Beehive, A MapReduce framework for highly distributed swarms

A.P. Sivakumaran Gheethan

Thesis submitted for the degree of Master in Network and system administration
30 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2017
Beehive, A MapReduce framework for highly distributed swarms

A.P. Sivakumaran Gheethan
Abstract

The exponential increase in data generated gives rise to more demanding requirements for processing it. A parallel programming paradigm known as MapReduce, has been used widely to process this data. There exist frameworks that have incorporated MapReduce style processing. However, these frameworks are not focused to be energy-efficient.

We investigate a novel approach for processing large amounts of data. This is done by developing a framework that supports distributed processing, with focus on utilizing green energy. The main aim of this project is to facilitate processing in data centers powered by renewable energy. This is attempted by combining technologies used in green computing, grid computing and cloud computing.

The results from this project were a fully developed task tracker along with reference implementation of the other components. At the current stage of development there was only found to be a 2% decrease in performance when compared to a local cluster. This project was also proved ready for future work, that would contribute to the main aim of the framework.
Contents

1 Introduction .............................................. 1
   1.1 Problem statement ................................. 2

2 Background .............................................. 5
   2.1 Climate Change ..................................... 5
   2.2 Virtualization .................................... 6
   2.3 Data Center ....................................... 8
   2.4 Cloud Computing .................................. 8
   2.5 Green Computing .................................. 11
   2.6 Big Data .......................................... 13
   2.7 Grid Computing .................................... 16
   2.8 Big Data Frameworks ......................... 17
      2.8.1 Batch Only Frameworks ..................... 18
      2.8.2 Stream Only Frameworks ................... 18
      2.8.3 Hybrid Frameworks ......................... 18
   2.9 MapReduce ........................................ 19
      2.9.1 How Google implemented MapReduce ......... 20
      2.9.2 Apache Hadoop .............................. 21
   2.10 Related Work in improving MapReduce ........ 23
   2.11 Tools used in implementing the project .... 24
      2.11.1 Docker ...................................... 24
      2.11.2 Python scripting ............................ 25
      2.11.3 JSON for communication ................... 25
   2.12 Summary .......................................... 26

3 Approach .................................................. 27
   3.1 Objectives ........................................ 27
   3.2 Design phase ....................................... 28
   3.3 Implementation phase ......................... 29
   3.4 Experiment phase ............................... 30
   3.5 Analysis phase ................................... 31
   3.6 Expected Project Deliverables ............... 32
   3.7 Summary .......................................... 33

4 Results I: Design ..................................... 35
   4.1 The Big Picture and where this project fits in 35
      4.1.1 Keywords and Terms ......................... 36
List of Figures

2.1 The 2 types of hypervisors. .................. 7
2.2 The 3 types of services the cloud provides. ......... 10
2.3 The top part of the illustration shows a situation, where the servers are underutilized. By consolidating the load, 2 out 3 servers are in an idle power mode consuming energy. .................. 12
2.4 Amount of global data over the years [24] [38] [15]. ........ 14
2.5 How data was stored in 1986 [24]. .................. 15
2.6 How data was stored in 2007 [24]. .................. 15
2.7 How Grid Computing Works. .................. 17
2.8 1. Master nodes assign roles to the workers (map or reduce). 2. The map worker retrieve a piece of data and perform the map function. 3. The mapped results are stored temporarily. 4. The reduce workers access data from intermediate store and reduce the data. 5. The reduced results showed as output .................. 20
4.1 1. The corpus is split into shards. 2. These shards are included in workers and deployed into the cloud. 3. The workers connect to the task tracker. 4. The user sends a job to the task tracker. 5. The task tracker sends a reduce job to a reduce worker. .................. 38
4.2 One of the ambitions of the framework is that the workers can migrate from data centers using brown energy to green energy ........ 40
4.3 The 2 distinct areas of management. The main tasks under each type of management is defined .................. 41
4.4 A timeline of what messages are passed in order to understand how the framework will function .................. 53
4.5 A predicted job scenario with 4 workers. The graph shows the different timestamps collected .................. 55
5.1 The structure of the statistics tree. The character S represents a Shard. The character J represents a Job. .................. 68
5.2 A detailed graph showing the start and stop times for each worker for a job. The x axis shows the time used and the y axis shows the worker ids. .................. 87
5.3 A detailed graph showing the progression of $D_t^{Max}$ for an experiment. The x axis shows the job ids and the y axis shows the time in seconds. .................. 88
5.4 A detailed graph showing start and stop times for each job. The blue line shows when the job was issued and the red line shows the time finished. The black lines in parallel show the time used for each individual job. ............................................. 89

6.1 An experiment conducted when the cluster is set to run 30 jobs. The blue line shows the time issued per job and the red line shows the time finished. The progression is described to be linear. ............................... 93

6.2 Graph depicting the effect of a slow worker on overall performance. As seen from the graph the red line \(T_{\text{issued}}\) gets farther away from the blue line \(T_{\text{finished}}\) as the jobs progress. ................................. 96

6.3 The first job processed. The graph shows that all workers begin at almost the same start time. The blue line shows the influenced worker. ......................................................... 96

6.4 The last job processed. Compared to 6.3, the blue line shows that the worker starts much later than all other workers. ................................. 97

6.5 The graph of \(D_{\text{Max}} T_{\text{issued}}\) vs job id. The increasing \(D_{\text{Max}} T_{\text{issued}}\) shows that overall completion time increases. ................................. 97

6.6 The graph of \(D_{\text{Max}} T_{\text{issued}}\) and \(D_{95} T_{\text{issued}}\) vs job id. ................................. 98

6.7 Graph of job scheduling when a duplicate worker is introduced. The blue line shows the \(T_{\text{issued}}\) and the red line \(T_{\text{finished}}\). The dotted green line shows the predicted line without the duplicate worker. Compared to graph 6.2, the jobs finish much quicker. ................................. 99

6.8 The graph of \(D_{\text{Max}} t_{\text{issued}}\) vs job id. The value of \(D_{\text{Max}} t_{\text{issued}}\) decreases after the duplicate worker is introduced. ................................. 100

6.9 Job distribution between the 2 corpuses. Jobs belonging to one corpus is not affected by the other. ................................. 101

6.10 Double cluster performance. The orange box shows the progression of the jobs after the cluster is double. The blue line shows \(T_{\text{issued}}\) and the red line shows \(T_{\text{finished}}\). The orange box shows 15 jobs being completed faster. ................................. 101

6.11 The graph of \(D_{\text{Max}} t_{\text{issued}}\) vs job id. ............................................. 102

6.12 Worker distribution from a job in the first half. ............................................. 103

6.13 Worker distribution from a job in the second half ............................................. 103

6.14 Distributed worker placement. The green dot shows the position of the task tracker. The red dots shows the worker placements. ............................................. 104

6.15 Job distribution in a distributed worker environment ............................................. 105

6.16 Job distribution in a local cluster ............................................. 105
List of Tables

2.1 The different Openstack flavours that are offered by the ALTO cloud ........................................... 9
3.1 Summary of the approach .............................................. 33
6.1 List of experiments performed ................................. 107
Acknowledgement

This thesis would not have taken form if not for the following.

- My supervisor and mentor Kyrre M. Begnum. Thankyou for your guidance and undivided attention throughout the master program and this thesis. Your positive thoughts and attitude was always a motivation.

- To all my professors throughout the master program. Thankyou all for a very fun, interesting and knowledge-rich master program.

- My parents and sister living in Oman. Thankyou for all your prayers and motivational words.

- My family in Norway. Thankyou for always keeping the doors open to good food when I most needed it.

- My friends throughout the master program. Thankyou for all the good times and help.

- The University of Oslo and the Oslo Akershus University College.

- The Almighty God, for giving me the strength and blessing to complete this master program.
LIST OF TABLES
Chapter 1

Introduction

A rising concern is the increase of carbon levels over the past few years. This has posed risks such as rising sea levels, global temperature and many more. NASA reports that the CO2 levels have increased rapidly by almost 100 parts per million (ppm) [32]. The burning of fossil fuels is one of the main reasons that cause these effects. Fossil fuels are burned to create energy in the form of electricity which powers the digital world. Greenpeace reports that over 12% of global electricity is demanded by the IT sector. The number is predicted to increase 7% annually.

One of the more requiring demands of modern IT is powering data centers. These are clusters of machines running together, providing resources to process and store data. One of the main services running on data centers is the Cloud. Cloud giants such as Amazon and Google provide these resources at a lower cost and on demand. One of the main advantages in offering such resources is that processing complex data is done without the need of setting up hardware privately. In its latest report Greenpeace predicts that the number of data centers required is expected to increase by almost 10 times by 2030 [15].

The need for more data centers occur due to the exponential increase in devices connected to global Internet and the data generated. In 2015 the amount of data on the digital world was estimated to be 4.4 zettabytes [15]. This number, is expected to increase 10 fold by 2020. Google reports that it currently processes almost 40,000 search requests per second, each which differ vastly in its nature. To prevent latency the service has to respond to requests at tremendous speeds. The volume of data creates a very unpredictable scenario for companies that plan on acquiring hardware to process and store this data. Therefore have companies looked to the cloud for renting processing power and storage. Currently, the trending methods of processing this data are by using frameworks such as MapReduce. These frameworks prove to be useful in processing data, but inefficient as idle machines consume electricity [27].

Due to the high energy requirement of data centers, a trend to run data centers on renewable energy has been developed. Most of the cloud
providers have a renewable energy factor depicting the green-ness of the cloud. Having a cloud depending on renewable energy depends on the availability of green-energy. This means that when in depletion of green energy, the data center switches to non-renewable energy, in short periods of time.

In the field of cloud computing, research has been performed towards developing greener solutions. One such is live migration of virtual machines to reduce energy consumption. The virtual machines are either migrated inside the same data center or even to other data centers. One such algorithm is "Follow-the-sun". The algorithm attempts to migrate virtual machines to data centers based on the availability of green power. The above mentioned Big data frameworks are limited to a single local cluster, thereby not taking advantage of these solutions. So what should be done to modify these traditional frameworks? How effective will it be?

The aim of this thesis is to explore how a MapReduce like framework works in a distributed and migratory environment. To have a distributed solution, there has to be a task tracker in place that can schedule and order tasks for the virtual machines to perform. A secondary goal of this thesis is to provide a path to future research into a novel framework which takes being green into consideration.

The potential gain by developing the prototype is that, one main piece of the puzzle in creating a distributed Big data processing framework is possibly solved. This would then facilitate that the virtual machines migrate to data centers with access to ample green-energy. By running such a solution only on green-energy, the data centers would require less fossil fuels to power that data center. This would thereby contribute in reducing the carbon emissions caused by data centers and all the adverse effects caused.

1.1 Problem statement

Design and develop a task tracker for dispersed and dynamic MapReduce-like frameworks in order to investigate the feasibility of follow the sun enabled big data clusters.

Task tracker in the problem statement refers to the solution that is to be developed. This entity is to have the function of keeping order of tasks and act as an asynchronous, centralized task scheduler.

Big data cluster is group of machines that is specifically tasked to processing big data. The Hadoop Cluster is an example of a MapReduce framework that is used to process and store big data. Companies such as Google and Facebook have clusters as big as 4500 nodes with almost 100,000 cores for processing.

Investigate refers to analyzing how this solution fares against the
traditional MapReduce and give insight on new potential hurdles.

The phrase *dispersed and dynamic*, describes the goal of this thesis, by having processing machines dispersed among data centers. The term dynamic describes how the virtual machines migrate between data centers depending on the availability of green-energy.

*Follow the sun*, describes the nature of the framework. This framework shall take into consideration the source of the energy that is used to power the machines. Green energy is the preferred type of energy here.

The term *MapReduce like* hints at a variation on an existing well known frameworks. The modification shall implement qualities that are required for a follow-the-sun enabled framework.
Chapter 2

Background

This chapter will give detailed information on the above mentioned technologies and concepts. Related work conducted in improving existing frameworks is also explored. Lastly, tools that are used to realise the project are introduced.

2.1 Climate Change

Climate change is one of most concerning topics that has prevailed the world in the last years. The global temperature of the earth has risen by almost 1 degree [32]. Although this may not seem as much the graph shows an alarming rate in the rise of temperature over the past few years. One of the main reasons for this temperature change is the burning of fossil fuels such as coal to generate energy. As the number of electronic devices and motor vehicles increase, larger amounts of fossil fuels are required to be used. Generating this amount of energy is not difficult compared with the other types such as solar, wind and hydro. However, the downside of using coal energy is the carbon dioxide that is emitted.

Carbon dioxide poses adverse effects to the earth. The increase in levels of carbon dioxide has caused the earth’s surface to be warmer. NASA reports to have measured the highest amount of carbon dioxide levels in the atmosphere measured in 2016 [32]. The signs of rising carbon dioxide levels are appaling. Apart from rising global temperature, rise in global sea levels by 17cm, melting icesheets in the Arctic, decreased snow cover and the increasing occurrences of extreme events such as tsunamis and hurricanes are some of the effects. A helpful way to reduce carbon dioxide levels are forests due to the vast amount of trees. However, WWF reports that deforestation is happening at the large and almost 56,000 square miles of forest is being cut down every year [41].

To combat these effects alternate energy sources have been experimented on. Solar, hydro and wind energy are examples of these, or also known as green energy. These energy sources get their denominator ‘green
energy’ due to having null carbon dioxide emissions. A downside of this type of energy is that it is not fully developed to replace the amount of energy required produced by burning coal. According to Greenpeace, [21] there are currently 2 main problems regarding green energy.

1. Green energy is not available all day. There are periods of time where there is no local source for green energy. For example at night or on a day with no wind, solar and wind energy cannot be generated. This can cause a deficit in energy resulting in power outages.

2. Based on current use of energy, if the world was to switch to only green energy, the result would be not having adequate energy to power everything. Germany, which already uses 25% renewable energy, aims to be 80% renewable by 2050. This highlights the need of more efficient solutions where more work is done for the energy used.

2.2 Virtualization

Virtualization is a way to abstract the hardware and system resources from the operating system [42]. This means that the physical hardware (Memory, Processor and Storage) also known as the host machine, either shares or splits up its resources with a “virtual” or guest machine. This technology makes it possible for many virtual machines to run on one host machine. Powering this technology requires a layer of abstraction acting as a wall to separate the physical from the virtual. This is known as the hypervisor.

The hypervisor is a piece of software that enables to create and power virtual machines, including the task of sharing the host machine’s resources. These machines can run their own version of operating system, without interfering with the host machine. Even though there can be many virtual machines running on the host machine, each virtual machine will be isolated from one another. There are 3 types of virtualization present. Full virtualization, Partial virtualization and para virtualization. The first two of the three have 2 different type of very common hypervisors.

1. Type 1 hypervisors provide full virtualization also known as hardware virtualization. As the figure 2.1 on the left illustrates, there are only 3 layers present. The hypervisor abstracts the available physical hardware for the virtual machines. These virtual machines have their own operating system and set of applications. This type of virtualization does not require an operating system to be installed on the host machine, but rather is installed on top of the physical hardware. This type of virtualization is very common in data centers. By comparison this type of setup has a performance advantage over the type 2 virtual machine since it only requires instructions to go through 2
2.2. VIRTUALIZATION

Figure 2.1: The 2 types of hypervisors.

layers as seen from the figure 2.1. A downside of this type of virtualization is limited hardware support. Microsoft Hyper-V, VMware ESXi, Citrix ZenServer are examples of Type 1 hypervisors.

2. Type 2 hypervisors provide partial virtualization also known as software virtualization. As seen from the 2.1 on the right, type 2 virtualization is installed on top of a host OS. The host user here has full control on provisioning the resources to every virtual machine. By comparison to a Type 1 hypervisor, type 2 is easier to setup and has a lower rate of hardware compatibility issues. This type of hypervisor is common in client use. The downside of this type of hypervisor is that it is comparatively slower than the type 1 hypervisors due to the extra host OS layer. Another security flaw with type 2 hypervisors is that a breach in the host OS affects all virtual machines. Virtual Box are examples of type 2 hypervisors.

By using virtualizing technology there are a number of benefits.

1. Live migration of virtual machines make it possible to move a virtual machine from one host machine to another. More on this in the section on Green Computing.

2. Virtual machines provide a more secure environment to test out program or piece of code. Any harm done to the machine is only done to the virtual machine, without doing extensive damage to the hardware.
3. A powerful machine or a server can be used to provision many virtual machines thereby providing virtual machines to many clients.

### 2.3 Data Center

As the name specifies, a data center is a center that facilitates high performance computing. It comprises usually arrays of servers, storage and networking equipment. In addition to all computing hardware, datacenter incorporates uninterrupted power source such as diesel generators and fire safety equipment. A data center can usually host more than a 1000 servers. Google reported to have almost 2.5 million servers [25]. This number is expected to increase when expanding or building newer datacenters. Service providers such as Google, Facebook and Amazon have datacenters in many locations around the world. Facebook has a current total of 6 data centers in the United States, Sweden and Ireland. Google has 14 datacenters in North and South America, Asia and Europe.

All this computational power produces a lot of heat which can be damaging to the components if not cooled properly. Data centers incorporate advanced cooling mechanisms by using air and water. A method used widely is pumping cooled air from underneath through the racks to transport heat out of the top of the rack. The warmed air is then cooled or transported out of the data center [10]. Other cooling techniques include using water as google does in its data center located in finland [6]. Data centers are known to consume most energy to power and cool down the equipment.

The Green Peace report from a year shows that data centers consume almost 300TWh per year which is almost 2% of global electricity [15]. Electricity powering a data center is usually generated by the burning of fossil fuels, nuclear power, gas and various forms of renewable energy such as solar, wind and hydro. All data centers have a green index, which classifies the percentage of renewable energy usage in a data center. According to the report, giant cloud service providers such as Google, Facebook and Apple had a higher green index compared to Amazon data centers. The consumption of electricity of data centers is expected to increase by at least 7% per year. By 2030 data centers are predicted to consume atleast 13% of global electricity. The predicted emissions from datacentres, telecommunication and terminal in 2020 is expected to be around 1430 million tonnes carbondioxide.

### 2.4 Cloud Computing

Cloud computing refers delivery of services over Internet based computing. The cloud is hosted in data centers. The services delivered can vary from the need to host a specific application, or hardware resources such
as processing power and vast amount of storage. The cloud is hosted on
data centers boasting tremendous hardware power. This combined with
software is known as the cloud [3]. The cloud is usually categorized as
public and private clouds. Public clouds are mostly accessible anywhere.
Use of cloud services such as Gmail, iCloud, Dropbox have been increas-
ingly popular due to the availability via the internet [23]. Private clouds
are usually setup in business for internal uses.

The Oslo and Akerhus College hosts an private OpenStack cloud, the
ALTO cloud. The ALTO cloud is primarily used for educational purposes
to facilitate teaching and researching. It runs on 2 racks of servers featuring
10 compute nodes with a total of 64 cores. It features also 4 terabytes of
RAM and almost 120 TB of storage capacity. Openstack features virtual
machines in different flavors. A flavor is a specification of a virtual machine
on the amount of Virtual CPU’s, RAM and storage available. The table 2.1
shows the flavors Openstack provides.

The Amazon cloud service offers a range of facilities such as compute
power, database storage and content delivery. It has over 42 availability
zones in 16 different geographic reasons. By using the Amazon Elastic
Compute Cloud (EC2) service, customers can launch virtual machines
different types and flavors in minutes, thereby giving instant access
to computing power. Companies such as Netflix, Expedia, Coursera
and many more are using this service to scale applications at ease when
required [1]. Amazon also offers storage solutions via the Amazon Simple
Storage Service (S3). Storage is offered here for almost $0.023 Gigabytes per
month [2].

Traditionally, scalability was more of a concern where hardware
upgrades and maintenance to facilitate computing required hours of planning
and implementation. The attractive aspect of the cloud contra traditional
computing is the elimination of personal investments, maintenance and
upgrade in hardware. The cloud offers this by renting out services or
known as pay-as-you-go plans[3], which requires customers to pay only
for uptime. As an example, Amazon offers a Linux small flavored instance
(see 2.1) for as less as $0.023 per hour [1]. Clients requiring a scalable
infrastructure should only then discuss with the cloud provider and agree
upon the new requirement. The quality of the service is agreed by a Service

<table>
<thead>
<tr>
<th>Name</th>
<th>VCPUs</th>
<th>Disk Size</th>
<th>VRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1.micro</td>
<td>1</td>
<td>0 GB</td>
<td>64 MB</td>
</tr>
<tr>
<td>m1.tiny</td>
<td>1</td>
<td>1 GB</td>
<td>512 MB</td>
</tr>
<tr>
<td>m1.small</td>
<td>1</td>
<td>20 GB</td>
<td>2 GB</td>
</tr>
<tr>
<td>m1.medium</td>
<td>2</td>
<td>40 GB</td>
<td>4 GB</td>
</tr>
<tr>
<td>m1.large</td>
<td>4</td>
<td>80 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>m1.xlarge</td>
<td>8</td>
<td>160 GB</td>
<td>16 GB</td>
</tr>
</tbody>
</table>

Table 2.1: The different Openstack flavours that are offered by the ALTO cloud
Figure 2.2: The 3 types of services the cloud provides.

Level Agreement (SLA). This document is an agreement on the promises the cloud provider issues the client with regards to the service.

The cloud currently provides 3 service models.

1. Software as a Service (SaaS)

2. Platform as a Service (PaaS)

3. Infrastructure as a Service (IaaS)

As seen from the figure 2.2, the 3 types of services depict the amount of freedom provided in using the service. This means that one dealing with an IaaS structure gives greater freedom in customizing the machines compared to a SaaS user.

Software as a Service (SaaS)

Software as a service is simply delivering applications through the network hosted by the cloud. As seen from the figure 2.2, this is the smallest of the triangle, due to the services offered being limited. Among the services usually offered, the common ones are email, storage and office tools. Applications such as Dropbox and Office 365 are all SaaS applications. These applications are accessed usually by a client such as a web browser. Dropbox for instance offers unlimited storage, advanced sharing, integration with Microsoft Office and access anywhere for only €15 a month.
Platform as a Service (PaaS)

Platform as a service enables the customer to run self-developed applications in vendor-provided frameworks [28]. As seen in the figure 2.2, this portion is bigger than SaaS but smaller than IaaS. These types of services offer more freedom to the customer to run self-developed applications, but are restricted to the functions supported by the framework. Amazon offers this type of service under the name Elastic Beanstalk. Node.js, Python, Ruby are examples of frameworks the service supports. The service provider usually handles infrastructure services such as load-balancing and scaling.

Infrastructure as a Service (IaaS)

Infrastructure as a service providing the customer with bare hardware such as processing, storage and networking. As seen from the figure 2.2, this is the largest portion. What this means is that the customer then has the greatest freedom in customizing on the functions of each server and define communication between them. IaaS also gives great freedom in choosing the operating system and software that is being run. This type of service is useful for customers that setup their own infrastructure.

2.5 Green Computing

In short a phrase, Green Computing means energy efficient computing. Energy efficient computing is a very progressing field of technology and for every year that passes by, more efficient devices are being made. Both personal and enterprise groups users have benefitted from greener technologies. This is usually done by producing lower energy consuming hardware and incorporating smart algorithms to reduce workload. For example, both mobile and computer processors are equipped with CPU throttling, which reduces the CPU speed, when not in use. Slower speeds translate then to lowered energy use. Smaller scale reductions in the use of energy has benefitted larger scale computing environments.

Data centers have incorporated more green computing in many forms. An example of this is constructing a data center in places where there is a good and steady supply of some form of renewable energy to power devices. According to the GreenPeace report the Apple cloud currently run on 83% renewable energy [15]. Research is being performed on reducing the energy consumption of data centers. An example of this is server consolidation, where the workload from many servers is consolidated into one server, to ensure that the other servers can be throttled down. This is done by migrating the virtual machines from one server to another, with no noticeable downtime. The major advantage of a solution such as this is that total energy usage is reduced since other servers are shut down or running on a low power mode [43]. As seen from the figure 2.3, the upper part of the...
Figure 2.3: The top part of the illustration shows a situation, where the servers are underutilized. By consolidating the load, 2 out 3 servers are in an idle power mode consuming energy.
illustration shows 3 servers that are capable of hosting 10 virtual machines each. The first situation shows the servers running at 20% capacity. By migrating all the virtual machines to server 1, the other servers can be put to a very low power mode thereby saving the total energy used. These type of algorithms are known as power-aware algorithms since they take into consideration total power being used [43].

Research has been also performed on developing minimal operating systems. Minimal Operating systems feature only the very crucial components that are required for an operating system to boot up. Such images are very small in size and thereby reduce boot up time. An example is a unikernel operating system, such as IncludeOS, which is only 158KB in size compared to a minimal Ubuntu16.04 image which is roughly 55MB. The result of this is that an IncludeOS instance takes only 0.3 seconds to boot up. A DNS service running on Ubuntu and IncludeOS was compared and results showed up to 20% fewer CPU ticks compared to a Ubuntu system[5].

As seen from the illustration 2.3, the end state of the servers are a low power mode. These servers still consume energy to be in this power mode due to cpu cycles being used to run background processes. A report from the Anthesis Group mentions that 30% of all servers are in a constant idle mode [26]. The 30% constitutes to around 10 million servers. The estimated energy cost used to power these idle servers are around $30 billion. The advantage of a unikernal operating system such as IncludeOS is that it uses no cpu cycles when idle [5]. This is very cost effective in a larger scale.

