private DataScheduler() { … }

    public static DataScheduler getInstance() {
        if (dataScheduler == null) {
            synchronized (DataScheduler.class) {
                if (dataScheduler == null) {
                    dataScheduler = new DataScheduler();
                }
            }
        }
    }
}
```

[19] A singleton is a class that is defined so that only one instance of it can be created.

Now we will always have exactly one instance of our `DataScheduler`. After that, two Java schedulers are needed. One is necessary for starting our periodic check if resources, services our cities have changed. The second scheduler will serve as a container for all the scheduled resources that will need data updates at periodic intervals.

```
private final ScheduledExecutorService executorService =
    Executors.newScheduledThreadPool(1);
private final ScheduledThreadPoolExecutor dataExecutor =
    new ScheduledThreadPoolExecutor(1000);
```

The first scheduler, our `executorService` will serve as the scheduler for our daily check of XML file modifications. The second `dataExecutor` will hold all scheduled resources. From now on, to make a distinction between the first (main) scheduler and the secnd (resource scheduler), they will be called respectively MainScheduler and ResourceScheduler. Now each scheduler contains a collection of `Runnable` objects. These reflect the tasks that can be executed at a specific interval. So for our MainScheduler we need one simple task, which is updating all our resources from the XML files, so they reflect the correct URL's and intervals in our ResourceScheduler. We create this new `mainTask` in the private constructor of our `DataScheduler` class.

```
private DataScheduler() {
    mainTask = new Runnable() {
        @Override
        public void run() {
            XMLReader r = XMLReader.getInstance();
            for (City city: r.getAllCities()) {
               try {
                   updateModifiedResources(city);
               } catch (HadoopException ex) {}
            }
            rescheduleModifiedResources();
        }
    };
}
```

This main task retrieves an instance of our `XMLReader`. Then we loop through the collection of all cities, which is called and parsed by the `XMLReader`. Then for every city, we call the `updateModifiedResources()` method. It is this method that implements the functionality for when resources are added, modified or deleted. Now when all new resources are added, existing ones modified

or deleted, we can call the `rescheduleModifiedResources()` method that takes care of updating the intervals at which the modified resources are scheduled. The implementation for these methods is too extensive and is therefore left out in this section. In Appendix 2, a full implementation of this class can be found.

Now because we need to have control over running tasks and their related resources, we need a reference to them. But using a `ScheduledThreadPoolExecutor`, you can only retrieve a queue of all scheduled tasks. And these tasks are `Runnable` objects. So how do we know which Runnable object corresponds to which resource? For this problem we created a new class called `DataScheduledTask` which implements the interface `RunnableScheduledFuture`. This way, we can add a reference to the resource that the task corresponds to. To illustrate this, here is the code of this class without implemented methods and constructor. In Appendix 3, the full code can be found.

```
public class DataScheduledTask implements RunnableScheduledFuture {
    private Resource resource;
    private DataPuller puller;
    private boolean running;
    private boolean cancelled;
    private long startTime;

    ...

    public Resource getResource() {
        return resource;
    }

    public void setResource(Resource resource) {
        this.resource = resource;
    }
}
```

Now we implemented our schedulers and the custom implementation of `RunnableScheduledFuture` to schedule objects on. This application runs on a web server, but how can we start our MainScheduler with this `mainTask`? Nothing triggers this task to be run. The ideal moment to start this task is when the application has just finished starting up. Then the `mainTask` may update all resources in the XML files. This is done by creating an `ApplicationEventListener` which is included in a separate package called

`org.glassfish.jersey.server.monitoring`. This is an interface that is used to define a class where certain events (like on web server startup) are captured. We won't go in detail about this now since this part belongs to data providing in chapter 10.3.3 Data providing. We simply start the main task of the data scheduling by calling the following method.

```
DataScheduler.getInstance().initialize();
```

Data Puller

Now we got the assignment to pull data from the schedulers. The next step in the data retrieval process, is executing a HTTP `GET` request to the resource's URL. For this functionality we created a new class called `DataPuller`. The only task of this class is to create an HTTP request, execute it and call the correct driver to create a job to map the data from the response body.

Java has a built-in HTTP client and server component library. But it's a little difficult to use and we found a much more user-friendly library from Apache, which is called `httpclient` from the `httpcomponents` package. We define this new library in our `pom.xml` file, so Maven can automatically download this dependency for our project.

```
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.4.1</version>
</dependency>
```

We must not make our `DataPuller` class a singleton, because each scheduled resource will create a new object of `DataPuller`, so it can request its data and not worry about other requests. There will be one request per instance of `DataPuller`.

There is one method we define in our `DataPuller` which is `pull()`. This method takes a `Resource` object as parameter and returns void as its return value. Executing an HTTP request is an asynchronous task, therefore the `httpclient` library provides us with a `ResponseHandler` class to handle the

HTTP response that is sent back from the host. This handler checks whether the status code that is returned in the response header is valid, meaning the status code must be between 200 and 300. If the status code is valid, the body of the response is written to an `HttpEntity` object that's part of the Apache `http` library. Let's take a look of how we implemented this `ResponseHandler`:

```
responseHandler = new ResponseHandler<String>() {
    @Override
    public String handleResponse(HttpResponse hr) throws
        ClientProtocolException, IOException {

        int status = hr.getStatusLine().getStatusCode();
        if (status >= 200 && status < 300) {
            HttpEntity entity = hr.getEntity();
            if (entity == null) {
                return null;
            }
            return EntityUtils.toString(entity);
        } else {
            throw new ClientProtocolException(
                "Unexpected response for resource pull with URL: "
                + url + ". Got response code: " + status);
        }
    }
}
```

We now have a way of handling the response we will get from the request. Now we need the request itself. The key objects for this are `CloseableHttpClient` and `HttpGet` from the Apache `http` library. `CloseableHttpClient` opens a new socket for a HTTP request. `HttpGet` is used to create a `GET` request header. Let's take a look at the necessary steps to create this `HttpGet` object and execute the `HttpClient` object with this `HttpGet` object and our previous `ResponseHandler`.

```
public void pull(Resource resource) throws IOException {

    CloseableHttpClient httpClient = HttpClients.createDefault();
    final String url = resource.getUrl();

    try {
        HttpGet httpGet = new HttpGet(url);
        String responseBody = httpClient.execute(httpGet,
            responseHandler);
    } finally {
        try {
            httpClient.close();
            JobClient.runJob(DriverFactory.createMapJob(resource));
            catch (ClassNotFoundException ex) {
                throw new MapperNameException(
```

```
                    "Wrong resource name link to Mapper");
            }
        }
    }
}
```

10.3.2 Data storage

Mapping

Hadoop works on the principle of providing an input path of a particular resource to apply some MapReduce functions on. Our `DataPuller` can now request data from a resource URL. We need to store the data that is returned in the response body, onto the Hadoop distributed file system. In our `DataPuller` class we created a new method to do this.