2.6 Big Data

Digitalization of traditional non-IT methods have revolutionized the working methods in many fields. For example, traditionally looking up records required sifting through pages of handwritten records. Currently this method is digitalized where the data is recorded on databases. By storing data on databases, processing this data such as sorting, searching and transferring data is hassle free. The diagrams below figure 2.5 and 2.6, show how the methods of storing data have changed from analog in 1986 to more digital in 2007. The amount of data has also increased from 2.6 exabytes in 1986 to almost 295 exabytes in 2007. The use of IT in many fields has exponentially increased the data that is stored. This is shown in the figure 2.4, which shows the amount of data estimated present in each year [24] [38] [15], and the exponentially increasing level.

One of the other reasons of exponential increase in data is primarily due to advancement of technology and the affordability of technology. Contributing to this volume of data is not only computers and mobile devices but also sensors, smart devices and surveillance cameras and microphones [31]. In the annual Facebook report from 2016 almost 1.86 billion users use Facebook. These active users sift through a variety
CHAPTER 2. BACKGROUND

Figure 2.4: Amount of global data over the years [24] [38] [15].

of content such as photos, videos and status updates. This generates a huge amount of data traffic that flows in and out of data centers hosting Facebook and was estimated up to 10PB per month. A common misconception of the term Big Data refers to only the size of the data, which is not the only problem regarding it. Data not only has to be stored but also processed and delivered in a timely fashion. This phenomenon of Big Data is used to refer to the increase in the volume of data that gives rise to difficulties in storing, processing and analyzing it through traditional technologies [23].

Big Data is characterized in many ways by different authors. The most common characterization are the 3V’s. Volume, Velocity and Variety. Some characterize it with 1 or 2 additional V’s, Value and Veracity [4]

- **Volume**, refers to the amount of data that is gathered [23]. The data gathered is used for data analysis and helps to find patterns and anomalies. In 2011 the amount of data found was 1.8ZB (1.8 billion terabytes)[12]. The GreenPeace report Click clean reports that the amount of data found in 2015 was 4.4ZB. The volume of data flowing through eBay and Google were estimated to be 100 PB.

- **Velocity**, refers to the speed at which the above mentioned data is produced and processed [23]. Velocity of the data being produced and processed is crucial in applications such as finance, medical and intrusion detection systems where due to time being a deciding factor. The velocity factor is dealt through smart processing algorithms involving machine learning and high performance clusters. Facebook reports processing data at hundreds of gigabyte per second [22]. In many enterprises the rate at which data is processed directly affects
the quality of service. Part of the Big data problem is the inability to process data using traditional data structures.

- **Variety**, refers to the different types of data that is collected from the different inputs [23]. Weather information, sensor data, financial data, pictures are all examples of data. Data is therefore classified in 3 forms. Structured, Semi-structured and Unstructured. Structured data such as retail and financial always possess certain characteristics. An item on sale always has a name in the form of a string and a price which is always a positive number. Due to the structure of data, it can be easily processed when stored in SQL databases. Semi-structured data is data that does not conform to a rigid structure, but enclosing fields are present that is common to all data. Emails do not always have a subject name but they data such as date and time can always be extracted. Unstructured data such as images, JSON strings are data that does not contain any form of structure [31]. Unstructured data
poses challenges when analysing data as to what and how to extract meaningful content [4].

Big data has been proven useful in many of fields such as medical, particle physics, earth science, astronomy and social computing [11]. CERN in Switzerland, is home to the largest science experiment, the Large Hadron collider. Accelerated particles are collided with one another almost 600 million times a second. Data that is recorded here is used to recreate the collision and analyze the data to find out about new particles. It takes up roughly about 30 petabytes [7]. In a survey conducted by McKinsey institute in 2012, over 51% of enterprises hoped that gathering data would contribute to increasing the operational efficiency [29]. Another well known example where big data is increasingly used, is in the automobile industry. Automotive manufacturers such as Mercedes-Benz, BMW and Volkswagen have incorporated safety mechanisms such as front collision safety, blind zone assist and a tiredness warning [17]. This is done by recording and analyzing information from sensors mounted around and in the car. These manufacturers aim to increase car safety and ride comfortability by also evaluating data found on cars to adapt newer methods. Future projects in all fields of study are proposed to collect data in huge amounts, requiring some form of analysis to possibly contribute to research [11].

2.7 Grid Computing

A collection of computer nodes connected together performing jobs related to a single task is known as a Grid Computing. Simply described it is a form of distributed computing. By connecting a number of computer nodes together, the grid provides an abstract of a supervirtual computer that is used to solve problems in large data sets [30]. By the term computing node, the grid can use all type of machineware from basic commodity desktop computers to powerful servers. These computing nodes need not to be tightly coupled as in a data center but can be connected over a simple ethernet connection or over the internet. Due to the pool of resources available at the grid, the task in concern has to be parallizable. One of the biggest Grids in Europe is the European Data Grid Project. The goal of this project is to facilitate scientific exploration that produces petabytes of data [34]. The grid was used mainly at CERN for recording and analysing data produced from the experiments at the Large Hadron Collider (LHC). The European Data Grid was also used to analyse data in the fields of earth sciences and bio informatics. Currently CERN uses its own LHC Computing Grid to perform calculations [9]. The Grid links more than 170 computing centers in over 40 countries. It has 4 tiers of classification which specify the type of service provided [8]. Tier 0 which is located at the CERN center processes over 1 million jobs per day with peak data outputs of upto 10 gigabytes per second.
2.8. **BIG DATA FRAMEWORKS**

As shown in the illustration 2.7 above Grid computing to work, 3 elements have to be present. A control node or a main machine, computing nodes running network software and a collection of software known as the middleware [36]. The control machine has the job of dividing the tasks to all the other nodes. It also has the responsibility of load monitoring and maintaining a good user experience for the compute nodes. Communication between the nodes is achieved by the middleware. This piece of software enables to run processes on the other computing nodes. Middleware also enables the user to submit, view status and retrieve the output of jobs.

The main advantages by using the Grid is that any machine with processing power independent of the operating system can join a Grid, and either use or be used to process tasks by efficiently utilizing power from underutilized computing nodes. In case a node is being utilized to the maximum locally, the grid makes sure the task is then run on another computing node on the grid thereby providing load balancing on the grid. Another main advantage of using the grid is that large datasets such as weather data, can be processed efficiently due to parallelizable tasks, thereby reducing the time consumed in processing.

2.8 **Big Data Frameworks**

As mentioned above in chapter big data, the volume of data requires a specific method of processing it, else there can be a lot of overlap in processing and analysis of data can take a longer period of time. These methods to process Big data effectively in a timely fashion are known as Big Data Frameworks. There are 3 types of frameworks.

- Batch-only Frameworks
- Stream-only Frameworks
• Hybrid Frameworks

2.8.1 Batch Only Frameworks

These type of frameworks are known as high latency frameworks. The applications of batch frameworks are usually where time is not a critical factor in such as retail, geodata and bioinformatics. Batch processing is suitable when a complete data set is required, in cases such as calculating sums and averages. This type of processing suits when the data set is very large thereby favoring the volume aspect of big data. A well known used example of batch processing is Apache Hadoop which uses MapReduce to compute. More on MapReduce later on.

2.8.2 Stream Only Frameworks

These type of frameworks are known as low latency or real-time frameworks. The difference between these frameworks contra Batch only is that time is a critical factor here. These frameworks are used while processing data such as intrusion detection, financial trading and medical systems. This type of processing suits when the data is supposed to be processed quickly favoring the velocity aspect of big data. Contra to Batch processing, data processed in real-time only takes into consideration a portion of the data set such as over the last hour. A well known used example of stream processing is Apache Storm.

2.8.3 Hybrid Frameworks

As the name suggests these type of frameworks can handle both batch and stream processing. Even though these frameworks handle both types of processing, there are a lot of variations in how they process data. Spark and Flink by Apache are examples of hybrid frameworks.

Apache Spark is a batch processing framework with stream processing enabled. The method in which Spark achieves streaming processing speeds is by using in-memory computation rather than reading from the disks. Various processing algorithms such as Directed Acyclic Graphs (DAG) also improve the speed of processing. DAG are used to show the relationship between the processing tasks and the data that is to be worked on. This results in more intelligent scheduling and higher speeds in processing [18]. Apache Spark is almost 100 times faster than Apache Hadoop [31].

Apache Flink is another stream processing framework which can handle batch loads. It acts upon batches of data as fixed sizes of streams to process them. Streams are processed by an item at a time. During the computation snapshots are created, for the purpose of recovery. The method in which Flink optimizes processing batch loads, is by analysing all
tasks that need to be done, giving knowledge on the actions and datasets to be performed upon [18].

2.9 MapReduce

MapReduce is a programming model developed at Google, used in the processing of tremendous amount of data in large scale computational systems [40] [13]. The approach to a MapReduce computation is largely similar to a divide-and-conquer scheme, meaning that data is broken down into smaller fragments to process[11]. The main advantage of MapReduce over the other programming models is the execution of tasks parallel fashion. This means that the computations are distributed across many machines to reduce time usage. The term Map and Reduce are 2 types of problem solving methods where the first maps the data set to a number of computational nodes, and the latter meaning to concise the output data. These methods are derived from existing functional programming languages such as Lisp. MapReduce was conceived at Google due to challenges faced in parallelizing the computation, distributing the data and handling failures [16]. A model was then required to process large amounts of raw data [16]. This model known as MapReduce, was a library that took care of all underlying computations and distribution of data, leaving the user with the only task of specifying the map and reduce functions. Since the birth of the MapReduce model over, 10,000 MapReduce programs are written and 100,000 jobs executed at Google [16].

Two main steps in how MapReduce is computed as seen from the figure.

1. Map function :- Processes the input pairs and returns a list of intermediated keys and values list. Simply put, this function groups the values by a user specified key and outputs values for the key types by mapping it into another domain for the reduce function [39]. This is summarized by the equation below.

   \[ \text{Map} (\text{key1},\text{value1}) \rightarrow \text{list} (\text{key2},\text{value2}) \]

2. Reduce function :- Concises the output produced by the map function and merges the results. By collecting the mapped results from the map functions, the results are reduced to one concise list sorted by combining values corresponding to the same domain [39]. This is summarized by the equation below.

   \[ \text{Reduce} (\text{key2},\text{list(values2)}) \rightarrow \text{list(value3}) \]
function map(name, document):
    for word in document:
        return (len(word), word)

function reduce(key, values):
    result = 0
    for value in values:
        result += length(values)
    return result

Listing 1: Pseduocode of a MapReduce function

Figure 2.8: 1.Master nodes assign roles to the workers(map or reduce). 2.The map worker retrieve a piece of data and perform the map function. 3.The mapped results are stored temporarily. 4.The reduce workers access data from intermediate store and reduce the data. 5.The reduced results showed as output

An example to a common MapReduce job is to find out the length of words in a bigdata set. The set is then split up to a cluster of compute nodes. These nodes now all contain a unique piece of the bigdata set. The Map function processes the data by returning an intermediated list. When each of the compute nodes return their intermediated lists, these are maybe sorted first and then merged together by worker nodes to give a final concise result. This is represented in the psedocode frame 1. As seen the map function transforms the key-value pair from a name of document-document content to length of word-word. The reduce function then groups the results from the map functions and returns the total results.

2.9.1 How Google implemented MapReduce

The figure above 2.8 shows how Google implemented the MapReduce programming paradigm. It operates on having a master node that tracks the job and assigns workers the roles required to perform the MapReduce
2.9. **MAPREDUCE**

1. The master node assigns roles to the workers. The roles that can be assigned are mapper or reduce. The master can choose also to change the role of the worker once the map function is completed.

2. The worker responsible for the map function read input splits from a local data store. They then perform the map function that is required.

3. Once the map computation is complete the data is written on an intermediate store.

4. Reduce workers then read data from this intermediate store.

This master node in this framework keeps in track of all map and reduce tasks. It stores the job status as either idle, in-progress or completed. The master node has also the responsibility of notifying the reducer node, when a map computation is complete. The results of this is a master node that is both reliable and fail tolerant [16].

Reliability is offered by a file system developed at Google, Google File System (GFS) [16]. It is a scalable distributed file system setup up on inexpensive hardware, that is used in large distributed applications [19]. As other file systems GFS offers performance, scalability, reliability and availability. In MapReduce each worker machines have an added storage drive, used to store data in. GFS then stores a copy of the data on 3 different machines, thereby ensuring no data loss in case of machine failure.

When managing more than a 1000 workers, failures occur periodically and must be handled without the execution of the job being disrupted. The master pings the worker nodes to check status. If a worker node working on an map execution task goes down, another worker on the same network switch containing the copy of the data is set to do the tasks executed on the failed machine. In addition to this all the reduce workers are informed of the map task failure and will not read the data from the failed worker. Reduce workers that fail do not have to have their tasks reexecuted, since this data is stored outside the framework.

### 2.9.2 Apache Hadoop

Hadoop is an open-source implementation of MapReduce in Java, licensed by Apache. It is mostly inspired by Googles version of MapReduce and in-house developed files system GFS. Since its birth, the Hadoop project been contributed to by large cloud service providers such as Yahoo, Facebook and IBM. Facebook has one of the largest Hadoop clusters processing almost 100 PB of data [33]. The Hadoop implementation includes 2 main components. The MapReduce and the Hadoop distributed file system (HDFS) [23]. Since, these 2 main components representing the processing
and storage system are highly related to each other, the ecosystem is co-deployed forming a single cluster [37]. There are also a number of smaller projects such as Hbase, Pig, Hive that support the Hadoop Ecosystem. Hadoop has been implemented in 2 major releases (1.0 and 2.0) with the latter adding significant components helping to remove the disadvantages from the previous one. Hadoop by default uses a First In First Out scheme to schedule jobs.

HDFS is a file system used to store data in the Hadoop ecosystem. It is usually deployed on a cluster of servers belonging to the same network. Similar in the way how GFS works, HDFS splits up files into smaller chunks of data and are replicated in each of the servers. This increases the reliability of the data and is not critical in the event of a node failing. HDFS uses a master-slave architecture to keep control of data. A dedicated server known as the NameNode is used for this purpose of storing metadata such as permissions and disk space quotas. The NameNode also records the location of each of the fileblocks Data is stored on the other servers known as DataNodes. A client requesting information of a specific file contacts the NameNode on the files location and then the content from the nearest node is the accessed [35].

In a similar way to how data is divided, processing also takes place in a master-slave architecture. This involves 2 component, JobTrack and the TaskTracker. The JobTracker assumes the role of the master and is responsible for scheduling jobs and distributing the tasks to the TaskTracker. The TaskTrackers assume the role of the slave and implement the jobs distributed by the TaskTracker. The JobTracker has also the role of monitoring nodes that fail execution of the task. It then has to start up the same job in a node where a copy of the data is found. A main difference compared to Googles MapReduce is that Hadoop has only one JobTracker handling all jobs and Googles has one master per job.

Hadoop 2.0 incorporates a component known as YARN (Yet Another Resource Scheduler). This component was essential to the Hadoop Frameworks due to performance issues experienced in clusters with over 4000 nodes. The bottleneck was identified due to the JobTrackers need to perform many roles. The introduction of YARN split the role of the JobTracker into a resource manager and an application manager. This solution of splitting up resulted to that each application gets its own application master for the lifecycle of the job. This is very similar to Googles original approach. The difference is that YARN can be used not only to run MapReduce jobs but also other jobs as well in parallel. [35]. Apache Spark is one of many applications powered by YARN that fits perfectly into the Hadoop ecosystem. [31]

As mentioned earlier reliability is offered by MapReduce frameworks by replicating data. To provide this reliability the idle nodes are also powered on to ensure data availability. This increases the idle nodes that are available and uses energy making the hadoop cluster ineffective. According to the authors, an idle node consumed upto 223 W when idle.
In an experiment conducted by the authors a 36 node hadoop cluster was configured to process MapReduce jobs. To compare the performance the same cluster was used to perform the same set of MapReduce job with 18 nodes being disabled. The authors mentioned that a 44% energy saving was noticed when 18 nodes are disabled [27].

2.10 Related Work in improving MapReduce

Climate change has impacted the earth with devastating effect already. Researchers and scientists have predicted the value of BigData and its increasing usage in enterprises in order to increase operational efficiency. The MapReduce programming paradigm, is used mostly to process BigData and is widely adopted and customized by many enterprises. One very well known implementation is Apache Hadoop. Over the years researchers have experimented in methods to both increase efficiency and reduce power consumption of a MapReduce cycle.

Green Hadoop

A project aimed at using the most of renewable energy is GreenHadoop [20]. It is implemented as an external wrapper for Apache Hadoop. The main contributions to this project is a job scheduler that schedules jobs based on 2 criteria, the availability of solar energy or the price of non-renewable energy. The job scheduler first predicts the availability of solar energy by watching weather forecasts. By using this data the jobs are scheduled according to the priority, with high and very high processed as soon as possible. A challenge faced as described by the authors is the unpredictability of resources for each MapReduce job. If the job is not completed as predicted, the system completes it when the price of brown energy is the cheapest. This project has reported to increase green energy consumption by 31% and decrease brown energy cost by 39%.

BEEMR

Another project that aims to reduce energy efficiency is BEEMR (Berkely Energy Efficient MapReduce) [14]. This research is based on the fact that Hadoop has nodes with unnescessary uptime. The goal of this project is increase the machine usage by processing more jobs per unit energy [14]. Challenges faced in this project as described by the authors is the unpredictability of workload. The idea behind this architecture is based on the type of job. Jobs that require an immediate attention, are classified as interactive. The other jobs are classified as batch or interruptive jobs, depending on the required time of completion. Interactive jobs have a independantly powered, always on cluster of machines used to process jobs requiring immediate attention. Batch and interruptible jobs are
powered by an independent cluster, that is not always powered on. The system uses a task queue to fill up the queue. At regular intervals the system is sent to a high power state so that batch jobs and interruptible jobs are executed. Batch jobs are attempted to be executed within the time-frame. If the job is not completed the system is powered on until the job is complete. Interruptible jobs have a lower priority than batch jobs and are interrupted at the end of every cycle, and continued at the next. BEEMR is an extension to the existing Hadoop implementation. Experiments conducted show a 40-50% saving in energy.

**DVFS in MapReduce**

Dynamic Voltage Frequency Scaling (DVFS) is used in modern processors which adapt the vary the clock speeds according to the load, thereby requiring lesser voltage and saving energy. The authors Thomas and Rong [40] claim that the MapReduce algorithm is affected by the processor frequency and the number of workers. Researching on the impact on the number of worker nodes, simulations are performed to find out the effect of varying the amount of nodes from 8 to 56. To test the impact of DVFS, 3 policies are implemented and experimented upon. Standard processor frequency throughout the whole cycle, Maximum processor during map and reduce cycles, A varying speed that allows 5% performance loss. By varying the amount of workers, test results showed that some scenarios did not show an ideal increase in performance. At its best some scenarios showed that with increased workers a speedup of 12 times was achieved with and energy consumption decreased by 30%. By varying the processor frequency, the best results were obtained by the last policy allowing a 5% performance loss. An increase in 23% efficiency was observed.

### 2.11 Tools used in implementing the project

This section will explore the tools that are used in the implementation of the task tracker.

#### 2.11.1 Docker

Docker is a CaaS (Container as a Service) which is used to compile an application with the necessary libraries as well. This property of Docker enables anyone to run the software independent of the base kernel that is running. Docker can be used to create containers. There are a number of base images found in Docker, which can be used as a foundation for the container. In the container, both necessary packages and libraries required are added as layers. The container can be built in stages as a live build or can be automated.
2.11. TOOLS USED IN IMPLEMENTING THE PROJECT

Docker containers and virtual machines seem to have the same purpose of usage, to launch applications without harming the underlying physical machine. Container technology differs from virtual machines in 2 significant ways. The hypervisor layer in virtualization is replaced by the Docker container engine. Docker containers have the advantage of packing only the very basic and necessary packages bundled with the application to reduce the size of the image compared to virtual machines which install a full copy of the operating system when running. This is observed when launching a Docker container that is pre-built on the system, which uses much less time compared to a virtual machine. The reduced size of the image makes it possible to run more containers compared to virtual machines on a server.

Images built to boot up containers are known as Docker images. Docker images are created by the help of Docker files. Docker files are lines of code which specify the machine what should happen in these layers and build these containers. This feature is highly useful when doing automated builds, and gives rise to minimal or no error if a need to repeat the build arises. An example of a Docker file is shown below. By creating such files, the software developed can be run by any client as long as Docker is installed on it.

```bash
1 FROM ubuntu:16.04
2 MAINTAINER gheeth
3 RUN apt-get update
4 EXPOSE 80:80
```

2.11.2 Python scripting

Python is a high level programming language used generally for scripting and web-programming. Being a high level language, Python features rich libraries which assist in solving larger problems in comparatively fewer lines of code. For example, the statistics library in Python makes it possible to calculate popular functions such as the mean, median and range with just one line of code.

2.11.3 JSON for communication

JSON stands for Java Script Object Notation. It is a lightweight, human readable messaging protocol for communicating between machines. JSON is relatively easy to create and parse since it only requires a key-value type of format. This format is similar to the object storage format dictionary in Python. A small snippet of a JSON message is shown below. The keys are required to be unique as to avoid duplicating. Python provides a library to parse and process JSON messages.
2.12 Summary

The background chapter introduced many technologies and concepts. The list below shows the main learning points:

- Virtualization is a method to abstract hardware.

- Cloud computing uses virtualization to deliver services over the Internet.

- Green Computing refers to methods of energy efficient computing. Live migration is a widely known technique in green computing.

- Unikernel OS’s are very small in size and use no energy when idle.

- Big data refers to data sets that are large in volume. Bigdata is characterised by its volume, variety and velocity required to process. It is used in many fields such as medical, sciences and astronomy.

- Grid computing is a collection of computer nodes performing a job related to a single task. The computer nodes need not necessarily be tightly coupled. Middleware is used to communicate between the computer nodes.

- MapReduce is a programming paradigm where the problem is solved in a divide-and-conquer scheme. It is used widely to process big data sets.

- Apache Hadoop is an open-source implementation of MapReduce. Hadoop nodes assume a local cluster. For the purpose of data availability, all idle nodes are powered on.

- Related work in improving MapReduce is based on intelligent scheduling algorithms that aim to conserve energy. These are implemented as a wrapper to Hadoop.
Chapter 3

Approach

This chapter will explain and justify the methodology used to realize the aims of this master thesis.

3.1 Objectives

As mentioned in the introduction chapter, a MapReduce like framework is to be designed first to explore a distributed framework. The proposed MapReduce framework will contain a 4 main types of components. The corpus splitter, task tracker, map worker, reduce worker. The functions of these components will be discussed in the design chapter, along with how these components contribute in a greener framework.

The focus of this master thesis is on developing the task tracker. By looking at the problem statement, key-words can be identified depicting the desired properties. "Design and develop a task tracker for dispersed and dynamic MapReduce-like frameworks in order to investigate the feasibility of follow-the-sun enabled big data clusters". As the problem statement specifies the main goals here are

- Designing and developing a task tracker
- Task tracker should perform similar to a master node in a MapReduce framework
- Task tracker should facilitate for communication between dispersed worker nodes

To achieve the goals above, a plan has to be put in place regarding how the solution is developed. This chapter describes the four main tasks and its subtasks.

1. Designing the model
2. Implementing the model
3. Testing the model
4. Measurements and Analysis

The design phase of the project will put together the knowledge gathered from the background chapter into a model that satisfies the goals mentioned above. It is also necessary to verify where experiments will be performed to gather data for analysis.

The implementation phase of this project will then implement the task tracker in code. By adhering to the model developed in the design phase, the necessary features will be implemented in the tool of choice justified. An interface for experiments should also be implemented to gather data.

The testing phase in this project will then simulate the task tracker by providing it with various scenarios. These various scenarios will attempt to give output via the interface that is implemented. Lastly, an attempt will be made to simulate a job to investigate the difference between running a MapReduce job in local cluster and a dispersed cluster.

The analysis phase can then be used to make sense out of the data that is recorded. This data can provide insight on how the model fares and hint to possible improvements on the existing system. Since this is the first step of a novel-framework, this can also provide details on whether this project is worth building upon.

3.2 Design phase

This phase will describe the model that is developed in order to achieve the goals mentioned above. At first, the overview of the whole framework should be modelled. All clarifications regarding the different components in the dispersed MapReduce framework will be discussed. As mentioned in the previous section, this thesis only focuses on implementing the task tracker. To do this, the necessary features have to be identified. The goals have to be broken into smaller goals and the respected outcome for each goal has to be described. When these minor goals are small in size, they can be described by the help of pseudocode, text and illustrations to specify the task that is being done. Each of these smaller tasks should have a purpose in contributing to the main goal. The design will also be decoupled as it helps in locating the error in case of debugging or upgrading with newer features. The word "investigate" in the problem statement, naturally means data has to be collected. The design phase will also include a rough overview on the methods of testing the system, which can be used to gather useful data. Due to the purposes of testing, a reference worker and a client has to also be developed.