```
private void storeOnHDFS(Resource resource, String data)
    throws IOException {

    Configuration conf = new Configuration();
    Filesystem fs = FileSystem.get(conf);
    Path outFile = new Path(resource.getName());
    if (fs.exists(outFile)) {
        fs.delete(outFile, true);
    }
    BufferedWriter br = new BufferedWriter(
        new OutputStreamWriter(fs.create(outFile, true)));
    br.write(data);
    br.close();
}
```

This method writes the Response in string format to a text file that is located on the HDFS file system. We need to provide a directory name, and for this we simply choose the name of our resource. This way when we need to run the mapper through the `Job` object, we can easily specify the location of our stored text file.

In Hadoop terms, a class that configures a `Job` is called a driver class. We need to configure a `Job` object, because we need to run a MapJob on our data, so we can store it in our HBase database. Depending on the resource, a `Job` can be set with various parameters. To make this easier to configure jobs, we created a `DriverFactory` class that creates jobs for specific types of data.

This class creates a Hadoop `Job` object and returns it to the caller. In our case we just need a Map function for our `Job`, so we'll create a method that creates a `Job` just for this.

```
public class DriverFactory {
    public static JobConf createMapJob(Resource resource)
        throws IOException, ClassNotFoundException {

        Class mapperType = MapFactory.createMapperType(
            resource.getName());
        JobConf conf = new JobConf();
        conf.setJobName(resource.getName());
        conf.setJarByClass(mapperType);
        conf.setMapperClass(mapperType);
        conf.setInputFormat(resource.getInputFormat().getClass());
        conf.setOutputFormat(DBOutputFormat.class);
        FileInputFormat.setInputPaths(conf,
            new Path(resource.getName()));
        FileOutputFormat.setOutputPath(conf, new Path("output"));
        return conf;
    }
}
```

This method creates a new `JobConfig` object that is used to run a Hadoop job. On this `JobConfig` instance, we set the `Mapper` class that the job should use to transform the data. For this type, we created a `MapperFactory` class that converts the resource name to the corresponding Mapper name in our application. We also need to set what type of `InputFormat` will be used for the Mapper, in our case the type of resource is defined in the `Resource` object. We write everything to our HBase database, so we set the `OutputFormat` value to `DBOutputFormat`. Next we specify the location of our resource on the HDFS. When we stored the file, we took the name of the resource as its directory name. Now we can provide our resource name as the location of our input path. Hadoop requires us to set an `OutputFormat` as well, but since we will write everything to HBase, only the directory 'output' will be created, with nothing stored in it.

After the `DataPuller` retrieves a response from the HTTP `GET` request, it closes the connection. It's at this point that it runs the Map job, to transform the data and load it into the database.

```
JobClient.runJob(DriverFactory.createMapJob(resource));
```

HBase

While continuing with our example of the vehicle detection (see chapter 6.3 MapReduce), we'll explain the schema of the `VehicleDetectionReading` table, which stores all of the actual sensor values.

Just to point this out, this is obviously not the only table we'll have in our system. There will be a table holding the information about the different sensors like the brand, how old they are, repair status, etc. There might also be a table called 'Crossroad' which connects all sensors located in the same crossroad. Without a doubt, there will be even more tables but only the most important one, the `VehicleDetectionReading` is discussed here. The schema of this table is shown on figure 25.

Row key	Datetime	Value
s0001	1425247200000	25
	1425250800000	20
	1425254400000	7
	1425258000000	19
	1425261600000	49
	...	
	1425330000000	18
	1425333600000	18
	1425337200000	19
	...	
	1425416400000	21
s0002	1425247200000	2
	1425250800000	0
	...	
	1425416400000	0

Figure 25 - HBase VehicleDetectionReadings schema

To be able to fully interpret this HBase schema, please make sure you have read the theoretical chapter on HBase (7 HBASE)

The schema is built up in the following way. First we take the sensor id as row key (`s0001`, `s0002`, …) to distinguish the different sensors. This sensor will be measuring data on an hourly interval, every single day. To differentiate these moments, we use the `Datetime` column which contains `Long` values representing the specific timestamp. The lower in the table, under the same row key, the more recent the data will be. For example the first `Datetime`, `1425247200000`, represents 00.00 AM on the 2nd of March 2015 and at that moment a value of 25 was registered. If we now take a look at the row beneath

that one, the `Datetime` represents the same date but at 01.00 AM. This goes all the way until 23.00 PM (the first row on figure 21 which is positioned beneath the first dots), because after this value 00.00 AM is reached on the next day. This means that the next value is registered on the 3rd of March 2015 which has the `Long` value of `1425333600000` and represents an actual value of 18. The next `Long` value will then represent the second hour (01.00 AM until 02.00 AM) with a value of 19, and so on and so on. What's good about this way of storing the dates and according time is that it's very easy to read them out, to transform them into `Timestamp` objects and to sort them.

The last column is filled with the actual values which are integers that represent the number of cars counted by a specific sensor on a specific moment. Like mentioned in the theory, a timestamp will be added to each stored value. This is not shown on the image above, to keep the example clear, but is important and it represents the moment when the data was put into the database.

This schema is built upon the table format of HBase and is pretty much the most efficient way to store data considering the goal of our system.

10.3.3 Data providing

This part of the system has as goal to actually provide data to the end user. This data provision is based upon requests and sends data from the database to the user through a web service.

Servlet container

For our application, we need to be able to provide an interface to our end-users so they can access our data stored in HBase. We can achieve this by creating a web service. One of the developers of this project, was already quite familiar with web services based on the REST structure, which will be explained later. Jersey is a framework to create RESTful applications, it makes use of annotated class definitions to define the REST services. The biggest advantage for our project of using this framework, is that this framework can be declared as a

dependency in our Maven `pom.xml` file. And on top of that, we can use the integrated Glassfish server that Jersey runs on top of.

```
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-server</artifactId>
    <version>2.17</version>
</dependency>
```

Now requests to a web server normally are requests to resources like HTML files, which are the most common. But the requests we will receive are those that request data from our HBase database. So we need our application to call this requested data from the database. So what the end user is actually requesting is an application, and not a resource. Running applications on a webserver are triggered by a servlet. A servlet is a container that can call various parts of an application that reside on the webserver.

Jersey provides us with an implementation of their servlet container, it's this container that will handle all our REST requests and call the correct part of the web service. First, we need to make sure our web application knows that it should use Jersey's implementation for the servlet container. In Java, we do this by creating a `web.xml` file. In this file you can define multiple servlets, but also listeners for your application (for example an application initialized event listener). We only need to define one servlet, the Jersey servlet container:

```
<web-app … >
    <servlet>
        <servlet-name>PublicServicesApi</servlet-name>
        <servlet-cass>
            org.glassfish.jersey.servlet.ServletContainer
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>PublicServicesApi</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Let's give a summary of the content of this servlet definition. First, we define a new `servlet` element, this element always needs to have at least the `servlet-name` and `servlet-class` child elements. The `servlet-name` is

something the developer can decide for his own, in our case we called it `PublicServicesApi`. The `servlet-class` element is where you define the container class, in our case its Jersey's implementation which is located in the `org.glassfish.jersey.servlet` package. Then we added one more element as child of the `servlet` element, `load-on-startup`. This element's value indicates whether this servlet should be loaded on startup, in our case this is set to 1 which means the servlet will be loaded when starting the application.