As the problem statement describes the developed framework will be MapReduce like. This means that the design has to facilitate computations
in a MapReduce like format. Knowledge gathered in the background chapter has to be put use on how MapReduce processes data. MapReduce also possesses characteristics such as reliability and fault tolerance which should also be attempted to replicate. The idea of being dispersed and dynamic is something that is not seen before in a MapReduce framework. Therefore the principles used in grid computing have to be put to use to understand on how it can be superimposed in a dispersed MapReduce like framework.

To summarize, the chapter should describe in detail:

- The proposed framework
- Keywords and terms
- The scope of this thesis
- The objectives of the project
- The function design with the aid of pseudocode
- Deriving performance metrics

### 3.3 Implementation phase

The implementation phase will proceed according to the model that is developed above. The aim of the thesis is to build a working prototype of the task tracker. There are several tasks to be completed before the prototype is implemented:

- Identify the tools and prerequisites to implement the prototype
- Identify the environment that the prototype shall be developed in
- How to develop the prototype
- What should be in place before experiments are performed
- A client for simple user interaction
- Proof-of-concept instrumentation for crude worker management

To develop the prototype there are two aspects regarding the tool for development. The first, regarding on what tool shall be used to develop the application. The second, being on how it shall be implemented. The application shall be developed in Python as a script. The reason for using Python is that it contains rich libraries which effectively compute required functions without the need of manually writing it. The development will be tested on a Linux environment therefore, bash scripts are the natural
choice of selection for deploying the framework both in small and large scale.

The development of this prototype will be divided into two parts. The first pass shall aim to implement a prototype that performs the functions required without having to stress on optimization. Here there has to be implemented a protocol for communication. A reference worker and reducer must be developed for the purpose of testing. A client must also be developed to query the task tracker.

Once the first pass is achieved, the prototype will be deployed on the cloud running on Oslo and Akershus University College. Due to the cloud providing more computing power, the implementation can be scaled up. The second pass of the implementation will focus on fine tuning and optimizing the prototype. The final version with a working prototype with the features mentioned above, will be used in the experimentation. As mentioned in the design phase, there will also be methods implemented to collect statistics on the job that is done. As a goal of this thesis is to investigate the feasibility of a dispersed MapReduce framework, there has to be collected enough data during the experiments. Data collected here is planned to be mostly timestamps and application logs. This has to be recorded during the lifecycle of a job. The implemented files will support being run from a Docker client, and therefore Docker files must be created to deploy the server and the clients. As mentioned in the background chapter, enterprises gathered more data with the hope of increasing operational efficiency. The same thought will be put to use here in collecting data on the hope of finding methods of optimizing the framework.

A typical MapReduce scenario features more than a 1000 workers. This should be replicated with all available resources on the ALTO cloud. Setting up a large scale implementation could take up a lot of time and therefore a plan on launching the framework must be mentioned. This could be one of the most repetitive tasks when experimenting. Depending on the magnitude of the experiments, there can be a lot of data that is created. This means that scripts should be developed to ease the analysis phase of the project.

### 3.4 Experiment phase

This phase is predicted to be the most interesting phase due to the nature of the project. Since the idea of a dispersed and dynamic MapReduce like framework is relatively new, there are many possible interesting paths to be discovered by experiments. The experiments therefore first have to test basic functionality. This has to be tested so that it works independent of the scale that it is deployed in. The basic functionality tests will aim to mimic a MapReduce behaviour. The whole loop of the of submitting a job to the job being sent to processing and displaying the results should be
3.5 ANALYSIS PHASE

Reliability is a feature that is provided by MapReduce traditionally, therefore experiments should be made to test for reliability.

Once the basic functionality is tested, experiments that are intended to understand the behaviour of the framework will be performed. These experiments will via influenced situations, test the framework on how performance is affected. Such examples of tests are influencing a worker to use more time than necessary per job or even deleting a job after it has been sent. Metrics should also be collected regarding the duration of the completion of a job. Lastly, to test the main purpose of this thesis, a comparison between a local cluster and a distributed cluster will be made to prematurely investigate into the feasibility of a dispersed framework.

The intention of performing these experiments is to gather data to understand how the framework performs under different conditions for the purpose of further improving the system. If time permits these experiments will be repeated to test for consistency.

The summary of the experiments needed to be performed are

• Test basic functionality.
• Perform influenced tests on the framework.
• Compare the MapReduce framework in both a local and a dispersed cluster.

The above mentioned experiments are a rough sketch on what has to be touched upon. This is due to the functionality implemented being unknown, in the task tracker. The detailed explanation of each experiment can be found on the Experiment chapter.

3.5 Analysis phase

The analysis phase of the thesis will be one of the most crucial parts of this thesis as it reveals how the framework performs. Data which is collected during the experiments will take the form of raw data. Raw data described here will be mostly in the form of timestamps and application logs. The magnitude of the data collected can be predicted to increase exponentially depending on the size of the framework. This meaning the higher the number of experiments or worker nodes, the larger the data that is required to analyse. This data will be converted into graphs and tables showing a more concise result of the experiment. A secondary goal of representing this data in graphs is to help the reader in understanding on how the framework processes a job.

These graphs and tables will show a number of performance metrics, such as time used per job and failure percentage. By looking at these graphs, both positive aspects and negative aspects can be found. Examples
of what to look for are anomalies in processing, in how an error in processing affects the other jobs. Further statistical operations performed on the data will be decided when the actual prototype is developed. With the help of the graphs and tables created, the data can then be used to derive conclusions on both the advantages and disadvantages of the framework. Since a major difference between traditional and the proposed framework is the property of being dispersed, therefore an analysis on this based on the experiment performed would give insight on how useful this thesis is.

The goal of the analysis is to answer the question in the problem statement regarding the feasibility of a dispersed framework. The summary of the tasks performed in the analysis section will be

- Convert raw data into graphs and tables
- Perform statistical calculations on the data collected such as average job completion time
- Draw conclusions on behaviour, advantages and disadvantages of the framework

3.6 Expected Project Deliverables

By designing and implementing the above mentioned, the expected deliverables of this thesis are

- A protocol developed to facilitate the communication between the different components, for the sake of reliable communication
- A prototype of a task tracker that handles job related tasks such as receiving and sending job requests between clients and worker nodes
- A thorough analysis of how the task tracker performs backed up by data collected during experiments
- Comparison of a local and a distributed cluster based on the prototype created

The tasks regarding design and implementation, seem to have a moderate grade of difficulty seen from the text above. The design tasks are critical to be implemented firmly. This is to both simplify the implementation phase. Performing experiments and simulating a dispersed scenario can be difficult at first to setup. Once this is setup up, comparing a local and dispersed cluster performance becomes interesting.

The risk in a such approach is very low since the development is not based on any existing technologies. Therefore no time is wasted on learning them. Python is a well documented language and therefore support can always be found online and on user manuals.
3.7 Summary

<table>
<thead>
<tr>
<th>Phase</th>
<th>Goals</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Describe Proposed framework</td>
<td>Diagrams and Text</td>
</tr>
<tr>
<td></td>
<td>Describe functions performed</td>
<td>Text and pseudocode</td>
</tr>
<tr>
<td></td>
<td>Derive performance metrics</td>
<td>Mathematical equations and formal notation</td>
</tr>
<tr>
<td></td>
<td><strong>Implementation</strong></td>
<td><strong>Methods</strong></td>
</tr>
<tr>
<td></td>
<td>Task tracker prototype</td>
<td>Code snippets</td>
</tr>
<tr>
<td></td>
<td>Worker, client and reducer prototype</td>
<td>Code Snippets</td>
</tr>
<tr>
<td></td>
<td>Define communication protocol</td>
<td>JSON message examples</td>
</tr>
<tr>
<td></td>
<td>Create deploy scripts</td>
<td>Bash code snippets</td>
</tr>
<tr>
<td></td>
<td>Test basic functionality</td>
<td>Use scripts and output logs</td>
</tr>
<tr>
<td></td>
<td>Perform influenced tests</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Compare dispersed vs local cluster</td>
<td></td>
</tr>
<tr>
<td>Analysis</td>
<td>Convert logs to graphs</td>
<td>Scripts to analyse and draw graphs</td>
</tr>
<tr>
<td></td>
<td>Perform statistical calculations</td>
<td>Scripts to calculate</td>
</tr>
</tbody>
</table>

Table 3.1: Summary of the approach
Chapter 4

Results I: Design

The purpose of the chapter is to present the design of the task tracker, worker, reducer and client. As mentioned in the approach chapter the goal by the end of this chapter to have a model of the prototype with all the designed functions in place. The tasks that are planned to be achieved are:

- Define the communication protocol used
- Define the functions implemented by the task tracker
- Define metrics that will function as useful statistics
- Describe the function in detail using both text and pseudocode
- Design a basic client and worker

Designing the model requires a good understanding of the technologies mentioned in the background chapter. It is also important to consider research done on MapReduce and green computing previously to incorporate knowledge that has been proven to work.

4.1 The Big Picture and where this project fits in

As explained in the background chapter there has been work done on making MapReduce more efficient. Research done shows a lack of investigation into distributed solutions, but rather uses smart scheduling to reduce energy. By combining one of the most known green computing techniques live migration and a well known big data framework, a design for a novel framework is thought upon.

The main idea for this type of design can be derived from Grid Computing. As mentioned in the background chapter, the grid uses computers that are not necessarily within the same network, which simulates a dispersed environment. The grid also takes advantage of all the
computers by performing parallel computing. The middleware that is used in grid computing will be attempted to replicate by using a communication protocol. This idea will be put to use in a MapReduce framework.

This section proceeds by first defining the keywords and terms used in describing the framework. The framework and how it is intended to be green is described. Clarifications regarding the scope of this thesis is also defined.

### 4.1.1 Keywords and Terms

This section will define the keywords and terms used to describe the design

**Corpus**

A corpus is regarded as a Big Data set. Being a big data set, it naturally posses the characteristics of Big data (Volume, Variety and Velocity). An example is the wikipedia database. The wikipedia database contains around 5 million entries.

**Shard**

A shard can be regarded as a piece of a corpus of a or big data set. This shard is smaller in size so that processing times are relatively lowered. A corpus is generally split into many distinct shards. Considering the example from before if wikipedia is a corpus then a shard would contain all the entries that begin with the letter A. Mathematically a shard can be represented as

\[ shard \subset corpus \] (4.1)

The above equation (4.1) shows that a shard is a subset of a corpus. This means that a shard contains a part of a corpus.

\[ shard_1 \cap shard_2 = \emptyset \] (4.2)

The above equation (4.2) shows that 2 shards contain a unique piece of a corpus. If 2 shards were to be compared with each other, there would be no overlapping information.

\[ shard_1 \cup shard_2 \cup shard_3 = corpus \] (4.3)

The equation (4.3) shows that if a corpus is said to contain only 3 shards, combining all shards would give a big data set.
Worker

Workers are regarded as the computing units of the framework. This is due to the workers performing a fraction of a larger calculation. In a typical MapReduce framework there are from tens to thousands of workers, which take part in performing bits of the calculation. Each worker shall perform its computation on a distinct shard.

Corpus Splitter

The task of a corpus splitter is to split the corpus. By splitting the corpus, many shards are created in order for the workers to process. The size of the shard can be varied according to the total size of the corpus and the amount of workers available.

Query

Queries are the actual processing required to be done on a corpus. An example of a query is finding out the occurrences of a specific word in the wikipedia corpus.

Jobs

A job is submitted by a user. All jobs will include a query. In addition to a query the jobs will also include a user id and corpus id.

Request

A request is defined as a user’s wish to acquire information. An example for a request is to view all jobs pertaining to the user or view the status of a job.

User

A user is defined as someone who submits a job via a client. The client mentioned here can be modelled as providing a simpler form for communication between the user and the MapReduce framework.

4.1.2 The framework

As seen from the illustration 4.1, there will be 5 main elements present.

1. Task Tracker
Figure 4.1: 1. The corpus is split into shards. 2. These shards are included in workers and deployed into the cloud. 3. The workers connect to the task tracker. 4. The user sends a job to the task tracker. 5. The task tracker sends a reduce job to a reduce worker.
4.1. THE BIG PICTURE AND WHERE THIS PROJECT FITS IN

2. Worker nodes
3. Reduce nodes
4. Corpus splitter
5. Client

The Task Tracker will have the main functionality to keep track of both jobs and worker nodes. This can be also regarded as the central node or hub of communication between the client and the worker nodes. If the task tracker is intended as the master, the worker nodes are the slaves.

Currently these five components are predicted to perform the main tasks required by the whole system. The intended method of solving a problem is as follows.

1. The corpus splitter will split up the data in shards and impose the function of the worker with it. This means that one worker has a distinct shard. The details for communication with the task tracker will also be imposed while creating the worker.

2. These workers will be then released into the cloud.

3. Once they are powered up, they attempt to connect to the task tracker. The locations of these workers are recorded in the task tracker.

4. Once a client submits a job to the task tracker, the task tracker will place the job in some representation of a queue. An example of a map function is to count the number of articles in wikipedia that contain the word "computing".

5. The worker connects to the task tracker based on an interval and reads from this queue and execute the job. After the job is executed the results will be sent back to the task tracker.

6. If a reduce function is specified, the task tracker will send the result to a cluster of reduce nodes. These will then reduce the mapped results according to the reduce function. An example of a reduce function is to sum up the occurences of the word "computing" from all the worker nodes. The reduced results are then sent back to the task tracker.

The structure of this framework is similar to how a beehive functions. The task tracker is regarded as the queenbee. Both the map and reduce worker nodes are regarded as the worker bees. The queen bee uses the worker bees for all needs such as food. The workers sign in based on intervals to check if the task tracker has a job to be processed. Due to this similarity, the framework is called the beehive.
4.1.3 Introducing green behaviour into the framework

Based on the research mentioned in the background chapter on making MapReduce effective, the authors [14] had suggested that MapReduce consumes energy by due to powered idle worker nodes. This framework will operate on a pull basis where the workers attempt to pull the list of jobs rather than the jobs being pushed to the workers. The workers will connect to the task tracker based on an interval and if no tasks are to be performed the worker can go into a low power state or even be shut down. For example a worker can connect to the task tracker every 2 minutes, and if no task is to be processed the worker shutdown for 2 minutes and attempts to reconnect. This would hopefully save energy if no tasks are to be processed. The background chapter mentioned the advantage of using an unikernal OS such as IncludeOS. If the worker nodes use IncludeOS as their base operating system, there is no energy consumed in between sleep intervals. This will not be implemented in this thesis, but can be done in the future.

As mentioned above, one of the main aims and differences between this framework and the existing solutions is the ability to find greener clouds. The workers in this situation act as agents which communicate with the task tracker. To achieve the property of migration into greener clouds, there

Figure 4.2: One of the ambitions of the framework is that the workers can migrate from data centers using brown energy to green energy.
has to be either an independent component or intelligence programmed into each worker. The migration will aim to then find out about the current source of energy in a datacenter. If there is a considerable use of brown energy, the decision should be made to migrate to a datacenter running on green energy. This is seen in the illustration 4.2. The illustration shows data centers running on brown and green energy. The ratio of brown energy to green energy present in a datacenter is depicted here by the colors. The process that takes place here is that worker nodes migrate from data centers which are running mostly on brown energy to data centers running on green energy.

### 4.1.4 Clarifications regarding the different components

The purpose of the figure 4.3 above is to clarify the differences between the 2 distinct areas of management. The first area revolves around managing jobs which will be the goal of the task tracker. This will have the responsibility of managing tasks around jobs such as job receiving from client, submission of to workers, control on how far the job is done. To support this, is the second area of management that revolves around worker management. The worker management will have the job to release workers into the cloud, manage the status of the workers. There has to be some sort of communication in the form of feedback between the two areas to maintain a good quality of service.
4.2 Objectives in detail

As clarified previously, the scope of the task tracker is developed with the intention of managing the jobs. The functions listed below are necessary for the task tracker to fulfill its use in the framework. These functions are derived from the description of how the proposed framework will perform.

1. **Receiving and storing worker information**: All workers will connect to the task tracker and the position of these will be recorded. This is to know where to send the job request.

2. **Receiving and storing jobs from client**: Jobs from clients will be stored in the task tracker. These jobs are then set on a task queue for each of the workers. The workers will access their queue and process the jobs.

3. **Providing status of jobs for the client**: The client should be able to see the status of the jobs submitted by the client. The status should show on if the job is queued or how far it has come into processing.

4. **Providing security for the clients**: Security for the client here means that only the client that specified the job should be able to modify or delete it.

5. **Create a protocol for communication between the worker, client and task tracker**: This communication protocol has to be designed to communicate with the 3 components. It is necessary that the size of this message is kept to a bare minimum to reduce processing and sending times.

6. **Providing statistics regarding job scheduling**: This type of statistics are similar to performance metrics. Metrics provided can be on the average completion time of a job and the slowest processing worker. This statistics should be provided in the hope of increasing operational efficiency.

7. **Handling special cases**: Define the method of action to be taken under special circumstances such as losing a worker permanently or an user deleting a job.

By realizing the objectives mentioned above, the basic functions of a dispersed task tracker are implemented.

4.3 Task Tracker Design

As seen from the above section, there are 2 types of types of main tasks that are to be developed. The first is known as the communication protocol that is required to be developed to pass the different types of messages between
the components. The second is the design of the function on how the task tracker will perform. As mentioned in the approach chapter the functions will be described by the help of text and pseudocode.

The functions require a global storage solution that will store data. The different types of data that will be stored is a list of corpuses, users, jobs and workers. In addition to this there has to be a status list from which the workers will receive their respective job list. The task tracker will be designed to handle multiple corpuses, workers, jobs and users. The type of lists chosen is decisive in the ease of programming the function.

1. **Corpus list** :- The list will contain the list of corpuses tracked by the task tracker. This list can also contain the jobs pertaining to the corpus.

2. **User list** :- The list will contain the list of users, and the list of jobs submitted by each user.

3. **Job list** :- The list will contain all the jobs submitted by all users. It will act as a global list.

4. **Worker list** :- The list will contain all the workers that have signed up with the task tracker.

5. **Worker Queue list** :- This list will contain the queue for each independent shard list sorted by corpus.

Many such lists mentioned above can be added later depending on the requirement of the task tracker, and its functions.

### 4.3.1 Communication protocol

To design a protocol, it is necessary to define the content of the message. By defining what the message will contain, unnecessary information is omitted from the message. This results in faster processing times and reduces bottlenecks when handling a constant stream of messages. JSON objects can be used to build and transmit messages. As mentioned in the background chapter, JSON objects are both lightweight and human readable. A JSON object can be described as having multiple key-value pairs or known as fields. An advantage of using JSON objects is the ease of modifying an existing message, which can be used in adding or removing fields when implementing newer features in the future.

Secondly, the communication between the different entities is identified. Since this thesis focuses on developing the task tracker, communication between the entities directly coupled with the task tracker is only regarded. As seen from the figure 4.1, the corpus splitter does not communicate to the task tracker. Therefore communication between the task tracker-client, task tracker-worker and task tracker-reducer is designed.
Designing the messages require a broad understanding of what tasks are required to be performed by the different entities. The objectives mentioned above describe above give hints on what type of tasks are expected to be handled by the task tracker.

- Send query :- client to task tracker
- Send task request :- task tracker to worker
- Send results :- task tracker to worker
- Initiate communication :- worker to task tracker
- Send job request :- client to task tracker
- Reply or confirm requests :- task tracker to client

From the list above the different type of messages passed between the entities is seen. Messages passed between entities have to be uniquely identified in order to prevent errors in communication. Therefore will all messages contain a type field which indicates what type of message is being sent. An example of a query sent from the client to the task tracker is shown below.

```
{
  "type" : "query",
  "request" : "Job status of job 1"
}
```

Listing 2: JSON example for a query request

In the snippet 2, there are two fields of data. The first as mentioned is the type field which specifies the type of the message. The second field is an example of what a request might look like. The content of each message will be discussed in the next section as the messages must be customized to each function.

### 4.3.2 Function design

The tasks that the task tracker will perform are defined in the objectives above. Although these seem short when mentioned, there will be a lot of underlying processing done when each of these tasks are requested to be performed. To implement the task tracker a deep understanding of the underlying tasks to take a job from start to finish has to be analyzed. The list below presents a chronological ordered set of real world tasks that are to be expected to perform by the task tracker.

- A worker connects with the task tracker for the first time
A job is received by the task tracker and is processed for delivery to the workers

A worker connects with the task tracker and receives a list of jobs

A worker is done with its job and sends the results back to the task tracker

A client sends a request on the status of the jobs sent by the user

Task tracker prepares a reduce computation

The user requests the result for a job

The section proceeds by describing in text and pseudocode on how the tasks will be processed from start to finish. Since the description of the functions involve message passing, the function design will also discuss on what the message requirements for each type of task are.

**Task tracker receives a job from a client**

This task is one of the main and common tasks processed by the task tracker. Here the input is a job from a client and the desired goal is to place the job in the queue of each worker belonging to the corpus. The job will be sent by the client in the form of a JSON object. Due to the task tracker being designed to handle multiple corpuses, the message sent by the client has to specify which corpus the job has to processed on. The client can also be used by many users, therefore some form of user identification has to be provided as well. In case a reduce computation is required, the user should be able to input a reduce script. The message received will then be parsed by the tracker. In order to verify the content of the message, tests have to be performed to find out if corpus exists in the system. Feedback must be sent if the corpus is not found. If the corpus is found, a unique job ID must be returned. This will be the unique identifier of the job and will be used for future reference, when requesting results or status. The job details with the query must be then placed on each worker's queue corresponding to the corpus. The pseudocode below describes how this situation will be handled.

As seen from the textual description and pseudocode, interaction happens only between the client and the task tracker. The type of messages that are being passed are:

- Job submission from client to task tracker
  - corpus name or id
  - query
  - reduce function (optional)
CHAPTER 4. RESULTS I: DESIGN

```python
if (corpus from message) exists then
    if (user from message) not exists then
        append user in user_list
        send_to_client(message with job_id)
        append job in job_list
        append job in users_job_list
        for each worker_queue in corpus_worker_list do
            append job
    else
        send_to_client(message with error)
```

Listing 3: Task tracker processing job

- Job ID reply from task tracker to client
  - Job ID
- Error message from task tracker to client
  - Content of error message

Initial worker connection to task tracker

This task is one of the very first tasks that has to take place, before job request are sent. If no workers are present then no corpus is registered on the task tracker. Therefore the worker has to connect with the task tracker and provide information. This task possesses similarity to a standard human interaction, starting with a introduction of identification. The message will then consist of the corpus id, shard id and location. This task tracker then has to establish if the corpus and shard id exists. If they do not exist then the worker is registered and associating empty job queues are created. If they exist then the task tracker will consider it as a duplicate worker. The purpose of the location field in the message is crucial to understand if the worker has migrated or if a duplicate worker was being booted up. In case a worker does not respond from its old location for a period of time, it is safe to assume the worker has been migrated or lost. Migration can be found out, if the worker with the same corpus and shard disappears from one location but begins responding from another location. This means that the total amount of worker with the same corpus and shard will be unchanged. The task tracker shall send a message of acknowledgement to the worker, specifying the status of the operation. The message that is sent back can also include a list of pending jobs. A new worker would receive an empty list, but a duplicate worker can receive a list of jobs. The method of receiving jobs is discussed in the next section below.

The pseudocode in listing 4 above describes the algorithm required when determining the status of a worker. As seen from the textual description
4.3. TASK TRACKER DESIGN

Listing 4: Worker Registration Process

above there is interaction between the worker and the task tracker. The messages passed are therefore:

• Worker identification from worker to task tracker
  – Location as an ip address
  – Corpus id or name
  – Shard id

• Acknowledgement from task tracker to worker
  – Status of signup
  – Job list (If successful signup)

Worker checks in to receive a job list

This task is expected to take place frequently depending on the interval used. The objective of this task is to provide the workers with a list of jobs that have not been started. The necessity of this task is due to the jobs being pulled by the workers and not pushed to them. As mentioned earlier, this feature enables green computing. The intended method of functioning starts when an established worker reconnects or checks in to the task tracker. Compared to the previous task, this task is similar in human interaction to greeting a friend. Once the worker is identified, the task tracker has to identify the jobs that were placed in queue, when storing the job. If the job has not been issued to any other workers before, the job status is modified to “issued”. As mentioned before, the task tracker
will also be designed to handle duplicate workers. The use of this is to
process jobs in parallel amongst the workers. The task tracker should then
be capable of securing the fact that duplicate workers do not receive jobs
that have already been delivered to another worker. The list of jobs, has to
be then encapsuled in a JSON object and sent to the worker. If no jobs are
queued, an empty list will be returned.

```python
if worker not in worker_list then
    add worker in worker_list
else
    for job in corpus_worker_list
        if job is not issued then
            append job to job_list
            set job as issued
    capsule job_list in JSON object
    send(JSON object) to client
```

Listing 5: Worker Receives job list

The above psedocode in listing 5 represents how the job list is created
to deliver to the worker. As similar to the previous task, communication
happens only between the worker and the task tracker. The communication
between the two entities here is in some ways similar to the task of signing
up a worker.