The second part in the `web.xml` file is the `servlet-mapping` element. Here we define the URL to access the servlet container. We do this by declaring the `servlet-name` exactly the same as declared in the `servlet` element (this is very important, since this links the `servlet` to the `servlet-mapping`). Then we can specify the URL-pattern that should be used to access our web container. In our case users will always make a request to the container when the URL has the following format:

```
http://hostname/api/*
```

With all this defined our web container will be accessible for end users.

Servlet resources

Now before getting on with building the actual web service, we need to configure some other features that are vital to the correct operation of our web service. But there is even something else we can do at this point. Remember the section where we explained the initialization of the `DataScheduler`? We need to have a trigger, to start the whole process of our `DataScheduler`. Because we defined our first servlet container, we can now add some features or tools to this container. In Jersey language, these are called resources.

What we need is an event listener, specifically an event listener that is triggered when the application is fully loaded on the webserver and thus is ready to receive requests. Jersey provides us with an interface you can use to define listeners on application-level. This interface is called `ApplicationEventLis-`

tener and is part of the org.glassfish.jersey.server.monitoring package. It defines two methods.

```
public void onEvent(ApplicationEvent ae);
public RequestEventListener onRequest(RequestEvent re);
```

It's the first method that is of importance to us (we also created an implementation for the onRequest() method, but this is beyond the scope of this thesis). We created a new class called PublicServicesEventListener, which implements this ApplicationEventListener interface.

```
public class PublicServicesEventListener
    implements ApplicationEventListener {

    @Override
    public void onEvent(ApplicationEvent ae) {
        switch (ae.getType()) {
            case INITIALIZATION_FINISHED:
                DataScheduler.getInstance().initialize();
                break;
        }
    }

    …
}
```

Let's now discuss the workflow of this class. Whenever an application event is thrown, the onEvent() method is called, which has a parameter with the ApplicationEvent in. Then we check the type of the ApplicationEvent, because we need to capture the event with the type being INITIALIZATION_FINISHED. If that's the case, we can call initialize() on our DataScheduler. This will start the process of scheduling resources.

We defined our new PublicServicesEventListener class, but we didn't yet tell our servlet container to use this listener for our application. There are several options Jersey provides to do this. We choose to do this by defining an Application class that inherits from ResourceConfig which is part of the org.glassfish.jersey.server package. In this class we can define all the resources, modules, features, etc. that our servlet container can use. We called our class PublicServices which is the name for our entire application.

```
public class PublicServices extends ResourceConfig {

    public PublicServices() {
        register(PublicServicesEventListener.class);
    }
}
```

We define our new listener by registering the classtype to the `ResourceConfig` class. Now we need to declare this `PublicServices` class in our `web.xml` file, so our servlet container knows that it should use this class as its resource pool. We can do this by adding initial parameters to the servlet definition.

```
<web-app … >
    <servlet>
        <servlet-name>PublicServicesApi</servlet-name>
        <servlet-cass>
            org.glassfish.jersey.servlet.ServletContainer
        </servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-val-ue>
                fi.karelia.publicservices.util.PublicServices
            </param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
</web-app>
```

The `param-name` element's value, declares that this parameter defines an `Application` class for the servlet container. The type of which is defined in the `javax.ws.rs` package. The `param-value` element contains our package name and classname for our `ResourceConfig` specialization.

To make development easier for our web service, we can also add a provider for our resources that can parse `Serializable` objects in JSON format. This way, when we need to send HTTP response objects back to the user, we can parse our data from objects to JSON automatically. For this to work, our servlet container needs to know that it can use this provider, therefore we need to register this resource in our `ResourceConfig` implementation.

```
register(ObjectMapperProvider.class);
register(JacksonFeature.class);
```

We won't go into detail about these classes because we only focus on the core components of our application. `JacksonFeature` is a class used by `Object-MapperProvider`, that's why we also need to register it in our `Application` class.

REST

REST, or Representational State Transfer, is way to build web services. What this means for us is that, since we'll use it, it will define how to build our web service.

REST web services will communicate using the HTTP response codes, media types and methods.

These methods are:

- GET, which is used to retrieve an object.
- POST, used to add a new object to the database.
- PUT, used to update an existing object to the database.
- DELETE, used to delete an object from the database.

The HTTP response codes are:

- 200: OK
- 201: Created
- 304: Not Modified
- 400: Bad Request
- 401: Unauthorized
- 403: Forbidden
- 404: Not Found
- 500: Internal Server Error

And the mentioned media types are XML, JSON, HTML, plain text, …

Below, you can see an example `GET` request.

```
GET /users/1 HTTP/1.1
User-Agent: Chrome
Accept: text/html
[CRLF]
```

Different parts can be distinguished in this example. First we have the HTTP method `GET` which means we request an object. Then the location of the resource is provided, here `/users/1`, which means that the user with id 1 will be returned in this case. Then the HTTP version and additional request headers (here the used web browser) are added. The actual request body in which the desired media type is mentioned follows these optional headers. The request is ended with a `CRLF`, which means Carriage Return Line Feed, to mark the end of the header. This request would be answered by a HTTP response that would look similar to the one below.

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 200
[CRLF]
<html>
     <!—Some HTML page/content with the name of the user with id
     1 -->
</html>
```

The response again starts with the HTTP version and then the response code, here 200 which means everything went well. This header line is followed by the content-type and -length fields with another a `CRLF` tag to define the end of the header. After the header, the actual response body is shown with the requested result.

The following example is an HTTP `POST` example, which simulates the creation of a new object.

```
POST /users HTTP/1.1
User-Agent: Firefox
Content-Type: application/json
[CRLF]
{“name”:”John”, “lastName”:”Smith”}
```

This one is quite similar to the previous one but what's different is the HTTP method, which is obviously `POST` in this case, and the request body. In this case, the request body holds the data to be updated in the database. The web service will then know that the user with id 1 needs to be updated, because it's defined that way. Again, there would be an HTTP response following this request and it would be similar to the one below.

```
HTTP/1.1 201 Created
Location: /users/1
[CRLF]
- Some confirmation message
```

And yet again, this message can be broken up into pieces. First the familiar HTTP version and response code are given. These are followed by a repetition of which object had to be updated, here it is indicated by using the folder structure, which will be explained later. The header is again terminated by a `CRLF` followed by a message to the user which can basically say anything.

In table 8 an example is shown of how the location URL's can be defined combined with the result of performing a request on them.

Table 8 - REST HTTP requests example

Resource	**POST** create	**GET** read	**PUT** update	**DELETE** delete
/user	Create a new user	List users	Update all users	Delete all users
/user/1234	Error	Show user 1234	If user exists, update 1234, if not, error	Delete 1234

Web service

We made all the necessary preparations for building our web service. Now it's time to finally create it. Using the Jersey framework, creating web services has never been easier. One part of the URL defines an object that the user can perform a request on. In our application we are working with sensors, therefore we will create a `SensorService` class. This class contains a reference to our

business logic layer for sensors, this way we can request resources. To make this class part of our web service, we need to annotate the class with the `Path` parameter. All these annotations are part of the `javax.ws.rs` package.