- Check in from worker to task tracker
  - Corpus id
  - Shard id
  - Location

- Job list from task tracker to worker
  - Job list

Worker returns back with results of a job

This task is very similar to the previous task. The worker will connect to
the task tracker by identifying itself. The task tracker then will verify if the
worker is a sign up or a check in. Once verified, the worker will contain a
list of jobs it has processed. If the result list is empty, the worker is either
working on a job or not received a job. If the result list is not empty, the
task tracker will then parse the result list and store the results in the task
trackers local list. The task tracker can also then check if all the workers for
all shards have submitted the results. If yes, the task tracker can mark the
status of the job as either finished or mapping finished. In case the user has
submitted a reduce function, the status will be stored as mapping finished.
4.3. TASK TRACKER DESIGN

Listing 6: Worker returns with result list

This task is seen to have a one way communication from the worker to the task tracker. The message sent contains:

- Corpus id
- Shard id
- Result list

Check status of jobs submitted by client

The possibility of a user submitting more than a job should be considered. In this case an overview of the jobs submitted by the user can be helpful in showing the status of all jobs. This method should be secure as to only show the jobs submitted by the user specified. Therefore the user id should be submitted by the client. Once the user id is verified by the task tracker, the task tracker will encapsulate the list of jobs and its status in a JSON object. This object will then be sent to the client.

Listing 7: Status request of jobs from user

This task has interaction between the task tracker and the client. The messages passed are:

- Status request from client to task tracker
  - User Id
- Job statuses from task tracker to client
- Job ID
- Job status

**Client sends a request on a job result**

The previous tasks focused on submitting and processing the job. Naturally, when a job is submitted the results are requested by the client. Due to the uncertainty of the time used to process the job, the client should be able to send a query to find out about the status of the job. The client then has to specify the user id and the job id. Upon receiving the message, the task tracker has to verify if the job is submitted by the client. There are 2 possibilities of responses that can be given by the task tracker. For a job that is either "queued" or "issued", the status can be returned as done in the task above. If the status of the job is completed, the results have to be sent to the client. This is described in the pseudocode in listing 8 below

```
1 if user_id in message is found then
2    if job_id in message is found in user_job_list then
3       if job_status is not finished then
4          send_to_client(job_status)
5       else
6          send_to_client(job_result)
```

*Listing 8: Request job status*

As seen from the text above, there is communication between the client and the task tracker. The type of messages passed are:

- Job status request from client to task tracker
  - User ID
  - Job ID

- Job status from task tracker to client
  - Job status

- Job result from task tracker to client
  - Job result

**Task tracker prepares a reduce computation**

This task will take place if the user has specified a reduce function, when submitting the job. Technically, this stage will take place right after all mapping computations are complete. The jobs with a reduce function will be placed on a reduce queue in a first in, first out manner. In similarity to
all worker nodes, the reduce nodes will also receive a job list during the sign in period. The reduce function specified by the user along with the results of the mapping computations will be encapsulated in a JSON object and sent to a reduce worker node. This node will run the reduce function on the mapped results. An example of a reduce function is sum up all occurrences of a specific character or word.

```plaintext
1 if job_status is map_finished and reduce_function not null then
2     reduce_queue.append(job)
3 for each job in reduce_queue do
4     send_to_reduce_worker(job)
```

Listing 9: Prepare Reduce Computation

The communication that happens here is between the task tracker and the reduce node:

- Check in from reduce node to task tracker
  - Corpus id
  - Location

- Job list from task tracker to reduce node
  - Job list
  - Results from Mapping
  - Reduce Function

Handle failed workers

Both Google’s and Apache’s MapReduce provide reliability and fault tolerance. As mentioned in the background chapter, a traditional MapReduce structure pings every worker to check if it is alive. A similar method will be used here to check workers that are alive. One of the main differences between the proposed framework and the traditional MapReduce framework is that workers have the ability to return to a low power mode when idle, rather than them being powered on unnecessarily.

To keep track of last response, there will be implemented a time stamp, mentioning the last time the worker reported to the task tracker. An ideal situation then would be that the worker reports to the task tracker as specified by the interval. In case of no response from the worker for more than 4 times the interval specified, the worker can be assumed to be dead. The task tracker then has to restart the jobs processed by the worker to another worker with the same shard and corpus id. The worker will also be removed from the worker list to prevent jobs being sent to it.
4.4 Worker and Client Design

As mentioned earlier, due to testing purposes of the task tracker, a reference worker and a client has to be developed. The term minimal means that only functions that are necessary are developed. A command line client has to be designed and implemented as an interface has to be made to test the above designed functions. The section will proceed with the design of the worker and client.

4.4.1 Worker Design

Workers are regarded as the computing units of a MapReduce framework. They have the job of performing a map or a reduce function based on the given data set. The intended functionality of a worker as seen from the description above are:

1. Worker connecting and receiving a job list from the task tracker
2. Worker delivering a result list to the task tracker

Worker connecting and receiving a job list from the task tracker

Once released from the corpus splitter the worker will, connect to the task tracker for the initial signup. This, as mentioned above will be done by a JSON object. Communication between the worker and the task tracker will be then defined by the interval predefined. If a job list is not returned, the worker can go into a low power mode until the task tracker is contacted the next time. If a job list is returned, the worker will parse the job list and perform the map function for each job. The results will be then stored with the job details. Once the job is completed the worker will continue with the next job in the list. If the job list is completed before the reconnection interval, the worker can return to a low power mode. The process will be similar for the nodes performing the reduce function.

Worker delivering result list

Independent of how many jobs the worker has completed, the worker will communicate with the task tracker. Before initiating connection, the worker will also create a progression report. This report will mention which jobs are completed, in processing and are queued. The details along with the results of the completed jobs will be encapsulated in a JSON object and transmitted.
4.4.2 Client Design

Clients are regarded as a user interface to the MapReduce framework. The client has the job of communicating with the user and the task tracker. The client can perform tasks such as submit a job, request status of a job, list all jobs the user has submitted. For the purposes of testing and experimenting, methods for downloading statistics on how the task tracker performs should be also implemented. These tasks all involve parsing user input into a JSON object and therefore should be not so complicated. This JSON object will be then sent to the task tracker and a reply will be expected. The reply should then be pretty printed for the user.

4.5 A timeline of messages passed

The previous sections mentioned the functions that are to be implemented and the messages that are to be passed. The timeline 4.4, shows the order of messages that are to be passed. The situation depicted here is an interaction between one map worker, one reduce worker and one job with a reduce script.

Figure 4.4: A timeline of what messages are passed in order to understand how the framework will function

- The first and second pair of messages is between when the map worker and reduce worker signs up for the first time
• The third pair of messages is when a client sends a job to the task tracker.

• The fourth pair of messages is when the task tracker sends the job to the worker.

• The fifth pair of messages sent is when the client requests the status of the job.

• The sixth pair of messages is when a map worker returns with the result of a map function.

• The seventh pair of messages sent is when the reducer signs in and receives a reduce job.

• The eighth pair of messages is when the reducer worker replies the result of the reduce computation.

• The final pair of messages is when the client requests the results of the job from the task tracker.

4.6 The Statistical aspect of the task tracker (Performance metrics)

As mentioned in the planning phase, the project will also have methods that gather statistical data on how the task tracker performs. The process of gathering data and analysing it is closely related to what is planned for the analysis phase of this project. The statistical data will be both used to understand how the task tracker performs through experiments and to provide performance metrics for the components responsible for worker management. Due to the latter reason, the methods used to calculate statistics must be implemented in the task tracker as functions are designed.

A statistical calculation is performed to explore and understand the behaviour of the task tracker. Statistics provide a number of functions that can be used. Therefore only the functions that describe performance will be chosen. Interesting patterns of behaviour are: average job completion time per corpus, how a slow worker affects rate of delay in job completion and administrative overhead per job. These performance metrics here are based on time. Therefore, timestamps will be collected from every stage in a job cycle. Examples of these are time created, issued and completed. These timestamps will then be put into mathematical equations to give performance metrics.

The diagram 4.5 above depicts a predicted job scenario. Its is a graph of the worker vs time. The graph shows a job from the time it was created to the time finished. The bar lines in the middle depict the worker start time and finish time. It is when the last worker is finished the job is predicted...
4.6. THE STATISTICAL ASPECT OF THE TASK TRACKER (PERFORMANCE METRICS)

Figure 4.5: A predicted job scenario with 4 workers. The graph shows the different timestamps collected to be completed. The short forms shown in the diagram are timestamps which are described below

- $T_{sl}$ - Total time from time created to time finished
- $T_t$ - Total time from time issued to time finished
- $T_{adm}$ - Administrative overhead from time created to time issued to first worker
- $T_{W_{max}}$ - The worker that completed in the longest time
- $T_{W_{min}}$ - The worker that completed in the shortest time

By applying these gathered time values in statistical functions, performance metrics can be derived. The performance metrics that are derived should show how workers per job and how job completion times progress. This requires understanding of how the workers are performing. Statistically the mean can be affected severely by a few workers that are outliers. Therefore the $95^{th}$ percentile is chosen to find out the processing time of $95\%$ of the workers.

$T_{W_{95}}$ - The time required by $95\%$ of the workers to complete the job.

This value has to be normalized in order to be used in comparison. The normalized value can be known as

$$D_{95}^T = T_t - T_{W_{95}}$$
To understand the effectiveness of job management in how synced each worker is, a calculation can be made. If all workers are perfectly synced the total time used for the job would be the time that is used by the slowest worker. Therefore the calculation can be formulated as

\[ D_{\text{max}}^T = T_t - T_{\text{max}}^W \]

4.6.1 Making sense of all the metrics

To make use of the collected values, the different situations can be analysed to understand the state of the jobs. The section will proceed by defining what these collected values can be interpreted.

When \( D_{\text{max}}^T >> 0 \)

In the situation \( D_{\text{max}}^T \) goes towards 0, the difference between the \( T_t \) and \( T_{\text{max}}^W \) should be smaller. Therefore the slowest worker takes as much time as the total time taken to complete. This also means that the workers are more synchronized and almost start processing at similar times. The situation where \( D_{\text{max}}^T \) is equal to 0, is an ideal situation where all workers are perfectly synchronized.

\[ D_{95}^T \text{ is LARGE } \&\& \text{ } D_{\text{max}}^T \text{ is SMALL} \]

In the situation where \( D_{\text{max}}^T \) is small, the job can represent a good synchronization between the workers. This means that all processing of the job is completed within the time frame during processing of the slowest worker. The value of a large \( D_{95}^T \) represents that only a few workers use a lot of time to process. The worker management can react to this by starting a duplicate of the slowest worker. This can in the long term increase the amount of jobs completed in a lesser amount of time. A duplicate of the slowest worker can means that 2 jobs are processed in parallel between these workers.

\[ D_{95}^T > D_{\text{max}}^T \]

The situation where \( D_{\text{max}}^T \) is smaller than \( D_{95}^T \) is similar to the one presented above. The difference is that if \( D_{\text{max}}^T \) increases, it means that a few of the workers are out of synchronization. It is usually caused by a worker that starts later than the others due to longer processing times in the previous jobs. In similarity to the action above, the worker management can react to this by starting up the slowest workers. The worker management can react to this by monitoring the \( D_{95}^T \) and \( D_{\text{max}}^T \). Once these values are over a certain threshold, a duplicate of the slowest worker can be booted up.
4.6. THE STATISTICAL ASPECT OF THE TASK TRACKER (PERFORMANCE METRICS)

\[ D_{95}^T \text{ is LARGE } \& \& \ D_{\text{max}}^T \text{ is LARGE} \]

The situation where \( D_{95}^T \) is large, means that in average all workers use a smaller amount of time to process the query. When \( D_{\text{max}}^T \) is large as well, it means that the slowest worker uses much lesser time than the total time used for completing the job. The only possible explanation to this is only that the workers are poorly synchronized which causes the longer completion time. The worker management can react to this by monitoring the values. When the values increase above a certain threshold, a duplicate cluster of workers can be booted up. This can attempt to increase synchronization between the workers and result in a lesser completion time.
Chapter 5

Results II: Implementation

This chapter will describe how the task tracker and the other components are implemented in code according to the design. Some of the milestones in this chapter are:

- Identifying prerequisites for the framework
- Describing how the objectives mentioned in the design chapter regarding the task tracker were implemented
- Describing how the client, worker and reduce nodes were implemented
- Scripts that were developed to start the framework showing a runtime example
- Scripts that were developed to analyze data gathered when performing experiments

5.1 Prerequisites

This section attempts to describe the pre-requisites that must be in place, prior to implementing the objectives mentioned above. As mentioned in the approach chapter, this project will be developed from the scratch. The environment that was chosen to run the applications were of the Debian Linux type. In specific the command-line terminal was used. This meant also that bash scripting was involved to get the framework running. The methods in how bash scripting was used is mentioned later on. To both provide version control and backup, Git will be used. Using Git helped when the system was broken to a last known good configuration. Therefore very little or no time was spent on trying to fix the system in case of larger errors, since it was always rolled back.

As seen above from the design chapter, there a different types of elements and action that take place. Due to both the time duration
and the ease of development, Python was chosen as the language of development. JSON objects, communication between the components, computing statistics and creating visual graphs are some of the tasks that were required to be performed. Python provided rich and optimized libraries for all these tasks. Some of these libraries were not pre-installed and therefore were required to be installed before the application can be run. Therefore, Docker files were created, that install the dependencies required for the application to work. As mentioned in background chapter, this reduced the hassle of running the application when working on machines with different operating systems. The Docker file written for running the task tracker is shown below.

```
FROM ubuntu:16.04
RUN apt-get update && apt-get -y install python python-pip
RUN pip install numpy
ADD src /opt/src
EXPOSE 5005:5005
CMD /opt/src/server.py -v
```

First development of the components involved in establishing and sending and receiving packets between them. A basic TCP connection is used to send and receive messages. This is shown in the first snippet below. The task tracker should be capable of continuously receive and send packets simultaneously. These packets can vary in type therefore a traffic control system was implemented, which is shown in the second snippet. By implementing this system the task tracker routed the packet to the different methods according to the contents of the package. This was tested by creating fake packets as JSON objects to send to the task tracker. As mentioned in the design chapter, the JSON objects were all identified by the type of the packet. The "type" field in the JSON object was included for the purpose of both identification of the packet and traffic control.

```
TCP_IP = "0.0.0.0"
TCP_PORT = PORT
BUFFER_SIZE = 8024

s.bind((TCP_IP, TCP_PORT))
s.listen(5)

conn, addr = s.accept()
data = conn.recv(BUFFER_SIZE)

if ( data_json["type"] == "job" and "jid" in data_json ):
    get_job_status(conn,data_json)
```
5.2. JSON OBJECTS IMPLEMENTED

There was very little or no intention on developing a graphical user interface (GUI) to send requests to the system, therefore a command line GUI was chosen. Both the worker nodes, client and task tracker have a number of option that should be specified. As an example a worker should mention the address of the task tracker, the port it will be connected to, the part of corpus and shard it possess and the interval that will be used to connect to the task tracker. All these features will be specified via the command line as arguments. To solve this problem a inbuilt library from Python 'argparse' was used to specify the arguments that can be used. A help command was also included describing what each option stands for. A runtime example of the worker with arguments is shown below.

```
python worker.py -s SERVER -p PORT -c CORPUS -s SHARD -i INTERVAL
```

The framework was developed with the mindset and prediction of using time for troubleshooting. Therefore a debug and verbose mode was included in all the components. The purpose of the verbose mode was to print out all the packets that are both sent and recieved from the component. The debug mode printed out the different phases the component were during packet processing.

```
def verbose(text):
    if VERBOSE:
        print("VERBOSE: " + text + "\n")

def debug(text):
    if DEBUG:
        print "VERBOSE: " + text
```

5.2 JSON Objects implemented

This was first documented before the actual implementation took place. The reason for this was the structure and content of JSON objects being clarified in the design chapter. The design chapter mentioned all the different types of JSON objects that are exchanged between the components at every stage of processing. The basic idea of what the JSON object will
include was created at first based on the design. During the development of all 3 components, minor changes in the design gave arise to minor changes in the JSON objects. This section will include a snippet of the different types of JSON objects that were finally implemented and a short description of the JSON fields that are created.

5.2.1 Between Worker and Task tracker

This subsection will focus on all communication between the worker and the task tracker. The task tracker implementation presented later on, discusses the minor changes done to the design on how workers exchange messages with the task tracker.

A map worker announcing itself to the task tracker

```
{
   "type" : "hello",
   "wid" : "wikipedia-A",
   "cid" : "wikipedia",
   "shid" : "A",
   "working_on" : "",
   "result_list" : ""
}
```

Listing 10: Hello message from the map worker to a task tracker in the startup phase

The snippet 10 shows a map worker that announce itself to the task tracker. As seen from the message here the implementation includes a result list and a working on field, which was originally not specified in the design. This was due to the fact that all worker communication to the task tracker was merged together due reducing the different type of messages being sent. In the startup phase the working on and the result list fields are empty, and are later filled up as jobs arrive for the corpus. The type of this JSON object is of the type "hello", which is used to identify the type of message as it arrives to the task tracker.

A reduce worker announcing itself to the task tracker

The reduce worker communicates in a similar way to how the map worker does. The only differences between the reduce and map workers are the "rid" which stands for reducer ID and "wid" in the map worker that stands for worker id. The reducer does not include a shard id ("shid") as well.
5.2. JSON OBJECTS IMPLEMENTED

Listing 11: Hello message from the reduce worker to a task tracker

```
{
    "type" : "hello",
    "rid" : "reducer1",
    "cid" : "wikipedia",
    "working_on" : "JS94KF934KW",
    "result_list" : "[STRYUTP09E,JDIEJR893V]"
}
```

A decree to a worker node

```
{
    "type" : "decree",
    "job_list" : "[JOJO342DO,J420FDJ496]",
    "worked_on" : "[]"
}
```

Listing 12: A decree message on a processing task

This message is given by the task tracker to a worker. The type of it was defined as a "decree", that commands the worker on the jobs being processed. It is a simple message which sends the job object in the job list. The worked on field gives an overview of the other jobs that are being worked on by other workers with the same shard id.

5.2.2 Between Client and Task tracker

This subsection will focus on messages that are being sent from the client to the task tracker

Submission of a job

This is a very straightforward message, that includes all the necessary information to start a job. The corpus, query and user are all input from the command line interface. The reply will contain the job id. As seen from the listing 13, both the request and reply are of the same type "job".

List of all jobs

The list of all jobs is a simple request of the type "list" as seen from the listing 14. The user id has to be input to list all jobs pertaining to the user. The job list contains job objects that can be parsed for information.
CHAPTER 5. RESULTS II: IMPLEMENTATION

Listing 13: A job submission and reply

Result on a specific job

This job status request is of the same type “job” in similarity to the job submission request. The difference between the 2 requests is the inclusion of the job id field in this request. This is shown in the listing 15.

5.2.3 List of workers and reducers

The snippet in listing 16 shows 4 types of messages being sent between the task tracker and the client. The first 2 types are requests to send a worker list and reducer list. The 2 bottom JSON messages are the replies that are sent by the task tracker. They contain a list of worker and reducer objects.

5.2.4 Jobs and worker objects

The above subsections mention storing jobs as job objects and workers as worker objects. This subsection will show how a job and worker object is structured.

Job Objects

The listing 17 shows the final job object that is stored in the job list and sent to the client when job result is requested. The message begins with in a smaller form as shown in 13 , and during processing and
5.2. JSON OBJECTS IMPLEMENTED

```
{
    "type" : "list",
    "user" : "gheeth"
}
{
    "type" : "list",
    "user" : "gheeth",
    "job_list" : "[ASDF823VF,342DWER24]" # List of job object
}
```

Listing 14: Job list request and reply from the task tracker to the client

```
{
    "type" : "job",
    "user" : "gheeth",
    "jid" : "ASDF823VF"
}
```

Listing 15: Job status request and reply from the task tracker to the client

The transfer is extended with more information. This snippet shows a job without a reduce function. If a reduce function is included, the message is extended by the "reduce_script", "reduce_time_started", "reduce_time_finished" and "map_results". These are the largest messages that are transmitted.

Worker objects

The workers have a much more simpler and smaller structure. The information that is stored are the shard id, the corpus id and the ip address of the workers. In case of a reduce worker, the "shid" field was removed.

5.2.5 Performance metrics object

The statistics tree had a more complicated structure. The snippet 23 shows an example of a job’s statistics. It contains all the statistics ($D_{95}^T$, $D_{Max}^T$, $T_{95}^W$, $T_{Max}^W$, $T_{Min}^W$, $T_i$, $T_{str}$, $T_{adm}$) that were discussed in the design chapter. Objects of this sort are sent to the client when statistics for a certain tree is requested. The $T_{Max}^W$ and $T_{Min}^W$ contain the shard ids and the time taken for the slowest and fastest worker to complete. The "shid_list" provides a detailed timestamps on when each worker started and finished.
CHAPTER 5. RESULTS II: IMPLEMENTATION

Listing 16: Request and reply for list all workers and reducers.

5.3 Task tracker implementation

This phase is the second phase of the implemented process. After all the prerequisites were identified and written in code, the objectives were implemented according to the design mentioned above. A list of the objectives that were implemented are listed below. This section will continue on describing how each of the objectives have been realized and the JSON objects that were created to assist in communication.

- A worker connects with the task tracker for the first time
- A job is received by the task tracker and is processed for delivery to the workers
- A worker connects with the task tracker and receives a list of jobs
- A worker is done with its job and sends the results back to the task tracker
- A client sends a request on the status of the list of jobs sent by the user
- Task tracker prepares a reduce computation
- The user requests the result for a job
- Handle failed workers
5.3. TASK TRACKER IMPLEMENTATION

The design chapter mentioned about data structures for storing data on workers, users, jobs and worker queues. Python dictionaries were implemented for all types of data structures. Dictionaries in Python feature a key-value principle where only one unique occurrence of a key-value pair is possible. The data structures created for book keeping workers, users and job were identified by their unique id numbers. By using Python, the library "random" was used to generate a 10 character random string with both uppercase character and numbers. JSON objects were chosen to be stored as the values in these dictionaries, since it required no manipulation when sent to processing.

The data structure of the job status tree was more complicated than expected. The use of this tree is to keep in track of jobs that have been sent for processing to the task tracker. The diagram 5.1 shows how the tree is structured. For each corpus that is present there must atleast be one shard. This shard contains an array over all the jobs that have been distributed to it. The array type provides a FIFO (First in First out) queue system. The

Listing 17: Job object without a reduce function

Listing 18: A map worker

5.3.1 Global data structures
CHAPTER 5. RESULTS II: IMPLEMENTATION

![Diagram of the statistics tree](image)

**Figure 5.1: The structure of the statistics tree. The character S represents a Shard. The character J represents a Job.**

intention of this tree is so that the incoming workers get their own personal queue on what jobs are to be processed. The methods on how the worker receives the jobs is discussed later on in the chapter.

A dictionary structure was chosen to store information regarding statistics computed for each job. This tree had a similar shape to the job status tree. The first level of the dictionary contained the corpus as its keys and a dictionary of the jobs relating to the corpus as its values. The values contained a dictionary structure with the keys being the job id and the values being a JSON object containing all the statistical information. In order to assist the reduce function, an array was chosen as the data structure of the queue. Since the reduce function is optional and contains no shards, the structure was not complicated as the worker queues.

### 5.3.2 Worker connection to the task tracker

The design chapter mentioned 3 different cases when a worker connects to a task tracker. By looking at the design, a lot of similarities were found. All 3 objects exchanged almost the same type of JSON messages. They all had to have a corpus id, shard id and location. Therefore all 3 tasks were combined into a single task. This resulted in a lesser complicated structure but a larger JSON message. The first 2 tasks were based on a JSON object type, "hello"

The task first started by checking against the worker list to see if the worker if a sign in or a sign up is performed. If the worker exists in the register, the process of signing up is skipped and the location is updated. As mentioned in the design chapter, handling failed workers required a last seen time stamp. During sign in, this counter was restarted to show that the worker had signed in recently.
The second task that was chosen to execute was processing the list of finished jobs by the worker. Here the results was parsed through and for each job the worker had processed, the results were stored locally. The results were stored in each job’s JSON object under a field known as "results". The task is then removed from the worker queue. The task tracker also checked if there were any remainings shards that had not returned with the job results. The time stamps of each worker are recorded in the statistics tree under the job id. If there were no shards remaining, the job was marked as "map_ finished" or "finished" depending on if a reduce function was provided. Once the job is marked as finished the finish time, total time used and the issued time is recorded.

The final task is to submit a list of new jobs to the task tracker. For this final task the JSON object with type "decree" was created. This was done by checking the worker queue list for the corpus and the shard the worker belonged to. The queue was parsed to see if there were any jobs that were not issued to avoid duplicative processing. The first job that was found to be not issued to a duplicate worker, was marked as "issued" and the job details were appended to the worker’s job list. The status of the jobs "issued", "finished" and "queued" are represented by a number . This was then sent back to the worker for processing. If no jobs are available for processing an empty job list is sent back.
mod_job = copy.deepcopy(JOB_LIST[jid])
mod_job["results"] = dict()
decree["job_list"].append(mod_job)
JOB_LIST[jid]["shid_list"] = hello["shid"] =

## 5.3.3 Job received by task tracker from a client

As mentioned in design chapter, the job request is sent in the form of a JSON object. The task tracker is implemented to check if the corpus mentioned in the JSON message is found. If the corpus is not found, an error message is sent back to the client. If the corpus is found, a JSON object is sent back to the client containing the job id that is randomly generated. In addition to this the JSON object that is created is filled up with details such as "time created". Timestamps are generated by using the Python library "time". This gives the actual system time to the second as a whole number. The status of the job is then set to "queued". By gathering the number of shards present for the corpus a dictionary is created as a JSON field, to keep track of the status of each shards.