```
@Path("/sensor")
public class SensorService {

    private final SensorBLL sBLL;

    public SensorService() {
        sBLL = new SensorBLL();
    }
}
```

This way, when an end-user requests a resource at `http://hostname/api/sensor`, the servlet container will automatically call this `SensorService`, because it was declared on this path. But making requests now, won't work. First, we need to declare some methods that represent the corresponding HTTP methods. We can do this again by annotating our methods.

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getAll() {
    List<Sensor> list = sBLL.getAll();
    return Response.ok(list).build();
}
```

This simple method returns all sensor information to the end-user by creating a `Response` object. There are many ways that we can construct this `Response` object, now we can simply call the `ok()` method providing our body as parameter. Now this looks a bit strange, `List` is a Java datatype, so how can we add something like this to the body of an HTTP Response? Well, in the previous section we added a provider to our servlet container that automatically parses objects into the desired format. We annotated our method with the `Produces` tag that defines the content-type in the HTTP response. So our `ObjectMapperProvider` will automatically parse this list into JSON format and add it to the body of the `Response` object.

Now let's say and end-user wants a sensor with a specific id. This would be an HTTP `GET` request and the id can be provided in the querystring of the URL that was requested. For such requests, a method like the following can be created.

```
@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response findById(@PathParam("id") int id) {
    Sensor s = null;
    try {
        s = sBLL.findById(id);
    } catch (ApplicationException ex) {
        return Re-
sponse.status(Status.NOT_FOUND).entity(ex.getMessage()).type(MediaType
.TEXT_PLAIN).build();
    }
    return Reponse.ok(s).build();
}
```

The `Path` annotation tells the service that there is a querystring parameter present. This parameter is (automatically) parsed by our `ObjectMapperProvider` to an `Integer`. It's possible that there is no sensor found for the given id, in that case an `ApplicationException` will be thrown stating that the sensor for that id was not found. But following the rules of REST, we also need to return the correct status code, which is 404 for when a resource is not found. In that case our body won't have to be parsed to JeSON format, therefor we explicitly tell the `Response` builder that the used content-type should be plain text.

In our web service, users should only be able to perform `GET` requests, data provision is done via our `DataPuller` which is called by our scheduled resources. So for now, two `GET` methods will be enough to provide our users of some basic data. In the future we can create more methods for specific requests, for example to retrieve the detection sensor on a specific crossroad.

Our web service class is ready, but as you may have guessed, we didn't declare this class anywhere. We need to tell our servlet container that this class handles all requests for the `http://hostname/api/sensor` URL. We need to define our service in our implementation of `ResourceConfig`. Because we will have many different web service classes in the future, we created these

classes in the package `fi.karelia.publicservices.service`. We can add this whole package as a resource for the container to use.

```
public class PublicServices extends ResourceConfig {

    public PublicServices() {
        packages("fi.karelia.publicservices.service");

        register(ObjectMapperProvider.class)
            .register(JacksonFeature.class)
            .register(PublicServicesEventListener.class);
    }
}
```

The web service is now registered on our servlet container and can start accepting HTTP request to the defined URL for resources. In the next section we will talk about how we retrieve the requested resource through our business logic.

BLL

The first layer where the request goes through is the Business Logic Layer. The goal of this layer is to prevent illogical requests and unauthorized access. All of the Business Logic classes are placed inside the `fi.karelia.publicservices.bll` package. If, for example, data from a sensor is requested and the specified sensor doesn't even exist, the BLL class will check this and return an appropriate message. It is important that these business rules are considered carefully so that every necessary possibility is checked. The list of required rules is usually composed by having a thought about every possible input of the user and what could go wrong when processing these requests. We'd also like to point out the importance of this layer. In our case, there won't be any manipulations on the data in the database, no user will alter the data. But, if it would have been possible, then it's very important to configure this layer very well. If a user would have bad intentions, or do bad things unintentionally, the system could break or different data injections could be possible. The BLL layer is created to secure this system from any unintended or unforeseen actions. Next to preventing this from happening, providing the user with clear messages considering the mistake he/she might have made is also important. All of this defines the importance of this logical layer.

In our project we have an example BLL class for retrieving sensors, called `SensorBLL`. This class is the link between the web service, which will take the request from the user, and the DAO layer, which makes connection with the database. All of the logical features considering sensors are placed in this class.

For now, there are two example methods in this layer. The first one is the `getAll()` method which will retrieve every single sensor out of the database. The method is displayed in the code snippet below.

```
public List<Sensor> getAll() {
    sDAO = new SensorDAO();
    return sDAO.getAll();
}
```

This method takes the request from the web service and passes it on to the next layer (DAO) which will then get all of the sensors out of the database. In this method, there is no logical check of any kind but it is advised that it is done this way for two reasons. The first reason is that it is important to maintain the layer system and always go through the BLL layer before connecting to the database. It should always be done this way and maintaining this workflow is beneficial because it makes the programmer keep using this structure. The second reason is that if any check should be added later on, it can just be applied to this method without still having to create it.

The second method, which is the `findById()` method, makes things more interesting. Again, the code can be found below.

```
public Sensor findById(int id) throws ApplicationException {
    sDAO = new SensorDAO();
    Sensor s = sDAO.findById(id);
    if (s == null) {
        throw new ApplicationException(
            "Sensor doesn't exist for id: " + id);
    }
    return s;
}
```

In this method, a sensor can be found by specifying an id. What happens in this layer is a check to see if there actually is a sensor with the specified id. If not, the user will get a message that there is no such sensor. This is actually a good

example of what the BLL layer serves for. These checks are created to see if the user requests correct data and they are very important but yet easy to implement in this layer. What also could be (and maybe will be) implemented is a check to see if the specific user should get actual access to this data. If there is some kind of user account system, the BLL layer can give a response that access is unauthorized and thus not granted.

DAO

After the requests are taken and checked for reasonability, the next layer is active. This layer is the DAO, or Data Access Object layer. This is where an actual connection with the database is set up. This layer gets the forwarded request of the BLL layer and performs the actions to fulfil the request.

Below, the code of how the DAO layer handles the two example methods from the BLL layer can be found. The class for this example is called `SensorDAO`.

```
public List<Sensor> getAll() {
    return getSensorCollection();
}

public Sensor findById(int id) {
    Sensor resS = null;
    for (Sensor s : getSensorCollection()) {
        if (s.getId() == id) {
            resS = s;
            break;
        }
    }
    return resS;
}
```

The first method is the `getAll()` method from before, which returns all of the `Sensor` objects found in the database. For now, the result of a `getSensor-Colletion()` method is returned which refers to another method which actually makes connection with the database and gets all of the `Sensor` objects out of it.

The second method is also familiar and is the `findById()` method. This method loops through the full collection of retrieved `Sensor` object, searching for the sensor with the desired id. The mentioned collection of sensors is retrieved in the same way as with the previous method, by calling the `getSensorCollection()` method. After performing this method either a Sensor is returned with the desired id or a null-object is returned, which will then be checked by the BLL layer.

These are two example methods but what should be obvious is that the DAO layer makes connection with the database and gets the actual objects. These objects can be anything, the previous examples used `Sensor` objects but they would usually be sensor values stored in the database. Then these objects or values are filtered by anything desired by the user. There could for example be a method called `getSensorValuesByDate()` which takes a `Date` as parameter and then filters all of the values by this date. The system is very agile and can be adjusted to whatever the end-user may request.

10.3.4 Error handling

To prevent our system from breaking we have written different Java Exception classes. Exception classes are written either to deal with user errors or to solve programmer errors. This chapter will explain all of the exceptions written for this project and in which case they are thrown.

HadoopException

The `HadoopException` class is our first exception and is written to prevent Hadoop-specific errors to crash the system. This exception is thrown in the `DataScheduler` class, when the resources of a specific city need to be updated, but the configuration file is not found. We made a `HadoopException` for this case because this error is caused from the HDFS structure. If the file is not residing in the Hadoop Distributed File System, then the update can’t be performed. This exception will then be caught in the run method of the `DataScheduler` since that’s where the update method is called and where it

would go wrong. In case this exception is thrown and thus something is actually wrong, the message of the exception and all of the necessary info will be forwarded into the appropriate log files. This makes it easy for the programmer to track where the problem was caused.

ApplicationException

The `ApplicationException` class is an exception designed to prevent the application from being interrupted or crashing when providing the data. When, for example, a value is not found in the database, the BLL layer will generate an `ApplicationException` with the following message: "Object doesn't exist for id: X", with X being the requested id. This updates the user of what happened and prevents the system from doing something with a non-existent object.

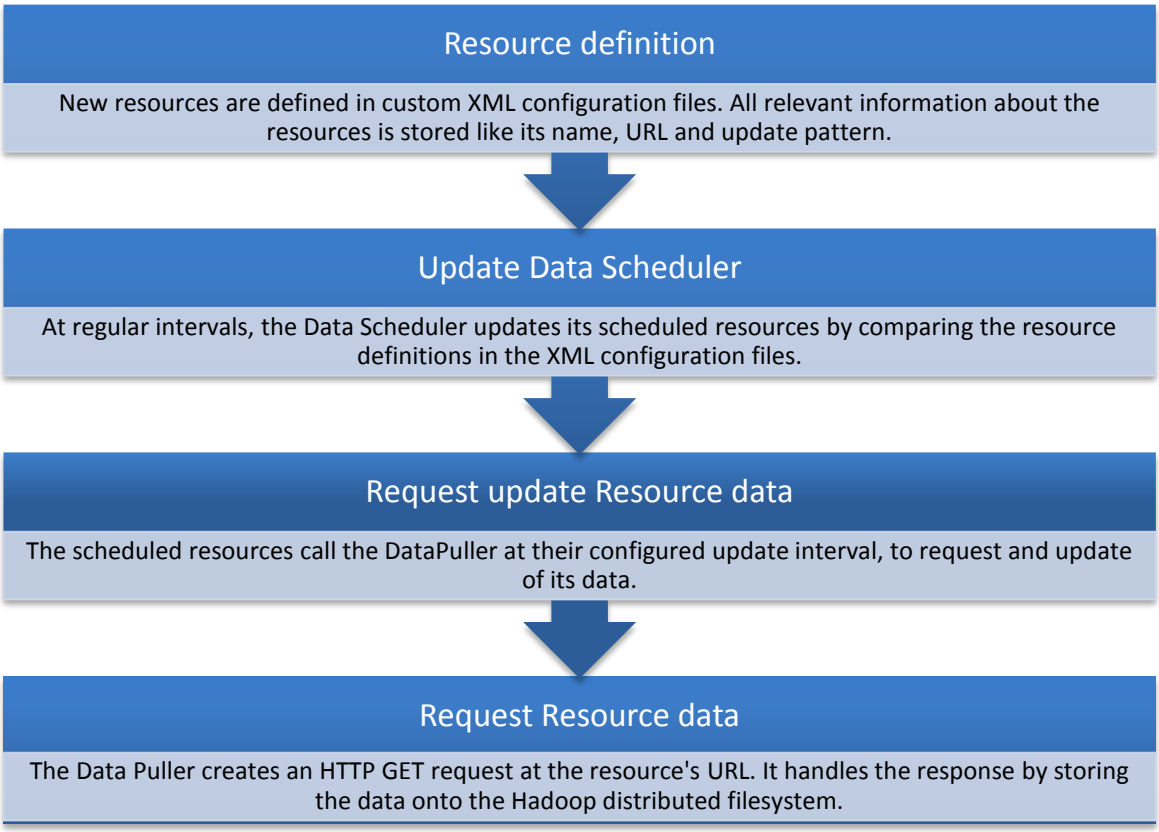
DBException

The next exception on this level of the system is the `DBException`. This exception handles everything that could go wrong considering the database. This exception is thrown, for example, when the DAO layer, could not access the database. So, in other words, what this exception basically does is handle the exceptions thrown by the HBase component. It handles them and shows the user what went wrong.

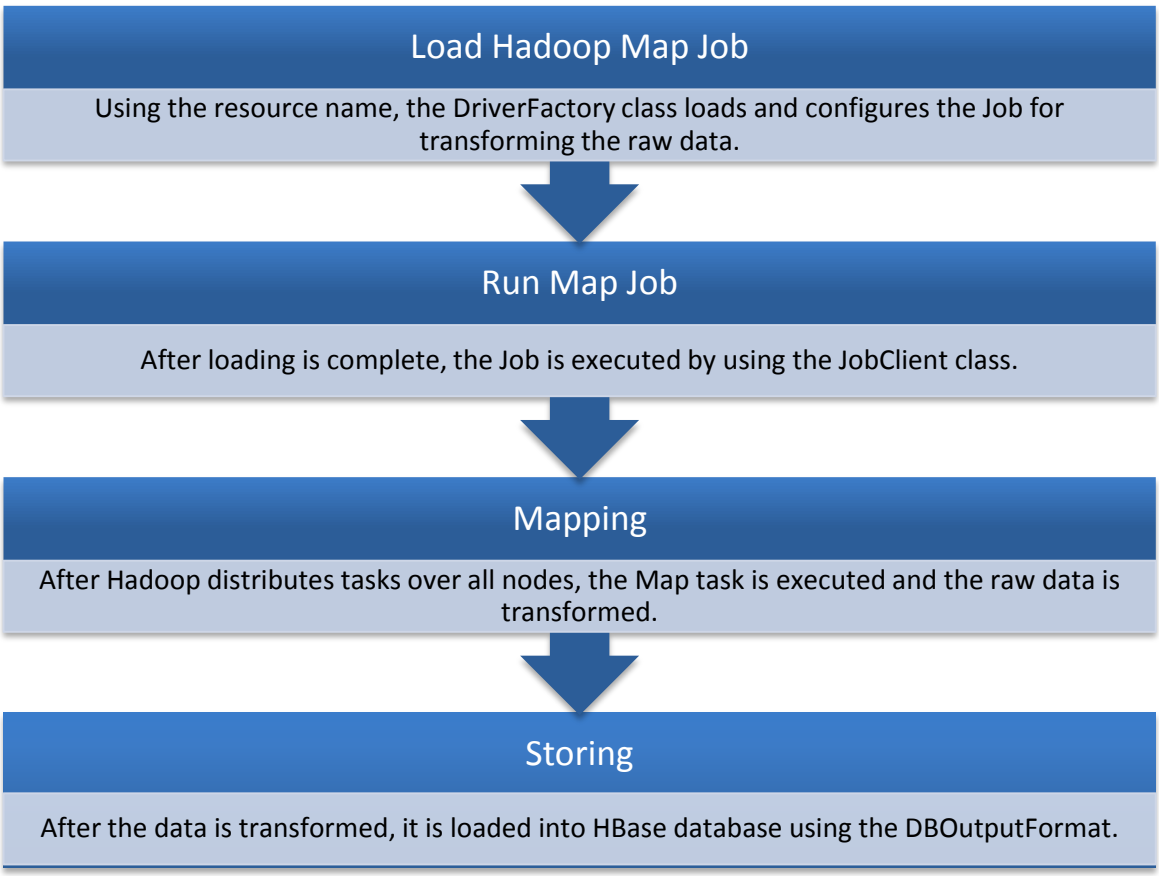
10.3.5 Workflow

We discussed all the transfer paths of data in and from our data warehouse. We described every part with extensive code samples and explanations. Now we want to give you an overview of the complete process step by step. We'll show how we get data from a source into our warehouse and how to we provide this data to end-users. To structure the overview, we split it up into the familiar three steps: data retrieval, data storage and data provision.