These are then stored along with each of the shards. At the initial stage all the shards get a status 0. Implementing this made it relatively easy to pinpoint how far the job had come into its processing. The JSON object created is then stored in the global job list. The random job id created is used as the key and the JSON object as the value. The job id is then stored on all the corpus's worker queues. This is done by using a simple for loop to parse through all the shards.

```python
workers = dict()
for shid in STAT_TREE[job_json["cid"]]:
    STAT_TREE[job_json["cid"]][shid].append(job_json["jid"])
    workers[shid] = 0
job_json["shid_list"] = workers
job_json["shid_remain"] = len(workers)
job_json["results"] = dict()
```

## 5.3.4 Client requests on status of all jobs

This task was one of the more easier tasks to implement. A request of this sort is a type "list" JSON object. The design chapter mentioned on providing a sense of security for the client, and therefore it was made compulsory to include a user field in the JSON object being sent by the client. This was required to send jobs that pertain to the user. The list of jobs and their statuses were sent, along with the status of each shards. To reduce the size of the message that was sent, a copy of the JSON object is created
and the "results" field was emptied. All these results were encapsuled in the same type of "list" object and sent back to the client.

```python
data_json["job_list"] = []
if data_json["user"] in USER_LIST:
    for jid in USER_LIST[data_json["user"]]:
        job = copy.deepcopy(JOB_LIST[jid])
```

5.3.5 Task tracker preparing a reduce computation

The task tracker was implemented for the user to specify a reduce script as an input. The function of performing a reduce computation is implemented immediately after the jobs is marked as finish. A simple if clause checks to see if a reduce function is included with the job. If a reduce function was present, the job was appended to the reduce queue as shown in the snippet below. The reduce nodes perform in a similar method to the worker nodes. It first responds with with a "hello" and is checked against a register if it is a sign up or a sign in. If it is a sign in the reduce node is checked for jobs that have been reduced. These results are stored in the task trackers local list. The reduce nodes were made to be both corpus specific and general. The JSON object sent from a reduce node has a "corpus" field. If empty this was a general worker. The option for handling a corpus specific node was created in case a corpus required a dedicated reduce node. Therefore the reduce node that was connected was checked to see if it was corpus specific. If not, the first job of the queue was sent to the reduce node and the task was erased from the queue. If the reduce node is corpus specific then a simple for each loop was created to parse through the reduce queue and the first job in the queue with the same corpus as the reduce nodes corpus was sent to the reduce node. The job was then removed from the queue.

```python
if "reduce_script" in JOB_LIST[job["jid"]]:
    REDUCE_QUEUE.append(job["jid"])
    JOB_LIST[job["jid"]]["status"] = "map_finished"
```

5.3.6 Client requesting the results for a job

For this task the client had to send a JSON object with the type "job". The "job_id" and the "user_id" had to also be specified. By specifying both the user id and the job id, a sense of security is created as to not peek into another's job. The task tracker then verifies if the job id and belongs to the user, and then sends back the JSON object of the job. This is done to reduce processing on the task tracker. At first, the implementation sent the status
of the job if the job was not done, but if the job was done the results were sent. This was identified as a time consuming process and therefore, the whole JSON object was sent back and the client had the job of processing the object. This is discussed later in the chapter.

```python
if data_json['jid'] in JOB_LIST:
    client.send(json.dumps(JOB_LIST[data_json['jid']]))
```

### 5.3.7 Providing Performance metrics

The task is initiated once all map computation for a job is completed. The computation is mainly performed on the data based from the statistics tree. The methods implemented are a direct implementation of the equations designed in the design chapter. First the slowest shard and the fastest shards are identified. Then the "math" library of Python is used to compute the 95th percentile of time that a worker uses for the computation. Once all the calculations are completed, the data is saved on the statistics tree under the job id.

```python
STATISTICS_TREE[cid]["job_list"]['jid']['Tw95'] =
    str(int(np.percentile(STATISTICS_TREE[cid]["job_list"]['jid']['shid_list_used'].values(),95)))
STATISTICS_TREE[cid]["job_list"]['jid']['DtMax'] =
    int(STATISTICS_TREE[cid]["job_list"]['jid']['Tt']) - Twmax[1]
STATISTICS_TREE[cid]["job_list"]['jid']['Dt95'] =
    int(STATISTICS_TREE[cid]["job_list"]['jid']['Tt']) -
    int(STATISTICS_TREE[cid]["job_list"]['jid']['Tw95'])
```

### 5.3.8 Additional features implemented

The task tracker had also additional features implemented that was not mentioned in the design. These functions are used mostly for debugging the task tracker and are not essential for the task tracker to perform.

- Deleting a job
- Listing all the workers
- Listing all reducer nodes
- Exporting statistics for a corpus
5.4 Worker Implementation

Deleting a job

This was implemented in case a user decided to delete the job after submission. The job can be in different stages of processing. Not issued, Issued and Finished are the most common stages. In case the user decided to delete the job, the job had to be first identified by the user and job id. Then the job id was removed from all the workers queues that had not been sent to processing. The shards that had been sent to processing was ignored. The job status in the job list was also changed to "deleted".

Listing all the workers

This was implemented to check the status of all the workers. When the client sends a "worker_list" request to the task tracker, the task tracker responds by dumping the whole register of worker JSON objects from the worker list into an array of JSON objects. This is then sent to the client, which then prints out the workers on the screen.

Listing all the reducers

This is very similar to the listing all the workers. The client has to send a "reducer_list" request to the task tracker. The task tracker then encapsules all the registered worker’s JSON objects and sends it to the client. The client then prints out all the reducers on the screen.

Exporting statistics for a corpus

For the purpose of analysis the statistics that are computed for each job are placed in a JSON object. These were stored in the statistics tree. When statistics for a corpse is requested, the statistical data found for the specific corpse is encapsuled into a JSON object and sent back to the client. The JSON objects were formatted and printed out clearly on the screen. One could also pipe the output to a text file using the command line.

```python
for i in STATISTICS_TREE[cid]["job_list"]:
    job = copy.deepcopy(STATISTICS_TREE[cid]["job_list"][i])
    job["jid"] = i
    data_json["job_list"].append(job)
```
tions that were implemented in both the workers were necessary to test all the functions in the task tracker. Although these are not the workers that are proposed to be used in the future, they perform a similar function. This section describes what is common for both map and reduce nodes, and the differences later on.

5.4.1 Common to both Map and Reduce Nodes

Both map and reduce nodes are classified as worker node. They possess many similarities such as startup, communication to the task tracker and JSON communication between the task tracker and worker. This subsection describes below how features common to both the map and reduce nodes are implemented.

Communication to the task tracker

As mentioned earlier all the components including the workers are initialized by using the command line. Arguments such as the corpus name, shard id, the location of the server and the port used to connect are compulsory for the task tracker to work. The workers first have a method that use the arguments specified to create a connection to the task tracker.

```python
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((SERVER, PORT))
```

The method of connection is shown above. Once a connection is established, a JSON object regarding the corpus, shard and an empty result list is sent to the task tracker. This message is defined as the type "hello". Once signup from the task tracker is completed, a job list may arrive. If no job list is received, the worker will sleep until the next time the task tracker is contacted. The wait interval is set at a default 60 seconds, but there is a command line option to either increase or decrease this period.

Processing jobs that are received by the worker

If a job list is received, the worker will parse through the job list by using a for each loop. Jobs that are sent from the task tracker contain the whole job JSON object. Therefore the jobs that are parsed through are sent to the workers function (map or reduce). The job that has to be executed is temporarily stored on a list known as "WORKING_ON". Prior to performing the calculation, time stamps such as time started and time finished are recorded for the purpose of statistics, which is achieved by using the Python library "time". Once the function is performed, the results are stored in the same JSON object. The job that was stored on the list "WORKING_ON"
5.4. WORKER IMPLEMENTATION

is the moved to a new list "FINISHED_ LIST". These lists were used to notify the task tracker of the jobs that are currently in processing and the jobs that are completed, in case the duration of the interval is shorter than the duration of the job.

```python
for job in decree["job_list"]:  
    start_work(job)
```

Transmitting jobs that are processed

Communication with the task tracker is present only at the interval that is provided. The jobs that are finished are encapsulated in the "hello" message with the job finished appended in the "result_ list". The JSON object is then sent to the task tracker for storing the job.

5.4.2 Map Node Workers

The map nodes are the first nodes to recieve the job. In theory the map function is individually defined as to what the map will compute. Since these workers are defined to only simulate a test, the query that was performed is a sleep function for the number that is specified by the client. This sleep time is considered to be the number of seconds a worker uses for the job. For the purpose of individually adjusting a worker to perform a longer map function, an advanced query format that slept longer on random workers was implemented. This was done to depict a near real life scenario. It involved using semicolons and colons.

```python
if ";" in query:
    arglist = query.split(";")
    waitlist = arglist[1].split(":")
    if int(SHID) == int(waitlist[0]):
        time.sleep(int(waitlist[1]))
    else:
        time.sleep(int(arglist[0]))
```

After the sleep time was completed, a random string of the order of 100 was saved as the result. Once this was completed the next job was sent to processing.

5.4.3 Reduce Node Workers

The reduce nodes recieved the job after all mapping function relating to the job was completed. Here the user had the option of specifying the reduce
CHAPTER 5. RESULTS II: IMPLEMENTATION

script that had to be performed. The reduce node began the computation by first creating a local temporary directory containing the script and results of all the map computations. The Python library "os" was used here for creating the folders and files.

```python
os.makedirs(workdir)
os.chdir(workdir)
```

Once these files were written, another library "subprocess" was used to start the reduce function. The example of a reduce function that was used was to sum up the number of occurrences of the character "A" in all the map results. The subprocess library invokes a bash terminal to run the script and piped the output to the reduce node. The temporary folder is then deleted and the result is saved in the job's JSON object.

```python
output = subprocess.check_output(path, shell=True, executable='/bin/bash')
```

5.5 Client Implementation

The final piece of the implementation phase was to develop the client. As mentioned in the design chapter, the client had the main task of interacting with the user and the task tracker. The functions that were needed to be implemented to test the framework were on submitting a job, querying both the status and the result of the job. There were also additional features that were not mentioned in the design that were implemented for the sake of debugging the system. The list below shows the functions that were implemented and the section will proceed by describing these in detail.

- Submitting a job
- Viewing all jobs' status pertaining to a user
- Getting the job status or results
- Listing all the workers and reducers
- Printing out all the statistics pertaining to a corpus

5.5.1 Submitting a job

This task was predicted to be one of the most processed task by a client. To perform a job the corpus, query and the user id was compulsory. All this was provided via the command line interface. In addition the task
5.5. CLIENT IMPLEMENTATION

Tracker’s location and port used to connect were specified. This data was first used to establish a connection with the task tracker. Once connection was established, the job details gathered from the command line, was sent to the task tracker in a JSON object. The client then had to either wait for an error message or a job id. If a job id was received, the job was successfully submitted.

```python
job = {
    "type" : "job",
    "cid" : corpus,
    "query" : query,
    "time_created" : str(int(time.time())),
    "user" : user
}
s = connect_to_queen()
s.send(json.dumps(job))
```

5.5.2 Viewing all jobs’ status

This task was to allow a particular user to view all the jobs that pertains to the user. A user can submit many jobs and by viewing all the jobs at once, the user could get an overview of the status of each job. The client was used here from the command line and the task trackers location and port and the user’s name were all required arguments. Once this was provided a JSON object containing the user id was sent to the task tracker as shown in the snippet below. All jobs found under the specified user id is returned to the client as a JSON object. To make sense of this data, the JSON object is parsed through and the results are printed out in a pretty form.

```python
list_request = {
    "type" : "list",
    "user" : user
}
s = connect_to_queen()
s.send(json.dumps(list_request))
```

The column headers "JOB", "CORPUS", "QUERY", "CREATED", "STATUS" and "TIME USED" are printed in a table like format. The details of each job is printed underneath the column headers with the id underneath the job column. The time used is calculated from the time stamps that are recorded in the JSON object. By using simple mathematic calculations, the time is printed in a more user readable form. The status column describes the status of the job as “issued”, “queued”, “map finished” and “finished”.
5.5.3 A particular job status or result

The result or status of a job can be gathered while the job is under processing or if it is done. The client initiates the connection by specifying the task tracker location and port. The user id and the particular job id are then encapsulated into a JSON object and then sent to the task tracker. The task tracker returns the JSON object of the job that is stored in the global job list. During submission of the status check, the user can optionally specify 2 options that prints richer detail on the screen.

- **x** (extended) : This is used when the job is still processing to find out how many shards have been processed. If the shard is not issued a ‘.’ is printed out. If the shard has a job under processing a ‘o’ is printed out. An ‘x’ is printed out for the shard that has a result present in the JSON object. All this is printed out in a grid like format. The current map results that are stored the JSON object is printed underneath it.

  ```
  client ~/q/src> watch python client -p 5005 -s 128.39.121.73 -u gheeth -i OXN8E0H3UK -x
  ```

- **a** (all results) : This is only used when a reduce computation is performed. Once a reduce computation is performed, it is assumed that the user is only interested in the reduced result. If the user wishes to view the results of the map computation, the option ‘a’ can be provided to view them.

5.5.4 List all workers and reducers

This function is a simple function that returns all the workers and reduce nodes. By using the command line, the client provides only the task trackers location and the port to connect along with an argument. The only difference in the methods of requesting a workers and reduce nodes list is provided by an argument in the command line. When requesting a workers list the "W" option must be specified. This then sends a JSON request of the type "worker_ list" to the task tracker as shown in the snippet below. When requesting a reducers list the "R" option is specified. This sends a JSON request of the type "reducer_ list" to the task tracker. Once the task tracker processes the list a JSON object containing all the nodes are sent back to the client. The client then prints out the "CORPUS", "IP" and "LAST_ SEEN". Printing this is very similar to printing the list of jobs pertaining to an user. The corpus here shows what corpus the node belongs to. The ip and location show where the worker is found and when it last connected to the task tracker.

```python
list_request = {
    "type" : "worker_list",
}  ```


5.5.5 Print out statistics relating to a corpus

Statistics that are calculated for each job are stored in the task tracker. This data is requested from the task tracker by sending a JSON request of the type "stats_request. The corpus has also to be mentioned, since the statistics data is gathered per corpus. This information is sent from the client to the task tracker. The task tracker then sends the dump of the statistics that pertains to the corpus. This data is then formatted and printed out in a more readable manner as shown in listing 23.

5.6 Running the Framework

Once all the components are developed, they were run to check if the framework works as expected. The Docker files created for the components were then started.

5.6.1 Starting the framework

The environment used to simulate the framework is the Openstack cloud at the Oslo university college. For the task tracker a virtual machine with the flavor large was created. Docker was installed on this machine. At first 4 virtual machines were created with a flavor small for the task of running the workers. The virtual machines ran all Ubuntu 16.04. Since the components were backed up by using Git, git was used to clone the repository on all the virtual machines. A personal laptop was used for the purpose of running the client.

To start the framework the task tracker had to be started first. This was done by first building the Docker image and then running it.

docker run -p 5005:5005 queenbee

Once the task tracker was running the workers were started on the worker machines. On each machine the worker was started. The code below shows how each worker was started.

```bash
# worker1
worker1 ~/q/src> python worker.py -c wiki -p 5005 -s 128.39.121.73 -i 5 -S "A"
# worker2
```
CHAPTER 5. RESULTS II: IMPLEMENTATION

The options here specify the corpus id (-c), port number (-p), task tracker address (-s), connecting interval (-i) and shard id (-S). Once the workers connect to the task tracker, a confirmation message appears on the task tracker logs. The client was also used to check the number of worker that had registered with the task tracker.

```
python client.py -p 5005 -s 128.39.121.73 -W
```

The option (-W) was used to list the workers connected. The task tracker then showed a list of workers that had registered with the task tracker.

<table>
<thead>
<tr>
<th>CORPUS</th>
<th>SHID</th>
<th>WORKER</th>
<th>IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>wiki</td>
<td>B</td>
<td>VGUSA0M77T</td>
<td>128.39.121.74</td>
</tr>
<tr>
<td></td>
<td>3s ago</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wiki</td>
<td>D</td>
<td>CILEXJ7V7L</td>
<td>128.39.121.75</td>
</tr>
<tr>
<td></td>
<td>3s ago</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wiki</td>
<td>C</td>
<td>YIIVN2WXT4</td>
<td>128.39.121.76</td>
</tr>
<tr>
<td></td>
<td>1s ago</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wiki</td>
<td>A</td>
<td>S6D51HJNUV</td>
<td>128.39.121.77</td>
</tr>
<tr>
<td></td>
<td>3s ago</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.6.2 Start a job without a reduce computation

Once all the workers were connected to the task tracker, the laptop was used to send a query of estimated time 20 seconds to the task tracker. Once the query was sent a job id was printed out on the client. The options -p, -s and -c are similar to the options used when initiating the worker. The options (-u) refers to the user id and the option (-q) refers to the query that is submitted. This is an example of a request that does not require a reduce computation.

```
client ~/q/src> python client.py -p 5005 -s 128.39.121.73 -c wiki -u gheeth -q 20
OXN8E0H3UK
```

The client was then used to list out the status of all jobs pertaining to the user. Together with it the bash command "watch" was used to ping the
5.6. RUNNING THE FRAMEWORK

Task tracker with the status request to watch the progression of the job. This showed a live progression of how far the job had come into processing. The (-l) option was used to list all the jobs.

<table>
<thead>
<tr>
<th>JOB</th>
<th>CORPUS</th>
<th>QUERY</th>
<th>CREATED</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>OXN8E0H3UK</td>
<td>wiki</td>
<td>20</td>
<td>3s ago</td>
<td>issued 2s ago</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50% queued, 50% issued</td>
</tr>
</tbody>
</table>

Once the job was completed the results were viewed by issuing the command for viewing job results on the client. This command replied the results for each individual shard. The option (-i) was used to specify the id of the job. The results shown in the snippet 20.

client ~/q/src> watch python client -p 5005 -s 128.39.121.73 -u gheeth -l

5.6.3 A job with a reduce computation

There was not much of a difference in starting a job with a reduce function. The reduce function also required reduce nodes to be present. A virtual machine was booted up for this purpose in a similar method to the worker machines started earlier. This is then connected to the task tracker.

reducer ~/q/src> python reducer.py -p 5005 -s 128.39.121.73 -c wiki -i 20

Once this is connected to the task tracker, a job can be submitted. The reduce function is specified in a script file by using the option (-r). The status of the job is viewed as mentioned earlier, and once map computations are finished, the status Map finished is displayed. This is shown in the listing 21. The script reduce.py was created to count all occurrences of the character ‘A’ in all map results. The contents of reduce.py is shown in the listing 19.

To view the results of the computation the command below was issued. The option (-a) was used to view both map and reduce results. The reduce result is displayed at first followed by the map results 22.
import sys
import json

with open(sys.argv[1]) as data_file:
    data = json.load(data_file)

count = 0
for v in data.values():
    count += v.count("A")
print count

Listing 19: The content of reduce.py
5.6. RUNNING THE FRAMEWORK

client "~/q/src" watch python client -p 5005 -s 128.39.121.73 -u gheeth -i 0XN8E0H3UK
100\% finished
{"A": "HKAFEEH20US4SULGSRONG3T93A8XNZI6XELL8ZEPYHRZEON5FZN689L22SI8GDCIXKCYXDP036KZMYA8CBV7QH4JS5PA7XOBL". "C":

"9HE57VDHPGC7DGXCT7XGWFL7EM4X9327HEW8B1YN34CEWE6G64CFXTM65EZJJG5T561HBBTL21D7XAN4142N34QH1AB5D8WS". "B":

"VBVCHTPAHLKYSWYOACLLLOO2PFADASR450KTFZ512GDADDG836DA98AYIC5Y32SWBNKXIBMRA98BMNIBQDOHZE5Z7PMKP683". "D":

"7WRANNIBM8XJEJNX25FJFW8YE13R0FNLALA196PBSQ01EGUI0TDUS1EAFNZWF214F2PDA94JYQ484VMH3LWX1XVCZ5871DAMU"}

Listing 20: Results of a MapReduce computation

client "~/q/src" python client.py -p 5005 -s 128.39.121.73 -c wiki -u gheeth -q 20 -r reduce.py
EQNV91DHCJ

client "~/q/src" python client.py -p 5005 -s 128.39.121.73 -u gheeth -l
EQNV91DHCJ

Listing 21: MapReduce computation job status
client "/q/src> python client.py -p 5005 -s 128.39.121.73 -u gheeth -i EQNV91DHCJ -a
100% finished
9

----------- MAP RESULTS -----------
{"A": "Y8O2KJ3IW304L9WP1KPP7XXYJHMYRHYRCC3G61JD1Q0S0BUS9TM3EWL63VADEIKWU65JP4ZRYDHCSSNC3JKLRS3R1XLN"}, "C":
  "BZCCFUA64RZJ77BVI64G46ABZ2GYTP2L66XK7M1XLPL79F4AKRQ5DE4V3K3UBFDY6PBRRQA13914X6VJKDINJAH2R730145"}, "B":
  "HPNGRUXMDXGQCNIACWQLTUKPL2IRFKYTTZQURKJ07NHZ58L9W56X57DK6AFUCZMGEDUQPNYYKQRMX8QKH136612CUVPY5UQR7"}, "D":
  "25H0IZPSXIIP73NMJQJZ3GJW89O7OG8PWCSC0H2XNV5XL94WDC2P90GUTBWC4C7FUGQBE5J8SVY9V3UG9M2NC4JZZZ7XKNU"

Listing 22: Result of a MapReduce computation with all results
5.6. RUNNING THE FRAMEWORK

5.6.4 Viewing statistics for the corpus

Statistics is viewed by issuing the command below. This command only requires the corpus name to be issued as an argument.

```bash
client ~/q/src> python client.py -p 5005 -s 128.39.121.73 -c wiki
```

```
{
  "corpus": "wiki",
  "job_list": [
    {
      "Dt95": 3,
      "DtMax": 3,
      "Tadm": 1,
      "Tstart_stop": 24,
      "Tt": 23,
      "Tw95": "20",
      "TwMax": [
        "A",
        20
      ],
      "TwMin": [
        "A",
        20
      ],
      "jid": "M5AMXN73WS",
      "shid_list": {
        "A": {
          "time_finished": 1494178012,
          "time_started": 1494177992,
          "time_used": 20
        }
      },
      "shid_list_used": {
        "A": 20
      }
    }
  ]
}
```

Listing 23: A example of a job from the statistics tree

The JSON objects are then printed onto the screen. A part of the statistics for the corpus wiki is shown in the snippet 23. To export this to a text file, the command shown below is used. This pipes the output to a text file and thereby writes it to the filename specified.

```bash
client ~/q/src> python client.py -p 5005 -s 128.39.121.73 -c wiki > stats.json
```
5.7 Large scale implementation

The implementation above mentioned testing the framework with 4 workers and 1 reducer. To test a large scale implementation, special software was required to be used. In the original implementation 1 worker was run per worker machine. The bash command "top" showed that the worker component used very little memory and CPU. The goal was to test the task tracker with 100 worker implementations. Therefore 6 more worker virtual machines were created. 10 worker instances per machine were run. This gave a total of 100 worker instances. Manually starting this 100 instances would take up a long time, therefore bash scripting was used to create the startup commands of these workers. The script that was used to create the start up scripts for all 10 workers is mentioned below.