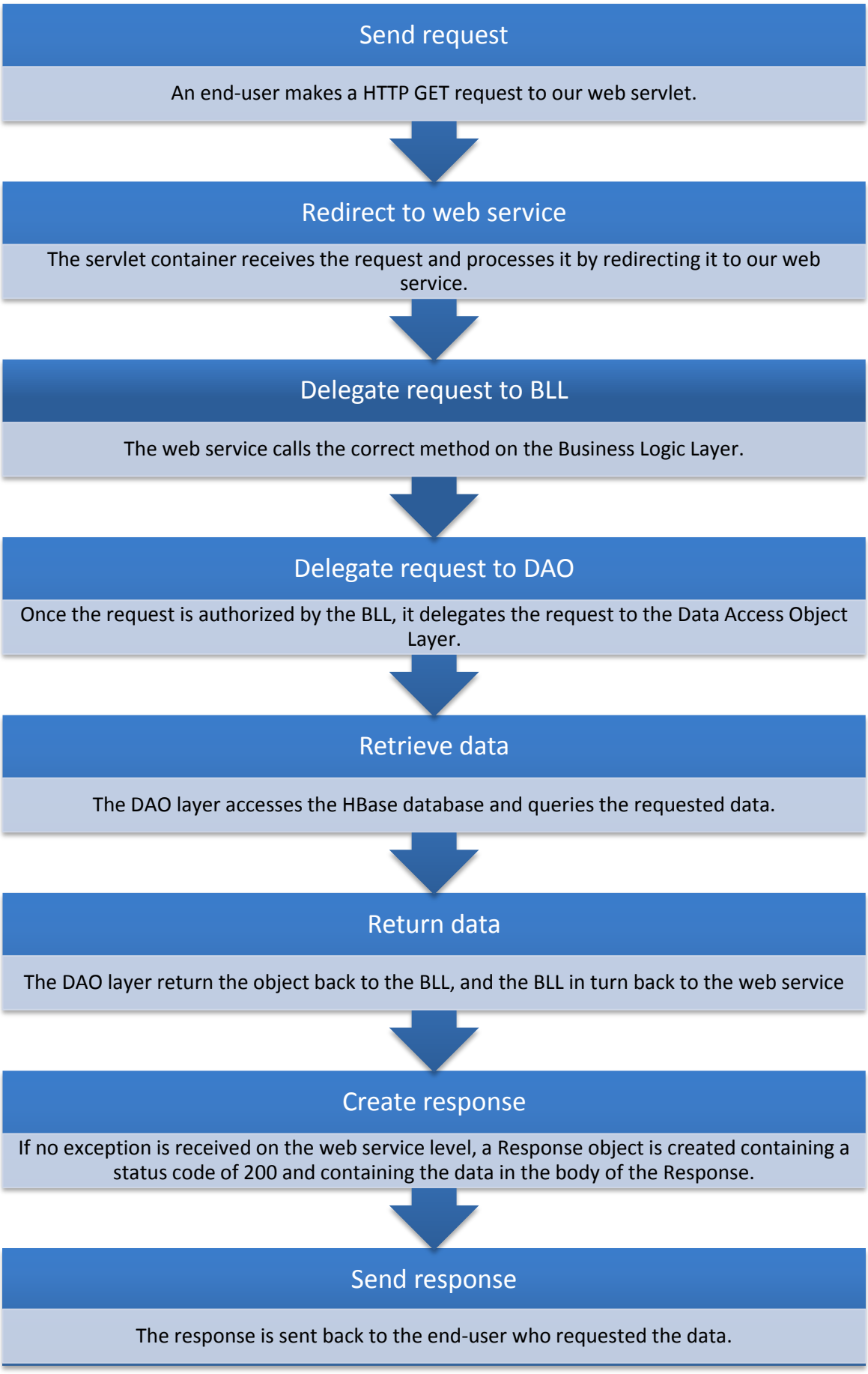
Data retrieval



Data storage



Data provision



11 REFLECTION

This final chapter provides a reflection on the project, the difficulties we encountered and our learning outcomes.

11.1 Difficulties

Delay in development

Since the idea was to keep the setup running after our final project is finished, we planned to run the configuration on some virtual servers on campus. We immediately got in contact with the teacher who is in charge of the networking and server equipment at the university. He told us that there was a problem in their setup and that it would take some time to fix it. Eventually the problem got solved but we lost about three weeks because of it. Although useful time was lost, we still managed to use this time wisely by diving into the theory and studying Hadoop and its components.

Restricted access to data

Initially we were planning on using data related to snow ploughs and the bus schedules for our example setup. The difficulty about this was that we couldn't access this data. We then had to wait for other sample data. So again we lost quite an amount of time, but eventually we got some data to work with.

As can be concluded, we lost time due to different issues but we used it wisely and managed to fulfil the goals set for this project. We have an example setup running right now and it can easily be expanded by adding other features.

11.2 Future thoughts

Since the system is not yet completely finished, we hope the development will continue in the future. In our point of view, the system has great potential if it would be developed further. Extra functionalities should be added to fit the needs of the end-users. For now, the system resides in its basic stadium but has everything it needs to become a large but solid data warehouse.

11.3 What did we learn

Since the first big part of our project was to familiarize with Hadoop and its components, research took quite a lot of time. We learned about the various features and requirements of Hadoop such as hardware requirements and optimisation, system development and HBase structures. We are very satisfied with the learning curve we had when studying all this material. There are still many rocks unturned but we got hold of more than the theoretical basics. All topics and components were new to us but we managed to study and comprehend them very well.

The implementation itself was also very interesting to us. We had never developed such an extensive application before and learned a lot from it. From writing configuration files to making access with a database, from using a scheduler to boot up tasks to writing mapper functions, this whole project was extremely instructive.

But we did not only learn something new due to the actual tasks of the project. To give an example, our experience with the Linux operating system also expanded through the progress of this project. So all in all, we learned a lot and reviewed and updated our existing knowledge and capabilities.

11.4 Workload balancing

Overall, this project has been the result of very good teamwork and most of the times we worked together. We discussed different design, implementations and possibilities but each of us also developed different parts of the system. In this part we'll discuss how the work was divided and which tasks took the longest.

To start off, we'd like to say that there was way more work to do during this project than one would think at first. Next to the fact that the complete system is quite huge, other tasks were executed too during the progress of the thesis.