```bash
for j in $(seq 0 1 10) ; do for i in $(seq $j 10 100) ; do echo
"/mnt/timer/worker.py -c wiki -w worker_$i -i 15 -S $i -s "$i" >>
start_worker_$j.sh ; echo "sleep 10" >> start_worker_$j.sh; done ; done
; chmod +x start_worker*
```

In short two for loops are used to create these files. The first one is used for the file identification, and base number. The second for loop hops over by a number of 10 + the base number and appends it to the file start_worker. The script start_worker_1 will then have start scripts of the workers 1, 11, 21, 31, 41, 51, 61, 71, 81 and 91.

To start the start script on all 10 machines the scripts were first copied by using the standard copy tool "scp" to copy the files over ssh. Once the scripts were copied another program "parallel-ssh" was installed on one machine with access to all 10 machines. The command shown below was issued to start the start scripts on all workers together. The option (-h) was used to specify all the ip addresses of the worker machines in a file, and the command to run was specified with the argument (-i)

```
parallel-ssh -h /root/workers.txt -i '/mnt/timer/start_$HOSTNAME.sh 128.39.29.135'
```

The virtual machine used for the reducer earlier was also issued a script to run 10 instances of the reducer component. This gave a total of 100 map workers and 10 reducer nodes.

5.8 Test Instrumentation

When large scale experiments were performed and statistics gathered, there was a lot of lines of data that had to be analysed. With the setup mentioned above, 20 jobs issued to the task tracker produced 12500 lines of
5.8. TEST INSTRUMENTATION

To both analyse and produce graphs from this data, 3 Python scripts were created.

- Worker performance per job
- Generate $D_{l_{\text{max}}}$ vs time
- Job distribution graph per experiment

The worker performance per job graph produced a chart per job showing when the different workers processed the job. Since the statistics were in the format of JSON, the data was parsable. This was then put into arrays and processed. By using the Python library "matplotlib" the graphs were generated. An example of a job with 100 workers is shown below. The horizontal lines show the worker processing periods. This is shown in 5.2.

![Job number 15 with id 714HHV340C](image)

Figure 5.2: A detailed graph showing the start and stop times for each worker for a job. The x axis shows the time used and the y axis shows the worker ids.

The graph of $D_{l_{\text{max}}}$ vs time was produced to see the effect of job completion times when worker count was altered. This was a relatively easier script to produce since the $D_{l_{\text{max}}}$ of each job is computed in the statistics tree. This was done by generating a script that extracted this information from each job and generated the graph as shown in 5.3.

The graph of job distribution was produced to see the effect of overall experiment completion times, when a change in the amount of workers were induced. The experiments conducted involved more than one job and
Figure 5.3: A detailed graph showing the progression of $D_{\text{Max}}$ for an experiment. The x axis shows the job ids and the y axis shows the time in seconds.

therefore it was interesting to find the rate of jobs that were processed. The output of this script is a graph that shows the job from $T_{\text{issued}}$ to $T_{\text{finished}}$. This is plotted against time as shown in 5.4.

5.9 Summary

This chapter sums up the implementation phase of this project. All phases that were mentioned in the design chapter was implemented. A list of what was implemented is presented below

Task Tracker

- Implement global data structures to store data
- Manage worker lists and connections
- Implement job management features such as receive, store and distribute jobs
- Calculate and provide performance metrics
Worker Implementation

- Implement a prototype of a worker that performs the minimal function
- Implement a customized map worker node
- Implement a customized reduce worker node

Client Implementation

- Implement a prototype of a client
- Submit job queries from the command line
- List all workers and reducers
- Request status and results of jobs
- View all jobs pertaining to a user

Additional tasks

- Runtime example of a map cycle
• Scripts for a large scale implementation
• Scripts for analysing statistical data
Chapter 6

Results III: Experimentation & Analysis

This chapter will explain in detail both the experimentation and analysis phase. First the prerequisites for the experiments are mentioned. Then the chapter will progress into the basic tests that were performed to verify the functionality of the task tracker. Once these are verified, the chapter will progress with the biased tests to understand further the task tracker's behaviour. These will be in the form of smaller questions which are answered by the influenced tests.

6.1 Prerequisites

Prior to experiments being performed, the framework had to be in place. This meant that the large scale implementation scenario that was mentioned in the implementation chapter had to respond.

To check if the framework was ready the commands to list all workers from the client was performed. On the first go this was not completely perfect since, some workers did not start up at all. After fine tuning the start up scripts for each worker, the workers without the reduce nodes started in about 48 seconds to start up.

The implementation chapter showed an example at runtime, with 4 workers connected to the task tracker. As mentioned in the background chapter a typical MapReduce framework handles about a 1000 workers. Testing an implementation of 1000 workers was not possible due to the availability of virtual machines. When testing a 100 workers, 10 virtual machines were used each hosting 10 worker instances. When this was doubled to 20 worker instances, the worker machines crashed and therefore, the most stable configuration was 10 worker instances per virtual machine.

To test the boundaries of the maximum number of workers the task
tracker can handle, 10 more virtual machines were booted up with similar start_worker scripts. The differences in these worker instances were that they belonged to another corpus and therefore had a different map function programmed. This made it to a total of 200 worker nodes. The reduce nodes were signed up with the task tracker as well making it a total of 200 worker nodes and 20 reduce nodes. Surprisingly this also took around 48 seconds to startup. This meant that there was no lag in sign up of the worker nodes, but rather the 5 second lag that was included in between starting each worker instance. The task tracker was run for 5 minutes, and all workers had connected to the task tracker at the specified intervals.

6.2 Basic Tests

The aim of these tests was to check and verify the basic functionality of the task tracker. This meant that the task tracker was subject to all types of tasks such as submitting different types of jobs. These tasks had to be verified, before the influenced tests are performed. A list of basic tasks that are required to be performed are listed below.

1. Send and request a result of job
2. Send a job with a reduce function
3. Try accessing a job that does not belong to the user
4. Delete a job
5. Verify job completion after a worker migrates

The process of a sending a job is the main task of the task tracker. The job that was sent to the task tracker was successfully processed. Once a request with a query of 30 seconds was sent, a job id was returned. This job id was used to both query the status of the result until map was finished. Prior to testing this function, 200 workers were registered with the task tracker. 100 of them belonged to corpusA and the rest belonged to corpusB. The map function performed on each corpus was defined differently. Part of the experiment was also to check if the correct results were printed out, in respect to the corpus’s map function. The expected type of results was printed out.

- The time taken until job id was returned was almost less than a second.
- The job took 47 seconds to run to completion.
- Time taken for the client to print the reply was almost a second.
To benchmark the task tracker as a base for comparison with the influenced tests, 30 jobs to corpus A, were sent by using a bash script. The query was set for 30 seconds. 100 workers were used to process this. The computation was completed in roughly 10 minutes. An interesting observation was made during the processing of a number of jobs. The workers processed one job at a time and waited until the next sign in with the task tracker until the next job was received. The graph below shows the progression of the jobs. As seen from the graph, the $T_{issed}$ and $T_{stopped}$ line run in parallel, depicting that no unexpected situations arose during the processing of the jobs. This also means that all worker sign-ins were received and processed in time.

The second task was to send a job with a reduce function to the task tracker. The difference between the previous task and this one is that the workers have a map function that is pre-programmed. The reduce function that is provided here is by the user. The job that was sent to the task tracker with the reduce function was performed successfully and the result returned. To verify the results the map results were printed out and they were manually fed into the reduce function. The results were similar verifying the success of the job.

- As before the job took 47 seconds to map completion.
- The reduce function took 10 seconds to complete.
The third task to be performed is accessing a job that does not belong to the user. Jobs were registered under another user’s name and were attempted to access. The client printed out an error message stating no job belonging to the user id was found. This implied that security was implemented as to not show the results of queries performed by other users.

The fourth task to be performed was to delete a job. This task was performed in 2 stages. The first is before the job was issued and second when the job was under processing. The job was deleted successfully both times. When a status for the job was requested, the client replied by printing out that the job was deleted. If the job was deleted under processing, there was no access to the results of the job.

The fifth task involved verifying job progression after worker migration. There were two scenarios that were possible. The first was worker migration during processing of a job. The second was when a worker that did not have a job under processing. This was tested by issuing 5 jobs to the task tracker. After 2 jobs were complete and the third job was under processing, a duplicate cluster with the same amount of workers were started and the original cluster was stopped. This was to imitate a migration. The worker migration was verified by listing the workers. The newly booted up cluster responded from a new IP address. The workers had migrated and the fourth and fifth job was issued to the newly migrated cluster. Therefore 4 out of 5 jobs were marked as complete. The interesting fact was that the third job was never completed, despite the fact that the task tracker was kept alive for more than 5 minutes.

The restrictions in hardware disabled the scaling to more than 200 worker instances. A long term test was performed on the task tracker by using the client to send 5000 jobs of a query of 30 seconds to the task tracker. This test was completed successfully in 41 hours and 35 minutes, and the client was able to be used to retrieve information on any job.

6.3 Influenced Experiments

This section will proceed with the influenced experiments that are performed to understand the behaviour of the task tracker. These tasks are influenced in the fact that the amount of workers and jobs performed are varied by force to understand how the task tracker reacts. A secondary goal of these experiments is to find out if the performance metrics delivered by the task tracker is sufficient enough for the future projects such as worker management. The list of experiments that are performed are listed below.

- Effect of a slow worker on overall performance
- Effect of a duplicate slow worker after half the jobs are complete
- Understand job scheduling between 2 corpuses
6.3. INFLUENCED EXPERIMENTS

• Doubled cluster performance after half the jobs are complete.

6.3.1 Effect of a slow worker on overall performance

How is the overall performance affected by a slow worker?

The aim of this experiment is to find out how the overall job completion time is affected by a slow worker. This is attempted to be found out by statistics that are collected during the experiments. The reason behind the influenced slow worker is to imitate a single shard using longer time due a more focused map function a particular shard. Such example is a repetitive query for "counting the number of words in articles beginning i the character A". The shard A will be then overloaded with processing the requests while, the other shards will have finished processing.

This experiment was performed by booting up 99 workers. The worker depicting the slow worker, was programmed with the map function sleeping 15 seconds extra compared to the other workers. This was given an identification number of 1. Once the framework was up, 20 jobs with a 30 second query was sent in succession to the task tracker. No reduce function was computed. When waiting for the experiment to complete, the progression of jobs was watched closely. Most of the jobs had reached the stage "99% finished 1% queued", meaning that 1 worker had yet to process the job. The 20 jobs was completed in 15 minutes. The statistics relating to the experiment was gathered and this data was plotted into graphs.

The graph 6.2 shows the results of the experiment that was performed. The parallel black lines show how long each job used in processing in ascending order. The blue line shows when the job was first issued to a worker until the last worker has reported the results. As seen from the graph the time used for each job increases. The first job uses roughly 45 seconds and the time used increases gradually where the last job is seen to use over 5 minutes. To understand the problem closely, the graph of individual workers processing was analyzed.

The graphs 6.3 and 6.4, show when the first 10 workers began processing the jobs. The first 10 workers are shown due to visibility issues when showing 100 workers. Some of these graphs are shown in Appendix A. From the graph 6.3, it is seen that the workers begin to process at almost similar times. The red line that represents the biased worker is seen to be using a longer time to complete the job. The job from the last graph shows that workers 2-10 have finished processing and are waiting on worker 1 to complete the job. Worker 1 begins working only after almost 4 minutes, which suggests that the overall job completion time is slowed down. The performance metric $D_{t}^{Max}$, rises with every job showing that the job completion increase for each job. This is shown in the graph 6.5 below.

When the graph of $D_{t}^{95}$ is drawn together with $D_{t}^{Max}$, the line produced shows that the $D_{t}^{95}$ is always larger than $D_{t}^{Max}$, suggesting that a few of the workers are slow. This is shown in graph 6.6. This confirmed both the
CHAPTER 6. RESULTS III: EXPERIMENTATION & ANALYSIS

Figure 6.2: Graph depicting the effect of a slow worker on overall performance. As seen from the graph the red line ($T_{issued}$) gets farther away from the blue line ($T_{finished}$) as the jobs progress.

Figure 6.3: The first job processed. The graph shows that all workers begin at almost the same start time. The blue line shows the influenced worker.
6.3. INFLUENCED EXPERIMENTS

Figure 6.4: The last job processed. Compared to 6.3, the blue line shows that the worker starts much later than all other workers.

Figure 6.5: The graph of $D_i^{Max}$ vs job id. The increasing $D_i^{Max}$ shows that overall completion time increases.
CHAPTER 6. RESULTS III: EXPERIMENTATION & ANALYSIS

6.3.2 Effect of a duplicate slow worker after

How is the performance affected with a duplicate of the slowest worker?

The previous experiment was conducted to investigate, how the overall performance is affected with a slow biased worker. This experiment was conducted to find out what happens if a duplicate of the slowest worker is introduced after half the jobs are complete. An example of where this experiment is used in a typical MapReduce scenario is, after high values of $D_{\text{Max}}^t$ is registered. The goal of this is to find out the effect on time taken to complete the jobs and how $D_{\text{Max}}^t$ is affected.

The experiment will be conducted in a similar method to the previous one. The framework will be started with 99 workers. One worker was set to be the biased worker that slept 15 seconds longer than the others. 20 jobs with a 30 second query was started. This time the list of jobs being processed was watched carefully. After 10 minutes passed, a duplicate of the slowest worker was launched. The results that were seen immediately was a reduction in time during processing. The experiment completed in 12 minutes, and therefore had a performance increase of 2 minutes. The graph of job scheduling is shown below.

As seen from the graph 6.7, the blue line represents $T_{\text{issued}}$ and the red line represents $T_{\text{finished}}$. Once the duplicate worker is introduced little after 600 seconds the graph takes a turn and the other jobs are completed relatively quickers than jobs 12 and 13. This is seen from the parallel black
6.3. INFLUENCED EXPERIMENTS

Figure 6.7: Graph of job scheduling when a duplicate worker is introduced. The blue line shows the $T_{\text{issued}}$ and the red line $T_{\text{finished}}$. The dotted green line shows the predicted line without the duplicate worker. Compared to graph 6.2, the jobs finish much quicker.

The graph 6.8 of $D_{\text{Max}}^t$ vs job id, shows a steady increase of $D_{\text{Max}}^t$ as seen before. However after the duplicate worker is introduced at 10 minutes around the 15th job, the value of $D_{\text{Max}}^t$ reduces. As mentioned earlier, the $D_{\text{Max}}^t$ metrics represents the overall job scheduling. This experiment proved that a duplicate of the slowest worker can improve overall job performance.

The worker distribution graphs for this experiments is shown in Appendix A.

6.3.3 Understand job scheduling

How are jobs requests scheduled between 2 corpuses?

This experiment is to find out about how jobs requests are scheduled and processed between 2 corpuses. 2 distinct corpuses contain their own set of workers. An example of this in a MapReduce scenario is when there are multiple requests to different corpuses.

The experiment contained 100 workers belonging to corpus A and 100 workers belonging to corpus B. Once the framework was up, the jobs were sent to the task tracker in a pattern such as 2 jobs are sent to corpus B and one job to corpus A. The query of the job was set to 30 seconds. A such
loop was repeated 10 times making a total of 20 job requests to corpus B and 10 job requests to corpus A. It was noticed that the jobs belonging to corpus A was completed in 3 minutes and 30 seconds. The requests belonging to corpus B was completed in 7 minutes. During the job processing the progression of jobs was monitored closely. An interesting event was that the jobs belonging to one corpus was not affected by the jobs of the other corpus. All the jobs belonging to the same corpus was scheduled in a FIFO queue. The job distribution for the whole experiment is shown in the graph below 6.9.

6.3.4 Doubling cluster performance

What is the effect on performance when a cluster of workers is doubled?

This experiment is to find out how doubling a cluster of workers effects the scheduling in jobs. Booting up another cluster of workers can increase in energy costs used. Doubling of a cluster can seem as the viable choice at times when there is a long list of jobs that are scheduled.

The experiment was started with 100 workers. Once the framework was up the client was used to send 30 jobs, with a query of 20 seconds in succession to the task tracker. Once the jobs were sent, the job completion rate was closely monitored. When 15 jobs were done processing, the second cluster of workers were booted up and connected to the task tracker. The immediate effect that was noticed was that the task tracker showed that 2 jobs were being processed at the same time. The experiment was completed in 8 minutes, which is a clear improvement in performance compared to a single cluster. The graph below shows how job scheduling progressed.
6.3. INFLUENCED EXPERIMENTS

Figure 6.9: Job distribution between the 2 corpuses. Jobs belonging to one corpus is not affected by the other.

during the whole experiment.

As seen from the graph 6.10, the left side of the graph depicts the single cluster performance. Completing 15 jobs took almost 6 minutes. Once the cluster performance was doubled, the second 15 jobs were completed in almost 2 minutes. This has shown a performance gain in the order of 3. A gain of 3 means that long job queues can be processed faster by booting up another cluster. An interesting observation was made when comparing both the halves of the graph. The first half of the graph to the left has job processing times that are relatively shorter than the jobs completed in the second half to the right. This was verified by generating the graph of $D_{t}^{Max}$.

As predicted the graph of $D_{t}^{Max}$ 6.11, showed that after the cluster was doubled, job processing times have increased. Increased job processing times, increases $D_{t}^{Max}$ as well. Therefore this was ruled out to be a worker job scheduling problem that had caused the job to wait before the job was marked as done. To verify this the individual graphs of a job in the first and second half of the graph was compared.

As seen from the figure 6.12, the workers all overlap all other workers during processing of the same job at some point. The x axis of the graph is comparatively smaller. The figure 6.13, shows a much more distributed processing of the job between the worker groups. The second group from the top of the graph does not overlap the first, second and fourth worker groups from the bottom.

The graph 6.10 also shows that the first half of the graph had a maximum of 2 overlapping jobs in processing as compared to the second half which had almost 5 overlapping jobs in processing, which is thought as the reason to faster processing. This is observed by the black parallel
Figure 6.10: Double cluster performance. The orange box shows the progression of the jobs after the cluster is double. The blue line shows $T_{\text{issued}}$ and the red line shows $T_{\text{finished}}$. The orange box shows 15 jobs being completed faster.

Figure 6.11: The graph of $D_t^{\text{Max}}$ vs job id.
Figure 6.12: Worker distribution from a job in the first half.

Figure 6.13: Worker distribution from a job in the second half
6.4 Dispersed Test

One of the main aims of this thesis is to facilitate computations in data centers running on green energy. The problem statement mentions that computation will happen in a distributed and migratory manner. Green data centers are found around the world and to test if a computation was possible from around the world, a larger cloud service provider was used.

5 Amazon EC2 virtual machines were created in 5 different regions, with the assumption that these were regions where green energy was used. The task tracker was run on a virtual machine in Norway. Worker scripts were generated to be run on all these instances and copied to each of these machines. The free tier of Amazon only included virtual machines of the micro flavor with 1 \( V_{CPU} \) and 1 GB of \( V_{RAM} \). Therefore 5 worker instances was run per virtual machine which gave a total of 25 workers. The map 6.14 below shows where the worker nodes were placed.

Once the framework was running, 20 jobs with a query of 30 seconds was sent to the task tracker. The experiment took a little over 10 minutes to complete. Once the experiment was complete the statistics were gathered and the job distribution graph was drawn.

As seen from the graph 6.15, the job distribution is shown to have a very linear progression, despite the fact that the workers were distributed. An observation made is that a query of 30 seconds was completed in almost 45
6.4. DISPERSED TEST

Figure 6.15: Job distribution in a distributed worker environment

Figure 6.16: Job distribution in a local cluster
seconds. The same experiment was conducted in a local cluster. The graph 6.16 shows the results of the experiment performed in a local cluster. As seen from the graph the experiment completed in a shorter period than the distributed cluster, but only by a matter of seconds.

This experiment shows that there has been a difference of seconds when completing 20 jobs in a distributed and local cluster. If the assumption of the distributed workers using green energy is true, then there is only a 2% decrease in performance compared to a local cluster using non-renewable energy.

6.5 Summary

The chapter presented methods of how the task tracker was evaluated. This was done by first testing the functionality of the task tracker for the functions that are proposed in the design. The task tracker is then evaluated by performing biased experiments to find out how the task tracker reacts. A summary of the experiments performed is listed below.
### 6.5. SUMMARY

<table>
<thead>
<tr>
<th>Description</th>
<th>Data</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>100 workers 30 jobs 30 second query</td>
<td>Statistics from Task tracker and Client</td>
<td>Test job management</td>
</tr>
<tr>
<td>100 workers 1 reduce node 1 job</td>
<td>Statistics from Task tracker and Client</td>
<td>Test job whole MapReduce cycle</td>
</tr>
<tr>
<td>Access another user's job</td>
<td>Output from client</td>
<td>Can another user's job be accessed ?</td>
</tr>
<tr>
<td>Delete a job</td>
<td>Output from client</td>
<td>Can a job be deleted before and under processing</td>
</tr>
<tr>
<td>Cluster migration</td>
<td>Output from client</td>
<td>Job management with cluster migration</td>
</tr>
<tr>
<td>5000 jobs, 100 workers, map only</td>
<td>Output from client</td>
<td>Long term testing</td>
</tr>
<tr>
<td>99 workers, 1 slow worker, 20 jobs</td>
<td>Graph of $D^\text{Max}_t$ and Job progression</td>
<td>Effect on overall performance by slow worker</td>
</tr>
<tr>
<td>Same as previous, duplicate of slowest worker launched after 10 mins</td>
<td>Graph of $D^\text{Max}_t$ and Job progression</td>
<td>Effect of duplicate of slowest worker</td>
</tr>
<tr>
<td>2 corpus, 100 worker per corpus, 30 jobs</td>
<td>Graph of Job management</td>
<td>Understand job management between 2 corpuses</td>
</tr>
<tr>
<td>100 workers, 30 jobs, double cluster after half time</td>
<td>Graph of job distribution, $D^\text{Max}_t$ and worker distribution</td>
<td>Understand effect of a doubled cluster on performance</td>
</tr>
<tr>
<td>Dispersed Test with worker placed worldwide</td>
<td>Graph of Job distribution</td>
<td>Understand performance in a distributed scenario</td>
</tr>
</tbody>
</table>

Table 6.1: List of experiments performed
Chapter 7

Discussion

This chapter will discuss the different stages in the process of implementing the framework. The framework was designed by combining technologies that are mentioned in the background chapter. Alternative approaches were also constituted when designing the objective of this project, which will be presented and discussed. Finally the overall achievements and shortcoming of this project, potential improvements and future work will be presented.

7.1 The framework developed

The problem statement contained words such as task tracker, dispersed and MapReduce like. The implementation of the model features a fully functioning task tracker that supports a dispersed cluster of worker nodes and processes jobs in a MapReduce like style. The experiments that were conducted support this statement. These are conclusions that are also based on statistics that are gathered and analyzed, in understanding how the framework performs. A minor investigation was done into the feasibility of a dispersed framework, although this was done based on only one fully developed component and the others being just minimal prototypes. The current findings with the assumption of workers in green clouds, showed only a 2% decrease in performance compared to a local cluster which is assumed to be running on non-renewable energy.

7.2 Expected vs Real results

The experiments conducted on the framework revelead on how the task tracker performed in relation to the proposed function of the task tracker. The analysis conducted on the results of the experiments showed that the task tracker performed very well and most of the expected results were achieved. The performance metrics that was mentioned in the design seemed to match the behaviour of the task tracker. This made it certain that
the metrics extracted from the data was useful. The influenced experiments that were performed had produced over 12500 lines of statistics, but data here was extracted and put into forms of graphs which made it easier to analyse.

One of the unexpected results (see 6.2 Task 5), that would indicate a shortcoming down to the design section is handling workers that were migrated. The workers that were migrated during a job, had the job hanging in the processing status without it being finished. This had to be notified to the task tracker and the processing of the jobs which was handled by those workers could have been given to the migrated worker cluster. This experiment was originally done to understand how migrated workers performed. A simulation of migration was performed by stopping one cluster and starting another to test if jobs were diverted to the new cluster. It also depicted another scenario that was of a cluster of failed workers. Once the first cluster was stopped the workers were killed thereby losing the data under processing. Some of the shard id’s for the third job was set to issued and no result was received. This was the reason for the job not being completed. To prevent this, a timeout can be set for the particular shard before reissuing the job to a new worker.

When compared to a real life MapReduce scenario as mentioned in the background chapter, a typical cluster uses almost over 4000 nodes and the task tracker implemented was tested to only 5% of that capacity. Considering the fact that the task tracker runs on a single thread, the issues that the task tracker will face with 4000 nodes is a dive into the unknown. The experiment regarding the doubling of a cluster showed that once a cluster was doubled, job completion times increased. This was a increase from 100 to 200 worker nodes. A larger scaled scenario such as from 2000 to 4000 worker node job completion times are unknown. All this was also outside the range of what the cloud could handle and the resources that are provisioned, which was comparatively less.

A scenario of an experiment is a more controlled environment compared to a production environment where the task tracker is running. Therefore to avoid unexpected situations, stress testing the task tracker would be a viable choice to understand the boundaries of what the task tracker can handle and what can be improved upon.

7.3 Implementing the task tracker

The design was considered as one of the main parts of this thesis for a successful implementation, and therefore a lot of time was mostly used in perfecting the design and understanding wholly the functioning of the framework and its components. First the main goals that the task tracker had to perform was outlined and defined. The implementation phase took considerably lesser time, which was assisted by the rich libraries of Python which helped in omitting lines of code. The basic functions
that were required for the task tracker was developed and tested one by one. The worker and client were both developed in parallel. Later on the necessary functions were then designed and implemented, until all goals regarding the functioning of the task tracker was achieved. On the whole the components were designed in an iteration where the function was designed, implemented and tested. This method seemed to be motivating and helped in progressing with the development.

There were some cases where the required function was designed but not implemented. One noticeable flaw that was designed but not implemented was handling worker failures. Although the task tracker registers the last seen timestamp of each worker, nothing was done with this information thereby rendering it unutilized. This information could have been used to find out workers that had not reported within a time frame. Workers that did not respond could have been given a temporary inactive status so that a duplicate active worker can start the jobs. Metrics of this sort should be also available for the worker management component.

Considering the success of the task tracker, almost all designed functions that were implemented. The task tracker was designed in mind to handle all job related tasks, and this was handled with smoothness and ease by the task tracker. Communication protocol defined in JSON format performed with a 100% success rate, where no situations regarding the misunderstanding of JSON messages arised.

7.4 Modelling and designing

The design section presented a solution to the problem statement. There was research conducted previously in improving MapReduce frameworks such as Hadoop. This type of research involved mostly in smarter scheduling methods to conserve energy, but were all still assumed to a local cluster. All such approaches also spoke of modifying the existing Hadoop framework by writing wrapper code on top of it. A such approach was considered as the first alternative approach to this thesis. By developing a modification to an existing solution, a more focused solution on implementing a distributed worker environment could have been focused upon. This meaning that areas of job, data and worker management are already established areas and would have not been required to develop. A possible outcome of this approach would have been a Hadoop framework supporting distributed computing. However, this choice was ruled out due to the limited time frame available to potentially understand a complex system such as Hadoop.

The second alternative approach was built upon the decision to implement a framework from the ground up. It was well established that a communication protocol was required to communicate between the components. A fault in communication can cause data loss which then
affect the job completion times. By the use of formal logic and theory, a redundant communication protocol could have been created before the components are in place. The studies then would have resulted in finding all possible outcomes of a message that for input provided. By the use of formal notation the then developed communication protocol could have been proved to only result in a single outcome. This approach was ruled out due to the fact that the communication messages were designed and implemented during the whole development cycle of the task tracker and therefore was unknown. This approach can however be done, once the whole framework is developed to prove the reliability of the communication protocol.

Therefore the approach chosen was based on technologies mentioned in the background chapter; The Live migration which is very well known in the green computing environment, large distributed grids such as the CERN grid that is used for terabytes of computations and the MapReduce programming paradigm that is used for solving big data problems. Since the approach of modifying a Hadoop framework was ruled out, a decision was made to design a framework from the ground up by combining the technologies mentioned above. A such approach is not seen much in related work in solving big data problems. This choice was made mainly due to more freedom that was provided in designing and also the advantage of omitting unnecessary prototypes. This thesis was decided to only implement and design the task tracker, and therefore it was also necessary that it was designed to be future proof. The future work that is decided based on this is discussed later.

The model was designed with the intention of completing tasks, and not secondary goals such as security. In a real life scenario, security is one of the main aspects that should be considered when dealing with large amounts of data. There was a sense of security for the user of the system, by using usernames that decided which jobs that belonged to the user. This could have been faked easily, and therefore should have been thought about. This would have only required having a hash based dictionary structure in Python that stored the user details with a hashed password for security relating to the job details.

The goal of the framework is to compute both map and reduce computations. The map workers are released into the cloud by the worker management component and therefore is not the scope of this thesis. The experiments performed showed that 100 worker nodes with 20 reduce nodes performed well. This was mainly because the map function performed was created to mimick a function outputting data. In all the experiments the worker returned a random string with length of 100. It is possible that a typical map worker that returns 500 megabytes of data. When this is multiplied by upto 4000 the data that has to flow through the task tracker to a reduce node can pose a bottleneck in performance. The aim of developing a task tracker is mainly job handling and therefore 2
possible solutions can be possible to avoid this.

- A cluster of reduce nodes are appointed prior to the map cycle is started and the reduce workers information is passed along with the job details.

- Data is stored in a central storage cluster after a map cycle is performed. The reduce nodes then check in based on intervals to access a global reduce job list and perform the job.

As the project is presented here, the development cycle was much of a waterfall method where the design was first presented followed by the implementation of the framework. The waterfall method has known shortcomings such as the inability to incorporate new changes after implementing and limited time for testing due to a shorter time period. The task tracker had such shortcomings (such as the inability to handle lost workers) and could have been prevented if an agile development cycle was chosen. This would have then forced the task to get into an early stage of testing. A project of such nature would be more benefitted by an agile development cycle. The intended functionality could have been noted down and could have been designed, developed and tested one at a time.

7.5 Future work

This section will discuss the future developments of the framework and discuss issues of the current development.

7.5.1 The immediate future of the framework

As mentioned in the design chapter, the immediate future that is crucial for the framework to perform as intended, is to incorporate worker management. By worker management the goals should be to initiate a new cluster for a corpus, process statistics provided by the task tracker and create intelligent decisions that affect faster job processing. This is shown in the diagram 4.3.

Previous research in making MapReduce green is by implementing intelligent algorithms for scheduling. The project GreenHadoop [20], schedules jobs by predicting the availability of solar energy. Such advanced scheduling can be also incorporated to the framework to make use of green energy. The BEEMR project [14] had jobs in 2 categories. One labelled as interactive and the other batch. All interactive related jobs were marked as time sensitive and the latter as not. The same idea can be put into having a cluster that resides closely to the task tracker making it possible for more interactive jobs. The batch jobs can be put into the dispersed cluster of
workers that have more focus on using green energy than job completion times.

The background chapter mentioned the use of energy by idle virtual machines. Unikernal OS's combat this by using no CPU cycles and thereby consuming no energy when idle. A real-life MapReduce scenario can use worker nodes in the number of thousands and can consume energy when idle. By using unikernal os's this number can be reduced to a minimum thereby saving energy in a larger scale.

7.5.2 When will the framework shine?

The most appraising property of this framework is to perform MapReduce calculations in greener data centers. The intended method of functioning is to use live migration to migrate workers that can reside in greener clouds. The first challenge then would be to migrate workers to greener clouds. Mapreduce cycles can contain larger amounts of data that has to be transmitted. The second challenge would then be to investigate how distance between a task tracker and worker would affect the performance of job completion times. As an example a job query that requires around 30 seconds to complete, must not take up to an hour in a distributed worker environment. This would drastically reduce performance and decrease quality of service. Therefore migration into greener clouds can be performed in a smarter way, where there is an established balance between performance and energy use.

7.5.3 Usability issues of the current framework

The question of usability arised when the framework was setup to run the experiments. The whole framework is to be started up by command lines and a variety of parameters. This was particularly confusing at times. For example sending a job from the client required the server address, port number, user id, corpus name and query. A graphical interface would solve this problem where the user can input all necessary variables and send a query. The same can be done for all the other tasks such as viewing job results.

As mentioned in the background chapter, Google implemented MapReduce by copying the specific map function into the workers when the framework was initiated. In comparison here, the workers all get to perform the same map function that is return a random string of length 100. The main idea of how the framework is intended to perform is that workers are released into the cloud and are to be reused. Therefore the map function will also be provided along with the query. Providing a query can raise security issues considering dangerous scripts can be also provided as queries. This is very similar to a SQL injection attack which is common in the field of databases.
7.5. **FUTURE WORK**

The current situation of the task tracker is such that all data including a list of workers, jobs and users is stored locally. In case the task tracker stops responding, all data is lost. Another shortcoming with storing data locally is that the task tracker cannot be scaled and, techniques such as loadbalancing cannot be incorporated. Hadoop incorporates a master node per job for the purpose of maintenance and the same can be done here as well. To solve this shortcoming, data can be stored in both locally and on a global database. Backup and scalability would be offered if data is stored on a database. By storing data locally the framework eliminates unnecessary traffic between the database.

**7.5.4 Is the task tracker ready for future projects?**

The development of the task tracker lead to fulfilling the requirements of job management. This was tested by using the reference implementation of the workers, reducers and client. Communication with the task tracker is specified in the form of JSON objects. Therefore future projects can use the task tracker and its functions by transmitting JSON objects. As seen in the design chapter, the 2 distinct areas of management are the job and worker management. The worker management task of this framework then has to launch and manage workers. The statistics provided by the task tracker are slowest worker, fastest worker, total time used for job completion, 95th percentile worker completion time and how synced all workers are in relation to processing the job. These statistics have been proven to be useful in deciding when to spawn new workers for efficiently completing the job, and therefore is sufficient to implement worker management.

During development of the framework, the task tracker may require modification to include new features. The task tracker was designed and implemented so that functions were broken down into smaller functions to decouple as much as possible. Implementing new functions can use these smaller functions to integrate with the new function to be developed.
Chapter 8

Conclusion

This project began with the aim of investigating methods into reducing energy usage in MapReduce computations. This was due to the predicted exponential increase in energy requirements to powering data centers. This posed grave threats to the environment in the long-term.

This problem was determined to be answered by investigating the feasiliblity of a dispersed and dynamic MapReduce cluster. It was approached first by designing the framework that is expected to solve the problem from scratch. A prototype of all the components were built with main focus residing on a fully developed task tracker.

The experiments and analysis prove the possiblity of a possible dispersed framework. The current stage of development showed a 2% decrease in performance compared to a local cluster. The main acheivements of this project are :

• Computing MapReduce jobs with a dispersed cluster of workers placed around the world.
• Providing performance metrics that can be used in future implement-ation of worker management.

These results provide both hope and a firm ground for future development of the framework until the ultimate goal of a dispersed framework is reached. A framework designed from the ground up can always be improved and provides countless paths of future work. Some of the possible paths found were:

• Implementing a automatic system for worker management that can regulate the workers based on requirements
• The creation and migration of lean worker nodes to greener clouds
• Refining the fine balance between green energy consumption and job completion times
An article about the proposed framework is planned to be submitted to the International Journal of Big Data Intelligence.
Bibliography


[26] Jonathan Koomey and Jon Taylor. ‘New data supports finding that 30 percent of servers are ‘Comatose’, indicating that nearly a third of capital in enterprise data centers is wasted’. In: June 3 (2015), p. 2015.


[29] James Manyika et al. ‘Big data: The next frontier for innovation, competition, and productivity’. In: (2011).


Chapter 9

Appendix A - Graphs

9.1 Worker distribution for slowed worker experiment
9.2 Worker distribution for duplicated worker experiment
Chapter 10

Appendix B - server.py