The first part of the final project was getting to know the topic, 'The Internet of Things'. To do this, our coordinator proposed to make a small setup with an Arduino computing unit. The goal was to save sensor readings to the unit and send them to a web service. Together we created this small test setup. Later on Ruben created a small web service on another machine to convert these readings into CSV files.

After that, we studied Hadoop and its components to be able to start the development. Ruben came up with the basic design for the whole system. We tweaked it here and there and optimised the complete setup. We also read the most important chapters of "Hadoop: The Definitive Guide" by Tom White and "HBase: The Definitive Guide" by Lars George. Meanwhile Ruben created a small example application to simulate the MapReduce programming model. For this, we installed a virtual machine with Ubuntu server and Hadoop. In the meantime, we got in contact with the students that created our networking setup. Since they had some trouble with it and Jonas is specializing in Networking, he helped them and eventually the complete network setup was finished.

For the application itself, Ruben set up the developing environment, which means configuring Maven and the web server. Jonas configured the version control system and Ruben started by creating the web service based on REST together with configuring the servlet container to handle requests. For data

retrieval, Ruben created the complete data scheduling and pulling mechanism. For data storage he created some basic mappers and drivers to use on our sample data.

Jonas created the DAO and BLL classes to provide the access to the database, along with the domain classes which represent the data. Next to that, he also created the HBase schema for our example. Considering the system, he also designed the XML files and wrote an XMLReader class to read out these configuration files.

If we now consider this thesis, the work was also well-balanced. Each of us wrote approximately the same amount of text and we both reread and tweaked the complete thesis.

REFERENCES

Apache HBase Team. 2015. Apache HBase Reference Guide. http://hbase.apache.org/book.html. 18.05.2015.

Apache. 2010. HBase/Shell. https://wiki.apache.org/hadoop/Hbase/Shell. 18.05.2015.

Cloudera. 2015. http://www.cloudera.com/content/cloudera/en/products-and-services/cloudera-express.html. 20.05.2015.

Cloudera. 2015a. http://www.cloudera.com/content/cloudera/en/downloads/cdh/cdh-5-4-2.html. 25.05.2015.

Cloudera. 2015b. http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/cm_ig_install_path_b.html#cmig_topic_6_6. 25.05.2015.

Datawarehouse4u. 2009. Data Warehouse. http://datawarehouse4u.info/. 04.05.2015c.

George, L. 2011. HBase: The Definitive Guide. O'Reilly press.

Greenplum. Hadoop Components. NDM. http://www.ndm.net/datawarehouse/Greenplum/hadoop-components. 22/04/2015.

Khurana, A. 2012. Introduction to HBase Schema Design. http://0b4af6cdc2f0c5998459-c0245c5c937c5dedcca3f1764ecc9b2f.r43.cf2.rackcdn.com/9353-login1210_khurana.pdf. 18.05.2015.

OpenTSDB. 2015. HBase Schema. http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html. 18.05.2015.

Steinberg, R. 2015. The Origin of the word Daemon. The Austin Chronicle. http://ei.cs.vt.edu/~history/Daemon.html. 31.05.2015.

Varley, I. 2012. HBase Schema Design. http://www.slideshare.net/cloudera/5-h-base-schemahbasecon2012. 18.05.2015.

White, T. 2012. Hadoop: The Definitive Guide, Third Edition. O'Reilly press.

Wikipedia. 2015a. Big data. http://en.wikipedia.org/wiki/Big_data. 22.04.2015.

Wikipedia. 2015b. Comma separated values. http://en.wikipedia.org/wiki/Comma-separated_values. 04.05.2015.

Wikipedia. 2015c. Data warehouse. http://en.wikipedia.org/wiki/Data_warehouse. 30.04.2015.

Wikipedia. 2015d. Internet of Things. http://en.wikipedia.org/wiki/Internet_of_Things. 21.04.2015.

Wood, A. 2015. The internet of things is revolutionising our lives, but standards are a must. The Guardian. http://www.theguardian.com/media-network/2015/mar/31/the-internet-of-things-is-revolutionising-our-lives-but-standards-are-a-must. 21.04.2015.

Appendices

Appendix 1 Terminal output of MapReduce Job

```
hadoop@localhost:~/NetBeansProjects/MaxVehicles/target$ hadoop
fi.karelia.maxvehicles.MaxVehicles /user/hadoop/input/data.csv
output
15/05/19 13:36:19 INFO client.RMProxy: Connecting to Re-
sourceManager at /0.0.0.0:8032
15/05/19 13:36:20 WARN mapreduce.JobSubmitter: Hadoop command-
line option parsing not performed. Implement the Tool interface
and execute your application with ToolRunner to remedy this.
15/05/19 13:36:20 INFO input.FileInputFormat: Total input paths
to process : 1
15/05/19 13:36:20 INFO mapreduce.JobSubmitter: number of
splits:1
15/05/19 13:36:20 INFO mapreduce.JobSubmitter: Submitting tokens
for job: job_1432031250876_0004
15/05/19 13:36:21 INFO impl.YarnClientImpl: Submitted applica-
tion application_1432031250876_0004
15/05/19 13:36:21 INFO mapreduce.Job: The url to track the job:
http://localhost:8088/proxy/application_1432031250876_0004/
15/05/19 13:36:21 INFO mapreduce.Job: Running job:
job_1432031250876_0004
15/05/19 13:36:34 INFO mapreduce.Job: Job job_1432031250876_0004
running in uber mode : false
15/05/19 13:36:34 INFO mapreduce.Job:  map 0% reduce 0%
15/05/19 13:36:48 INFO mapreduce.Job:  map 100% reduce 0%
15/05/19 13:36:54 INFO mapreduce.Job:  map 100% reduce 100%
15/05/19 13:36:54 INFO mapreduce.Job: Job job_1432031250876_0004
completed successfully
15/05/19 13:36:54 INFO mapreduce.Job: Counters: 49
     File System Counters
          FILE: Number of bytes read=2766
          FILE: Number of bytes written=216059
          FILE: Number of read operations=0
          FILE: Number of large read operations=0
          FILE: Number of write operations=0
          HDFS: Number of bytes read=936
          HDFS: Number of bytes written=16
          HDFS: Number of read operations=6
          HDFS: Number of large read operations=0
          HDFS: Number of write operations=2
     Job Counters
          Launched map tasks=1
          Launched reduce tasks=1
          Data-local map tasks=1
```

```
            Total time spent by all maps in occupied slots
(ms)=11106
            Total time spent by all reduces in occupied slots
(ms)=4170
            Total time spent by all map tasks (ms)=11106
            Total time spent by all reduce tasks (ms)=4170
            Total vcore-seconds taken by all map tasks=11106
            Total vcore-seconds taken by all reduce tasks=4170
            Total megabyte-seconds taken by all map
tasks=11372544
            Total megabyte-seconds taken by all reduce
tasks=4270080
      Map-Reduce Framework
            Map input records=25
            Map output records=24
            Map output bytes=2712
            Map output materialized bytes=2766
            Input split bytes=108
            Combine input records=0
            Combine output records=0
            Reduce input groups=1
            Reduce shuffle bytes=2766
            Reduce input records=24
            Reduce output records=1
            Spilled Records=48
            Shuffled Maps =1
            Failed Shuffles=0
            Merged Map outputs=1
            GC time elapsed (ms)=378
            CPU time spent (ms)=5910
            Physical memory (bytes) snapshot=446705664
            Virtual memory (bytes) snapshot=1356087296
            Total committed heap usage (bytes)=326107136
      Shuffle Errors
            BAD_ID=0
            CONNECTION=0
            IO_ERROR=0
            WRONG_LENGTH=0
            WRONG_MAP=0
            WRONG_REDUCE=0
      File Input Format Counters
            Bytes Read=828
      File Output Format Counters
            Bytes Written=16
```