```python
#!/usr/bin/env python
from __future__ import division
import socket
import json
import argparse
import time
import sys
import random
import string
import copy
import numpy as np

# Argument parsing
parser = argparse.ArgumentParser(prog='server.py')
parser.add_argument('−v', '−−verbose', dest='verbose', help='Turn verbosity on', default=False, action='store_true')
parser.add_argument('−d', '−−debug', dest='debug', help='Turn debug messages on', default=False, action='store_true')
parser.add_argument('−p', '−−port', dest='port', type=int, help='What port to use connect to on the server', metavar='N', default=5005)
arguments = parser.parse_args()
VERBOSE = arguments.verbose
DEBUG = arguments.debug
PORT = int(arguments.port)

# Other global variables
i = 0

WORKER_LIST = dict() # List of all workers
USER_LIST = dict() # User list maintaining job list for each user
JOB_LIST = dict() # Universal Job list
REDUCER_LIST = dict() # Universal reducer list
STAT_TREE = dict() # CID, SHID, JID list
STATISTICS_TREE = dict() # Recording statistics
CORPUS_LIST = dict() # List of corpus and jobs according to it
```

125
CHAPTER 10. APPENDIX B - SERVER.PY

REDUCE_QUEUE = []

ID_LENGTH = 10

WORKER_HEALTH_INTERVAL = 300

# #########################################
def verbose(text):
    if VERBOSE:
        print("\nVERBOSE: " + text + "\n")

def debug(text):
    if DEBUG:
        print("VERBOSE: " + text)

def process_new_job(client, data_json):
    global i
    reply = {
        "type": "job",
        "jid": get_random_id(), # the JID that the job got
        "query": data_json["query"],
        "cid": data_json["cid"],
        "time_created": data_json["time_created"],
        "user": data_json["user"], # the user issuing the job
        "status": "Queued" # The following states: queued, started, finished
    }
    i += 1 # increase job nr

    if "reduce_script" in data_json:
        reply["reduce_script"] = data_json["reduce_script"]

    store_job(reply)
    client.send(json.dumps(reply))
    client.close()
    return

def get_job_status(client, data_json):
    # Based from a user id and the job. MUST define status
    uid = data_json["user"]
    jid = data_json["jid"]

    if data_json["jid"] in JOB_LIST:
        if uid == JOB_LIST[jid]["user"]: verbose("Returning JSON" + json.dumps("job")))
            client.send(json.dumps(JOB_LIST[data_json["jid"]]))
    client.close()
    return
```python
def list_jobs(client, data_json):
    debug("list_jobs called")

    data_json["job_list"] = []
    if data_json["user"] in USER_LIST:
        for jid in USER_LIST[data_json["user"]]:
            job = copy.deepcopy(JOB_LIST[jid])
            # we remove the results from the job in
            order to save space
            job["results"] = dict()
            data_json["job_list"].append(job)

    verbose("Returning: " + json.dumps(data_json))
    client.send(json.dumps(data_json))

def handle_hello(client, hello, ip):
    debug("handle_hello() called")

    if "wid" in hello:
        verbose("This is a worker hello")
        handle_worker_hello(client, hello, ip)
    elif "rid" in hello:
        verbose("This is a reducer hello")
        handle_reducer_hello(client, hello, ip)

def handle_reducer_hello(client, hello, ip):
    debug("handle_reducer_hello() called")

    # check if worker exists in WORKER_LIST
    if hello["rid"] in REDUCER_LIST:
        REDUCER_LIST[hello["rid"]]["last_seen"] = str(int(time.time()))
        REDUCER_LIST[hello["rid"]]["ip_address"] = ip
    else:
        # create new entry in WORKER_LIST
        REDUCER_LIST[hello["rid"]]["map_result"] = JOB_LIST[job["jid"]]["results"]
        REDUCER_LIST[job["jid"]]["reduce_result"] = job["reduce_time_started"]
        JOB_LIST[job["jid"]]["reduce_time_finished"] = job["reduce_time_finished"]
        JOB_LIST[job["jid"]]["reduce_time_started"] = job["reduce_time_started"]
        STATISTICS_TREE[job["cid"]]["job_list"][job["jid"]]["TrStart"] = job["reduce_time_started"]
        STATISTICS_TREE[job["cid"]]["job_list"][job["jid"]]["TrStop"] = job["reduce_time_finished"]
```
decr\-

   "type" : "decr\-
            "job_list" : [], # relative to shid
   "worked_on" : [] # JIDs of jobs being worked on (only relevant when multiple bees on the same shid
   }

if len(RE\n
   # Check if the reducer is corpus specific
   print REDUCE_QUEUE
   if hello["cid"] == "None": # If not pick the first job in the queue
      REDUCE_QUEUE[0]).append(JOB_LIST[REDUCE_QUEUE[0]]
   else: # Pick the first job that can be used on
         REDUCE_QUEUE.remove(RE\n
   for job_id in REDUCE_QUEUE:
      if hello["cid"] == JOB_LIST[job_id]
         REDUCE_QUEUE.remove(
             JOB_LIST[job_id]["jid"]
         break

   verbose("Returning: " + json.dumps(decr\-
   client.send(json.dumps(decr\-

   def handle_worker_hello(client, hello, ip):
      debug("handle_worker_hello() called")

      # check if worker or reducer

      # check if worker exists in WORKER_LIST
      if hello["wid"] in WORKER_LIST:
         WORKER_LIST[hello["wid"]["last_seen"] = str(int(time.time()))
         WORKER_LIST[hello["wid"]["ip_address"] = ip
      else:
         # create new entry in WORKER_LIST
         WORKER_LIST[hello["wid"]["cid"] = dict()
         WORKER_LIST[hello["wid"]["ip_address"] = ip
         WORKER_LIST[hello["wid"]["last_seen"] = str(int(time.time()))
         WORKER_LIST[hello["wid"]["cid"] = hello["cid"]
         WORKER_LIST[hello["wid"]["shid"] = hello["shid"]

      # create new entry in STAT_TREE
      if hello["cid"] in STAT_TREE:
         if not hello["shid"] in STAT_TREE[hello["cid"]]:
            STAT_TREE[hello["cid"]][hello["shid"]] = []
      else:
         STAT_TREE[hello["cid"]] = dict()}
129

189

STAT_TREE[hello["cid"]][hello["shid"]] = 

# Create the cid in the statistics tree as well

190

STATISTICS_TREE[hello["cid"]]["job_list"] = dict()

191

STATISTICS_TREE[hello["cid"]]["cid"] = = dict()

192

hello["cid"]

193

STATISTICS_TREE[hello["cid"]]["job_list"]

194

CORPUS_LIST[hello["cid"]]["job_list"] = []

# Create the cid in the statistics tree as well

195

CORPUS_LIST[hello["cid"]]["job_list"] = []

# go through finished jobs

198

verbose("Got hello: " + json.dumps(hello))

199

for job in hello["result_list"]:  

print "handling finished job: " + json.dumps(job)

200

print "handling finished job: " + json.dumps(job)

201

if job["jid"] not in JOB_LIST:

202

break

203

# update jid results list with result from job

204

if not hello["shid"] in JOB_LIST[job["jid"]]["results"]:  

205

# reduce shid_remain with

206

JOB_LIST[job["jid"]]["results"][hello["shid"]] = job["result"]

207

JOB_LIST[job["jid"]]["shid_remain"] -= 1

208

JOB_LIST[job["jid"]]["shid_list"] = 2

209

# Register shid id with time used = time finished - time started

210

STATISTICS_TREE[job["cid"]]["job_list"][job["jid"]]["shid_list"][hello["shid"]]["time_used"] = int(job["time_finiished"]) - int(job["time_started"])  

211

STATISTICS_TREE[job["cid"]]["job_list"][job["jid"]]["shid_list"][hello["shid"]]["time_finiished"] = int(job["time_finiished"])  

212

STATISTICS_TREE[job["cid"]]["job_list"][job["jid"]]["shid_list_used"][hello["shid"]] = int(job["time_finiished"]) - int(job["time_started"])  

213

# see if shid_remain is 0

214

if JOB_LIST[job["jid"]]["shid_remain"] == 0:

215

# if shid_remain 0: update

216

if "reduce_script" in JOB_LIST[job["jid"]]  

217

REDUCE_QUEUE.append(job["jid"])
status" = "map_finished"

    JOB_LIST[job["jid"]]["status"] = "map_finished"

    else:
        JOB_LIST[job["jid"]]["status"] = "finished"

    timeFinished" = str(int(time.time()))
    # fill out time taken, adn overhead

    STATISTICS_TREE[job["cid"]]["job_list"]
    "job_list"[job["jid"]]["Tt"] = int(job["time_finished"]) - int(job["time_created"])
    STATISTICS_TREE[job["cid"]]["job_list"]
    "job_list"[job["jid"]]["Tadm"] = int(job["time_issued"]) - int(job["time_created"])
    STATISTICS_TREE[job["cid"]]["job_list"]
    "job_list"[job["jid"]]["Tstart_stop"] = int(job["timeFinished"])
    "job_list"[job["jid"]]["time_issued"] = int(job["time_created"]) - int(job["time_issued"])

    calculate_stats(job["cid"], job["jid"])

    # Check if reduce is necessary
    if JOB_LIST[job["jid"]]["status"] != "deleted":
        remove(job["jid"])

    decree = {
        "type": "decrce",
        "job_list": [], # relative to shid
        "worked_on": [] # JIDs of jobs being worked on (only relevant when multiple bees on the same shid)
    }

    # get a list of new jobs
    if hello["cid"] in STAT_TREE:
        for jid in STAT_TREE[hello["cid"]][hello["shid"]]:
            # Do not hand out duplicates
            if JOB_LIST[jid]["shid_list"][hello["shid"]]["jid"]:
                JOB_LIST[jid]["status"] = "issued"
                if not "time_issued" in JOB_LIST[jid]:
                    JOB_LIST[jid]["time_issued"] = str(int(time.time()))
                    mod_job = copy.deepcopy(JOB_LIST[jid])
                    del mod_job["results"]
                    del mod_job["shid_list"]
                    decree["job_list"]["shid_list"]["job_list"]["jid"] = 1
                    break

    # decree["job_list"]["shid_list"]
    "job_list".append(fakejob1)
    verbose("Returning: " + json.dumps(decree))
    client.send(json.dumps(decree))
```python
def store_job(job_json):
    debug("Storing job in job_list")
    # Create array for shid list according to corpus

    # Update STAT_TREE with job
    if not job_json["cid"] in STAT_TREE:
        # No corpus of that name yet, job not submitted
        return False

    workers = dict()
    for shid in STAT_TREE[job_json["cid"]]:
        STAT_TREE[job_json["cid"]][shid].append(job_json["jid"])
        workers[shid] = 0

    # Add the array created to the dict object
    job_json["shid_list"] = workers
    job_json["shid_remain"] = len(workers)
    job_json["results"] = dict()
    verbose("Storing job: " + json.dumps(job_json))

    if not job_json["user"] in USER_LIST:
        USER_LIST[job_json["user"]] = []
        USER_LIST[job_json["user"]].append(job_json["jid"])

    # Store job in Statistics Tree
    STATISTICS_TREE[job_json["cid"]]["job_list"] = dict()
    STATISTICS_TREE[job_json["cid"]]["job_list"] = dict()
    STATISTICS_TREE[job_json["cid"]]["job_list"] = dict()
    CORPUS_LIST[job_json["cid"]]["job_list"].append(job_json["jid"])

    if len(CORPUS_LIST[job_json["cid"]]["job_list"]) == 1:
        CORPUS_LIST[job_json["cid"]]["first_job"] =
        job_json["time_created"]
    else:
        CORPUS_LIST[job_json["cid"]]["last_job"] =
        job_json["time_created"]
        td = int(CORPUS_LIST[job_json["cid"]]["last_job"])
        - int(CORPUS_LIST[job_json["cid"]]["first_job"])
        if td > 0:
            CORPUS_LIST[job_json["cid"]]["arrival_rate"] =
            len(CORPUS_LIST[job_json["cid"]]["job_list"]) / td
        else:
            CORPUS_LIST[job_json["cid"]]["arrival_rate"] =
            len(CORPUS_LIST[job_json["cid"]]["job_list"])

    print CORPUS_LIST

def delete_job(job_json):
    # Delete job from list
```

debug("delete_job called")

# Find out if job is users
if not job_json["uid"] in USER_LIST:
    return False

if not job_json["jid"] in USER_LIST[job_json["uid"]]:
    return False

verbose("Deleting job with id: " + job_json["jid"])
job = JOB_LIST[job_json["jid"]]

# Remove the job from all the shids in the STAT_TREE
for shid in STAT_TREE[job["cid"]]:
    STAT_TREE[job["cid"]][shid].remove(job_json["jid"])

# Set status of job on job list as deleted
JOB_LIST[job_json["jid"]]["status"] = "deleted"
verbose("Deleted job ")

def get_random_id():
    debug("get_random_id called")
    return ".join(random.choice(string.ascii_uppercase +
string.digits) for _ in range(ID_LENGTH))

def get_worker_list(client, wlist):  # List all workers
    debug("get_worker_list called")
    wlist["worker_list"] = []
    for wid in WORKER_LIST:
        worker = {
            "wid" : wid,
            "ip_address" : WORKER_LIST[wid]["ip_address"],
            "shid" : WORKER_LIST[wid]["shid"],
            "cid" : WORKER_LIST[wid]["cid"],
            "last_seen" : WORKER_LIST[wid]["last_seen"],
        },
        wlist["worker_list"].append(worker)

    verbose("returning: " + json.dumps(wlist))
    client.send(json.dumps(wlist))

def get_reducer_list(client, rlist):
    debug("get_reducer_list called")
    rlist["reducer_list"] = []
    for rid in REDUCER_LIST:
        reducer = {
            "rid" : rid,
            "ip_address" : REDUCER_LIST[rid]["ip_address"],
            "cid" : REDUCER_LIST[rid]["cid"],
        },
        rlist["reducer_list"].append(reducer)
"last_seen" : REDUCER_LIST[rid]["last_seen
"],
        ]
    rlist["reducer_list"].append(reducer)
    verbose("returning: " + json.dumps(rlist))
    client.send(json.dumps(rlist))
def calculate_stats(cid, jid):
    debug("calculating_statistics for job done")
    Twmin = -1
    Twmax = -1
    for shid, tid in STATISTICS_TREE[cid]["job_list"][jid]["shid_list_used"].items():
        if Twmin == -1 and Twmax == -1:
            Twmin = [shid, tid]
            Twmax = [shid, tid]
        if tid < Twmin[1]:
            Twmin = [shid, tid]
        if tid > Twmax[1]:
            Twmax = [shid, tid]

    STATISTICS_TREE[cid]["job_list"][jid]["Tw95"] = str(int(np.percentile(STATISTICS_TREE[cid]["job_list"][jid]["shid_list_used"].values(), 95)))
    STATISTICS_TREE[cid]["job_list"][jid]["TwMax"] = Twmax
    STATISTICS_TREE[cid]["job_list"][jid]["TwMin"] = Twmin
    STATISTICS_TREE[cid]["job_list"][jid]["DtMax"] = int(STATISTICS_TREE[cid]["job_list"][jid]["Tt"] - Twmax[1])
    STATISTICS_TREE[cid]["job_list"][jid]["Dt95"] = int(STATISTICS_TREE[cid]["job_list"][jid]["Tt"] - int(STATISTICS_TREE[cid]["job_list"][jid]["Tw95"]))
    verbose("Calculated Stats : " + json.dumps(STATISTICS_TREE[cid]["job_list"][jid]))

def get_stats(client, job_json):
    debug("get stats for a corpus")
    cid = job_json["corpus"]
    if cid not in STAT_TREE:
        data_json["result"] = "No corpus found"
    else:
        data_json["job_list"] = []

    for i in STATISTICS_TREE[cid]["job_list"][jid]["job_list"][i]):
        job = copy.deepcopy(STATISTICS_TREE[cid]["job_list"][i])
        job["jid"] = i
        data_json["job_list"][i].append(job)
    verbose("returning: " + json.dumps(data_json))
    client.send(json.dumps(data_json))
```python
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
TCP_IP = "0.0.0.0"
TCP_PORT = PORT
BUFFER_SIZE = 8024
s.bind((TCP_IP, TCP_PORT))
s.listen(5)

ip_add_reg = [] # Register of id; ip address; data string received

while 1:
    conn, addr = s.accept()
    print('Connection address:', addr)
    data = conn.recv(BUFFER_SIZE)
data_json = json.loads(data)
    verbose("Got JSON: " + str(json.dumps(data_json)))
ip_add_reg.append([i, addr[0], data])
    # Inspect the JSON object to see what it is and send it to the appropriate
    # method to handle it