Appendix 2 DataScheduler

```
package fi.karelia.publicservices.data;

import fi.karelia.publicservices.data.domain.City;
import fi.karelia.publicservices.data.domain.Resource;
import fi.karelia.publicservices.data.domain.Service;
import fi.karelia.publicservices.exception.HadoopException;
import fi.karelia.publicservices.util.XMLReader;
import java.io.File;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
* @author Ruben
*/
public class DataScheduler {

    private static volatile DataScheduler dataScheduler = null;

    private final ScheduledExecutorService executorService =
        Executors.newScheduledThreadPool(1);
    private final ScheduledThreadPoolExecutor dataExecutor =
        new ScheduledThreadPoolExecutor(1000);

    private final Runnable mainTask;

    private final Map<String, Long> modificationTimestamps =
        new HashMap<>();

    private DataScheduler() {
        mainTask = new Runnable() {
            @Override
            public void run() {
                // Fetch XML file data
                XMLReader r = XMLReader.getInstance();
                for (City city : r.getAllCities()) {
                    try {
                        // Update modified resources
                        updateModifiedResources(city);
                    } catch (HadoopException ex) {
                        Logger.getLogger(
                            DataScheduler.class.getName())
                            .log(Level.SEVERE, null, ex);
                    }
                }

                // Reschedule modified resources
                rescheduleModifiedResources();
            }
        };
```

```
}

public static DataScheduler getInstance() {
    if (dataScheduler == null) {
        synchronized (DataScheduler.class) {
            if (dataScheduler == null) {
                dataScheduler = new DataScheduler();
            }
        }
    }
    return dataScheduler;
}

public void initialize() {
    // Start the main task to pull all metadata from XML files
    executorService.scheduleAtFixedRate(mainTask, 30000,
        86400000, TimeUnit.MILLISECONDS);
}

public void updateModifiedResources(City c)
    throws HadoopException {
    // Check if city file has been modified since its last update
    File f = new File(c.getFileName());

    // Check if the file exists
    if (f == null) {
        System.out.println("City filename does not exist: " +
            c.getFileName());
        throw new HadoopException(
            "Error while trying to update city schedule for city "
            + c.getName() + ": File not found: " +
            c.getFileName());
    }

    // Check if our array of lastModified dates contains the
    // city name, if not we know this is a new added city
    System.out.println("Checking modification timestamps");
    if (!modificationTimestamps.containsKey(c.getName())) {
        // Add the new last modified date for this city file
        System.out.println(
            "New value added to modification timestamps for city:"
            + c.getName());
        modificationTimestamps.put(c.getName(), f.lastModified());
    }

    // Check if the lastModified date exceeds the last stored
    // lastModified date
    if (modificationTimestamps.get(c.getName()) >
        f.lastModified()) {
        // If the file wasn't changed since last time,
        // update nothing
        System.out.println("Nothing to update for city: "
            + c.getName());
        return;
    }

    // Loop through all resources to check if they are
    // modified or added
    for (Service s : c.getServices()) {
        for (Resource r : s.getResources()) {
            boolean present = false;
            // Loop through our scheduled resources to check
```

```
                // if the current one is running
                for (Runnable task : dataExecutor.getQueue()) {
                    DataScheduledTask dpt = (DataScheduledTask) task;
                    if (dpt == null) {
                        throw new HadoopException(
                            "Wrong task type assigned");
                    }
                    if (dpt.getResource().getId() == r.getId()) {
                        // Update resource on the DataScheduledTask
                        System.out.println("Resource already running:"
                            + dpt.getResource().getName());
                        dpt.setResource(r);
                        present = true;
                        break;
                    }
                }

                // Check if the resource is a new resource
                if (!present) {
                    // Add new task to the pool
                    System.out.println("Added new scheduled resource:"
                        + r.getName());
                    DataScheduledTask dpt = new DataScheduledTask(r);
                    dataExecutor.scheduleAtFixedRate(dpt, 0,
                        r.getSchedulingInterval(),
                        TimeUnit.MILLISECONDS);
                }
            }
        }

        // Loop through all scheduled resources to check if there
        // are deleted resources
        for (Runnable task : dataExecutor.getQueue()) {
            DataScheduledTask dpt = (DataScheduledTask) task;
            boolean deleted = true;
            for (Service s : c.getServices()) {
                for (Resource r : s.getResources()) {
                    if (r.getId() == dpt.getResource().getId()) {
                        deleted = false;
                    }
                }
            }

            // If the resource is no longer present in the XML
            // config files, cancel the scheduled resource
            if (deleted) {
                System.out.println("Deleted scheduled resource:"
                    + dpt.getResource().getName());
                dpt.cancel(true);
            }
        }
    }

    private void rescheduleModifiedResources() {
        // Implementation required
    }
}
```

Appendix 3 DataScheduledTask

```
package fi.karelia.publicservices.data;

import fi.karelia.publicservices.data.domain.Resource;
import fi.karelia.publicservices.data.domain.SchedulingType;
import java.io.IOException;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.RunnableScheduledFuture;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
* @author Ruben
*/
public class DataScheduledTask implements RunnableScheduledFuture {

    private Resource resource;
    private DataPuller puller;
    private boolean running;
    private boolean cancelled;
    private long startTime;

    public DataScheduledTask(Resource resource) {
        this.resource = resource;
        cancelled = false;
        running = false;
    }

    @Override
    public boolean isPeriodic() {
        return getResource().getSchedulingType() !=
            SchedulingType.INITIAL;
    }

    @Override
    public void run() {
        try {
            running = true;
            startTime = System.currentTimeMillis();
            System.out.println("Start running task with resource id:"
                + resource.getId());
            puller = new DataPuller();
            puller.pull(getResource());
            puller = null;
            running = false;
        } catch (IOException ex) {
            Logger.getLogger(DataScheduledTask.class.getName())
                .log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public boolean cancel(boolean mayInterruptIfRunning) {
        if (puller.isProcessed()) {
```

```
            cancelled = true;
        }
        return cancelled;
    }

    @Override
    public boolean isCancelled() {
        return cancelled;
    }

    @Override
    public boolean isDone() {
        return !running;
    }

    @Override
    public Object get() throws InterruptedException,
        ExecutionException {
        return null;
    }

    @Override
    public Object get(long timeout, TimeUnit unit) throws
        InterruptedException, ExecutionException, TimeoutException {
        return null;
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return 0;
    }

    @Override
    public int compareTo(Object o) {
        return 0;
    }

    public Resource getResource() {
        return resource;
    }

    public void setResource(Resource resource) {
        this.resource = resource;

    }
}
```