    # a new job from a client
    if (data_json["type"] == "job" and "jid" not in data_json):
        process_new_job(conn, data_json)
        conn.close()

    # status query about existing job from a client
    if (data_json["type"] == "job" and "jid" in data_json):
        get_job_status(conn, data_json)
        conn.close()

    # list request from a client
    if (data_json["type"] == "list"):
        list_jobs(conn, data_json)
        conn.close()

    # hello from a worker or reducer
    if (data_json["type"] == "hello"):
        handle_hello(conn, data_json, addr[0])
        conn.close()

    # request worker list
    if (data_json["type"] == "worker_list"):
        get_worker_list(conn, data_json)
        conn.close()

    # request worker list
    if (data_json["type"] == "reducer_list"):
        get_reducer_list(conn, data_json)
```

conn.close()

# delete job
if (data_json["type"] == "delete_job"):
    delete_job(data_json)
    conn.close()

if (data_json["type"] == "stats_request"):
    get_stats(conn, data_json)
    conn.close()

# nothing else fits, let's close the connection
conn.close()
Chapter 11

Appendix C - worker.py

#!/usr/bin/env python
import socket
import json
import argparse
import time
import sys
import random
import string

## Global / Default variables
TCP_PORT = 5005
BUFFER_SIZE = 8024

# Argument parsing
parser = argparse.ArgumentParser(prog='worker.py')
parser.add_argument('-v', '--verbose', dest='verbose', help='Turn verbosity on', default=False, action='store_true')
parser.add_argument('-d', '--debug', dest='debug', help='Turn debug messages on', default=False, action='store_true')
parser.add_argument('-s', '--server', dest='server', help='IP address of server to connect to')
parser.add_argument('-p', '--port', dest='port', type=int, help='What port to use connect to on the server', metavar='N', default=TCP_PORT)
parser.add_argument('-i', '--interval', dest='interval', type=int, help='Interval to connect to queen', default=60)
parser.add_argument('-c', '--corpus', dest='corpus', help='What corpus to use')
parser.add_argument('-S', '--shid', dest='shid', help='What shard ID to use')
parser.add_argument('-w', '--worker-id', dest='wid', help='Worker ID to use')
parser.add_argument('-W', '--wait-time', dest='wait', help='Wait time for execution—Testing purposes only', default=0)
arguments = parser.parse_args()

VERBOSE = arguments.verbose
DEBUG = arguments.debug
SERVER = str(arguments.server)
PORT = int(arguments.port)
INTERVAL = int(arguments.interval)
CORPUS = str(arguments.corpus)
SHID = str(arguments.shid)
WID = str(arguments.wid)
WAIT = int(arguments.wait)

# Other global variables
WORKING_ON = None
STARTED_WORKING = 0
FINISHED_LIST = []

if WID == "None":
    WID = ''.join(random.choice(string.ascii_uppercase + stringdigits) for _ in range(10))

def verbose(text):
    if VERBOSE:
        print("\nVERBOSE: " + text + "\n")

def debug(text):
    if DEBUG:
        print("VERBOSE: " + text)

def connect_to_queen():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((SERVER, PORT))
    return s

def start_work(job):
    global WORKING_ON
    global STARTED_WORKING
    WORKING_ON = job
    WORKING_ON["time_started"] = str(int(time.time()))
    WORKING_ON["status"] = "started"
    WORKING_ON["result"] = map_function(WORKING_ON["query"])
    WORKING_ON["time_finished"] = str(int(time.time()))
    WORKING_ON["status"] = "finished"
    FINISHED_LIST.append(WORKING_ON)

# the actual mapping function
def map_function(query):
    if ":" in query:
        arglist = query.split(";")
        waitlist = arglist[1].split(";")
        if int(SHID) == int(waitlist[0]):
            verbose("extra sleep time selected")
```python
time.sleep(int(waitlist[1]))
else:
time.sleep(int(arglist[0]))
else:
time.sleep(int(query))
time.sleep(WAIT)
return ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in range(100))

while True:
    debug("New cycle starting")
    # create hello message
    hello = {
        "type": "hello",
        "wid": WID,
        "cid": CORPUS,  # the corpus ID / string?
        "shid": SHID,  # some number?
        "working_on": "jid",  # the actual job being processed
        "result_list": []  # the results of compelted jobs
    }
    for job in FINISHED_LIST:
        hello["result_list"].append(job)
    try:
        queen = connect_to_queen()
    except:
        print("No conenction to queen. Sleeping 30s")
        time.sleep(30)
        continue
    verbose("Sending: " + str(json.dumps(hello)))
    queen.send(str(json.dumps(hello)))
    # get decree back
    reply = queen.recv(BUFFER_SIZE)
    verbose("got decree back: " + reply)
    queen.close()
    decree = json.loads(reply)
    FINISHED_LIST = []
    now = int(time.time())
    # put jobs on internal list of jobs
    for job in decree["job_list"]:
        # do one job
        start_work(job)
        break
    finished = int(time.time())
```
# sleep for a period
sleep_time = INTERVAL - (finished - now)

if sleep_time < 0:
sleep_time = 0

time.sleep(sleep_time)
# Chapter 12

## Appendix D - reducer.py

```python
#!/usr/bin/env python
import socket
import json
import argparse
import time
import sys
import random
import string
import shutil
import os
import subprocess

## Global / Default variables
TCP_PORT = 5005
BUFFER_SIZE = 8024

### Argument parsing

parser = argparse.ArgumentParser(prog='reducer.py')
parser.add_argument('-v', '--verbose', help='Turn verbosity on', default=False, action='store_true')
parser.add_argument('-d', '--debug', help='Turn debug messages on', default=False, action='store_true')
parser.add_argument('-s', '--server', help='IP address of server to connect to')
parser.add_argument('-p', '--port', help='What port to use connect to on the server', type=int, metavar='N', default=TCP_PORT)
parser.add_argument('-i', '--interval', help='Interval to connect to queen', default=60)
parser.add_argument('-c', '--corpus', help='What corpus to use')
parser.add_argument('-r', '--reducer-id', dest='rid', help='Reducer ID to use')
parser.add_argument('-W', '--wait-time', dest='wait', help='Wait time for execution - Testing purposes only', default=0)
```
CHAPTER 12. APPENDIX D - REDUCER.PY

32 parser.add_argument('−P', '−workdir−path', dest='workdir_path', help=
                          'Directory to use as base for storing data and code',
                          default='/tmp')
33 arguments = parser.parse_args()
34
35 VERBOSE = arguments.verbose
36 DEBUG = arguments.debug
37 SERVER = str(arguments.server)
38 PORT = int(arguments.port)
39 INTERVAL = int(arguments.interval)
40 CORPUS = str(arguments.corpus)
41 RID = str(arguments.rid)
42 PATH = str(arguments.workdir_path)
43 WAIT = int(arguments.wait)
44
45 WORKING_ON = None
46 STARTED_WORKING = 0
47 FINISHED_LIST = []

48 if RID == "None":
49    RID = ''.join(random.choice(string.ascii_uppercase +
                                  string.digits) for _ in range(10))

49
def verbose(text):
50    if VERBOSE:
51        print("\nVERBOSE: " + text + "\n")
52
def debug(text):
53    if DEBUG:
54        print("VERBOSE: " + text)
55
def connect_to_queen():
56    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
57    s.connect((SERVER, PORT))
58    return s
59
def start_work(job):
60    global WORKING_ON
61    global STARTED_WORKING
62
63    WORKING_ON = job
64
65    WORKING_ON["reduce_time_started"] = str(int(time.time()))
66    WORKING_ON["reduce_status"] = "started"
67
68    WORKING_ON["reduce_result"] = reduce_function(WORKING_ON["reduce_script"], WORKING_ON["results"])
69
70    del WORKING_ON["results"]
71    WORKING_ON["reduce_timeFinished"] = str(int(time.time()))
72    WORKING_ON["reduce_status"] = "finished"
73
74    WORKING_ON["status"] = "finished"
verbose("Finished job: " + str(WORKING_ON) )
FINISHED_LIST.append(WORKING_ON)

# the actual mapping function
def reduce_function(reduce_function, results):
    # 1. get random number
    workdir = PATH + "/" + ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in range(10))
    verbose("New workdir: " + workdir )

    # 2. create folder for work
    os.makedirs(workdir)
    os.chdir(workdir)

    # 3. dump result into JSON file
    file = open("result.json","w")
    file.write(str(json.dumps(results)))
    file.close()

    # 4. dump script to file
    file = open("script","w")
    file.write(reduce_function)
    file.close()
    os.chmod("script",0700)

    # 5. execute script on result file
    path = workdir + "/script result.json"
    output = subprocess.check_output(path,shell=True, executable=’/bin/bash ’)
    print output

    # 6. delete folder
    shutil.rmtree(workdir)

    # 7. return result
    return output

def long_read(conn):
    total_data=[]
    while True:
        data = conn.recv(8192)
        if not data: break
        total_data.append(data)
    return ' '.join(total_data)

while True:
    debug("New cycle starting")

    # create hello message
    hello = {
        "type" : "hello",
        "rid" : RID,
        "cid" : CORPUS, # the corpus ID / string?
        "working_on" : "jid", # the actual job being processed
        }
"result_list": [], # the results of completed jobs

for job in FINISHED_LIST:
    hello["result_list"].append(job)

try:
    queen = connect_to_queen()
except:
    print "No connection to queen. Sleeping 30s"
    time.sleep(30)
    continue

verbose("Sending: " + str(json.dumps(hello)))
queen.send(str(json.dumps(hello)))

# get decree back
reply = long_read(queen)
verbose("got decree back: " + reply)
queen.close()

decree = json.loads(reply)
FINISHED_LIST = []

now = int(time.time())
# put jobs on internal list of jobs
for job in decree["job_list"]:
    # do one job
    start_work(job)
    break

finished = int(time.time())

# sleep for a period
sleep_time = INTERVAL - (finished - now)
if sleep_time < 0:
    sleep_time = 0

    time.sleep(sleep_time)
Chapter 13

Appendix E - client.py

#!/usr/bin/env python
import socket
import json
import argparse
import time
import sys
import math

### Global / Default variables
TCP_PORT = 5005
BUFFER_SIZE = 8024
TCP_IP = '0.0.0.0'

# Argument parsing
parser = argparse.ArgumentParser(prog='hiveclient.py')
parser.add_argument('--v', '--verbose', dest='verbose', help='Turn
verbosity on', default=False, action='store_true')
parser.add_argument('--d', '--debug', dest='debug', help='Turn
debug messages on', default=False, action='store_true')
parser.add_argument('--s', '--server', dest='server', help='IP
address of server to connect to')
parser.add_argument('--p', '--port', dest='port', type=int,
help='What
port to use connect to on the server', metavar='N',
default=TCP_PORT)
parser.add_argument('--c', '--corpus', dest='corpus', help='What
corpus to use')
parser.add_argument('--l', '--list', dest='list', help='List executing
jobs', default=False, action='store_true')
parser.add_argument('--u', '--user', dest='user', help='The user to
identify with to the server')
parser.add_argument('--i', '--id', dest='id', help='Id of query')
parser.add_argument('--q', '--query', dest='query', help='The query to
eexecute on the corpus')
parser.add_argument('--w', '--wait', dest='wait', help='The query to
execute on the corpus', default=False, action='store_true')
parser.add_argument('--x', '--extended', dest='extended', help='Print
extended info about the state of individual shards when...
printing job info", default=False, action="store_true")
parser.add_argument('−a', '−−all−results', dest="allr", help="Print
all map results", default=False, action="store_true")
parser.add_argument('−W', '−−worker−list', dest="worker_list", help="List the available workers", default=False, action="store_true")
parser.add_argument('−D', '−−delete', dest="delete", help="Deleting a
job based on the job id", default=False, action="store_true")
parser.add_argument('−r', '−−reduce', dest="reduc", help="Specify the
reduce function as a script", default="")
parser.add_argument('−R', '−−reducer−list', dest="reducer_list", help="List the available reducers", default=False, action="store_true")

arguments = parser.parse_args()
VERBOSE = arguments.verbose
DEBUG = arguments.debug
SERVER = str(arguments.server)
PORT = int(arguments.port)
CORPUS = str(arguments.corpus)
LIST = arguments.list
USER = str(arguments.user)
ID = str(arguments.id)
QUERY = str(arguments.query)
WAIT = arguments.wait
WORKERS = arguments.worker_list
REDUCERS = arguments.reducer_list
EXTENDED = arguments.extended
ALL_RESULTS = arguments.allr
DELETE = arguments.delete
REDUCE = str(arguments.reduc)

# verbose

def verbose(text):
    if VERBOSE:
        print("\nVERBOSE: " + text + "\n")

def debug(text):
    if DEBUG:
        print("VERBOSE: " + text

def connect_to_queen():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((SERVER,PORT))
    return s

def send_query(user, corpus, query, reduc):
    debug("send_query(" + user + "," + corpus + "," + query + "," + reduc + ")")
    # transform the query to a job in JSON
    job = [
        "type": "job",
        "cid": corpus,
        "query": query,
        "time_created": str(int(time.time())),
        "user": user
    ]
if reduc != "":
    rf = open(reduc, "r")
    job["reduce_script"] = rf.read()

# j_job = json.loads(job)
verbose(str(json.dumps(job)))

# open connection to queen bee
s = connect_to_queen()

# submit job
s.send(json.dumps(job))

# get ID from returned job
data = s.recv(BUFFER_SIZE)

# print ID back to user
s.close()
verbose("received data: " + data)
reply = json.loads(data)

print reply["jid"]

def list_jobs(user):
    debug("list_jobs(" + user + ")")
    list_request = {
        "type": "list",
        "user": user
    }

    s = connect_to_queen()
    s.send(json.dumps(list_request))
    reply = long_read(s)
    s.close()
    verbose("Recieved data: " + reply)
    list_reply = json.loads(reply)

    # pretty printing
    header_list = ["JOB", "CORPUS", "QUERY", "CREATED", "STATUS", "TIME_USED"]

    row_format = "{:<25} " * (len(header_list))
    print row_format.format(*header_list)
    for job in list_reply["job_list"]: 
        now = int(time.time())
        if job["status"] == "deleted":
            stat_text = "Job Deleted"
        elif job["status"] == "map_finished":
            stat_text = "Map Finished"
        else:
            stat_text = create_shid_stat_text(job["shid_list"]) 
        time_used_text = "" 
        if "time_issued" in job:
time_used_text += "issued " + create_time_since_text(now, int(job["time_issued"])) + " ago"

if "time_finished" in job and job["status"] == "finished":
    time_used_text += ", finished " + create_time_since_text(now, int(job["time_finished"])) + " ago"
    time_used_text += ", time_used " + create_time_since_text(int(job["time_issued"]), int(job["time_issued"])))

elif "time_finished" in job and job["status"] == "map_finished":
    time_used_text += ", map_finished " + create_time_since_text(now, int(job["time_finished"])) + " ago"
    time_used_text += ", time_used " + create_time_since_text(int(job["time_issued"]), int(job["time_issued"])))

row = [job["jid"], job["cid"], job["query"],
create_time_since_text(now, int(job["time_created"])) + " ago",
stat_text, time_used_text]
print row_format.format(*row)

def create_time_since_text(now, then):
    time_diff = int(now - int(then))
    m, s = divmod(time_diff, 60)
    h, m = divmod(m, 60)
    last_seen = ""
    if h > 0:
        last_seen = "%dh %dm %ds" % (h, m, s)
    elif m > 0:
        last_seen = "%dm %ds" % (m, s)
    else:
        last_seen = "%ds" % (s)

    return last_seen

def create_shid_stat_text(shid_list):
    finished = 0.0
    issued = 0.0
    queued = 0.0

    for shid in shid_list:
        if shid_list[shid] == 2:
            finished += 1
        elif shid_list[shid] == 1:
            issued += 1
        else:
            queued += 1

    verbose("Counted stats: finished: " + str(finished) + " issued: " + str(issued) + " queued: " + str(queued))

    finished_p = int(math.floor(finished / len(shid_list) * 100))
    issued_p = int(math.ceil(issued / len(shid_list) * 100))
if queued > 0:
    queued_p = 100 - finished_p - issued_p
else:
    queued_p = 0

verbose("calculated percentages: finished: " + str(finished_p) + " issued: " + str(issued_p) + " queued: " + str(queued_p))

stat_text = ""
if queued_p > 0:
    stat_text += str(queued_p) + "/% queued "
if issued_p > 0:
    if queued_p > 0:
        stat_text += ", "
    stat_text += str(issued_p) + "/% issued "
if finished_p > 0:
    if issued_p > 0:
        stat_text += ", "
    stat_text += str(finished_p) + "/% finished"

return stat_text

def long_read(conn):
    total_data = []
    while True:
        data = conn.recv(8192)
        if not data:
            break
        total_data.append(data)
    return ".join(total_data)

def list_workers(user):
    debug("list_jobs(" + user + ")")
    list_request = {
        "type": "worker_list",
    }
    s = connect_to_queen()
    s.send(json.dumps(list_request))
    reply = long_read(s)
    s.close()
    verbose("Recieved data: " + reply)
    list_reply = json.loads(reply)
    # pretty printing
    header_list = ["CORPUS", "SHID", "WORKER", "IP", "LAST_SEEN"]
    row_format = "{: <20}" * (len(header_list))
    now = time.time()
    print row_format.format(*header_list)
    for worker in list_reply["worker_list"]:
```python
CHAPTER 13. APPENDIX E - CLIENT.PY

time_diff = int(now - int(worker["last_seen"]))
m, s = divmod(time_diff, 60)
h, m = divmod(m, 60)
last_seen = ""
if h > 0:
    last_seen = "%dh %dm %ds" % (h, m, s)
elif m > 0:
    last_seen = "%dm %ds" % (m, s)
else:
    last_seen = "%ds" % (s)

row = [ worker["cid"], worker["shid"], worker["wid "] , worker["ip_address"], last_seen + " ago" ]
print row_format.format(*row)

def list_reducers(user):
    debug("list_jobs(" + user + ")")
    list_request = {
        "type" : "reducer_list",
    }
    s = connect_to_queen()
    s.send(json.dumps(list_request))
    reply = long_read(s)
    s.close()
    verbose("Recieved data: " + reply)
    list_reply = json.loads(reply)
    # pretty printing
    header_list = ["CORPUS","REDUCER","IP","LAST_SEEN"]
    row_format = "|{:<20}|
    now = time.time()
    print row_format.format(*header_list)
    for reducer in list_reply["reducer_list"]: 
        time_diff = int(now - int(reducer["last_seen"]))
m, s = divmod(time_diff, 60)
h, m = divmod(m, 60)
last_seen = ""
if h > 0:
    last_seen = "%dh %dm %ds" % (h, m, s)
elif m > 0:
    last_seen = "%dm %ds" % (m, s)
else:
    last_seen = "%ds" % (s)

    row = [ reducer["cid"], reducer["rid"], reducer[" 
ip_address"], last_seen + " ago" ]
    print row_format.format(*row)

def delete_job(id, user):
    debug("deleting job with id " + id + " by user " + user)```
delete_request = {
    "type" : "delete_job",
    "uid" : user,
    "jid" : id
}

s = connect_to_queen()
s.send(json.dumps(delete_request))
s.close()

def get_job_state(id, user, extended, all):
    debug("get_job_state(" + id + "," + user + ")")
    # transform the query to a job in JSON
    job = {
        "type" : "job",
        "user" : user,
        "jid" : id
    }

    # j_job = json.loads(job)
    verbose(str(json.dumps(job)))

    # open connection to queen bee
    s = connect_to_queen()

    # submit job
    s.send(json.dumps(job))

    # get ID from returned job
    data = long_read(s)

    # print ID back to user
    s.close()
    verbose("received data: " + data)

    if data == ":
        print("No job found")
        return

    job = json.loads(data)

    if job["status"] == "deleted":
        print "Job Deleted"
        return

    print create_shid_stat_text(job["shid_list"])

    if extended:
        if len(job["shid_list"]) > 4:
            columns = math.floor(math.sqrt(len(job["shid_list"])))
    verbose("calculated column width: " + str(columns))

        row_format = "{:<1}" * (int(columns))
        row = []
        col_count = 0
        shid_count = 0
for shid in job["shid_list"]:
    if job["shid_list"][shid] == 0:
        row.append(".")
    elif job["shid_list"][shid] == 1:
        row.append("o")
    else:
        row.append("X")

    col_count += 1
    shid_count += 1

    if col_count == columns:
        print row_format.format(*row)
        row = []
        col_count = 0
    elif shid_count == len(job["shid_list"]):
        # add blanks to the last row

        for num in range(col_count, int(columns)):
            debug("Padding additional space")
            row.append(" ")

        print row_format.format(*row)

    print json.dumps(job["results"])  

    if allr:
        print """"MAP RESULTS """
        print json.dumps(job["map_result"])  

def get_stats(cid):
    debug("Get stats for corpus " + cid)

    list_request = {
        "type" : "stats_request",
        "corpus" : cid
    }

    s = connect_to_queen()
    s.send(json.dumps(list_request))
    reply = long_read(s)
    s.close()

    verbose("Recieved data: " + reply)

    list_reply = json.loads(reply)

    print json.dumps(list_reply, indent=4, sort_keys=True)

    # TRAFFIC CONTROL
    verbose("corpus: " + CORPUS + " QUERY: " + QUERY + " USER: " + USER)

    # someone wants to list workers
    if ( WORKERS ):
if ( REDUCERS ):
    list_reducers(CORPUS)
    sys.exit(0)

# Someone submits a new query
if ( CORPUS != "None" and USER != "None" and QUERY != "None" ):
    send_query(USER,CORPUS,QUERY,REDUCE)
    sys.exit(0)

# Someone lists all their jobs
if ( LIST and USER != "None" ):
    list_jobs(USER)
    sys.exit(0)

# Deleting the job
if ( ID != "None" and USER != "None" and DELETE):
    delete_job(ID,USER)
    sys.exit(0)

# Someone checks the status of their job
if ( ID != "None" and USER != "None" ):
    get_job_state(ID,USER,EXTENDED,ALL_RESULTS)
    sys.exit(0)

if (CORPUS != "None"):
    get_stats(CORPUS)
    sys.exit(0)

# If all else fails
print "No suitable combination of options was found, exiting"
sys.exit(1)

########################################################################
Chapter 14

Appendix F - Scripts for instrumentation

14.1 Draw DtMax Graph

```python
import matplotlib.pyplot as mp
import json
import subprocess

with open("Experiments/Experiment5/stats_5.txt") as data_file:
    data = json.load(data_file)

jobs = subprocess.check_output("cat Experiments/Experiment5/stats_5_listjob.txt | awk '{print $1}' | grep -v JOB", shell=True)
job_list = jobs.split("\n")
dt_list = []
order = []

for job in data["job_list"]:
    order.append(job_list.index(job["jid"]) + 1)
    dt_list.append(job["DtMax"])

print(dt_list)
print(order)

ordered_list = [dt_list for (order, dt_list) in sorted(zip(order, dt_list))]

mp.plot(ordered_list, marker='o')
mp.ylim(5, 15)
mp.title("DtMax vs time for Experiment 6")
mp.xlabel("job id")
mp.ylabel("time in seconds")
mp.grid()
#mp.savefig("Experiments/Experiment4/DtMax_Experiment4.pdf")
mp.savefig("Experiment_png/Experiment5/DtMax_Experiment5.png")
mp.show()
```
14.2 Draw Worker Distribution

```python
import matplotlib.pyplot as mp
import json
import numpy as np
import subprocess

with open("Experiments/Experiment6/stats_6.txt") as data_file:
    data = json.load(data_file)

jobs = subprocess.check_output("cat Experiments/Experiment6/
    stats_6_listjob.txt | awk '{ print $1 }' | grep -v JOB", shell=True)
job_list = jobs.split("\n")

for job in data["job_list"]:  
    tid_us = []
    tid_st = []
    shid = []
    title = "Job nr " + str(job_list.index(job["jid"]) + 1) + 
        " with id " + job["jid"]

    shids = job["shid_list"]
    for shidnr, stats in shids.items():
        shid_split = str(shidnr).split("u")
        shid.append(int(shid_split[0]))
        tid_us.append(stats["time_used"])  
        tid_st.append(stats["time_started"])  

    mod_tid_st = [i - min(tid_st) for i in tid_st]
    xticks = np.arange(0, max(tid_us) + max(mod_tid_st) + 10, 30)
    yticks = np.arange(0, len(shid) + 10, 10)
    # yticks = np.arange(0, 11, 1)

    mp.xlim(-1, max(tid_us) + max(mod_tid_st) + 10)
    mp.ylim(-1, len(shid) + 2)
    # mp.ylim(0, 10)
    mp.xticks(xticks)
    mp.yticks(yticks)

    mp.xlabel("Time in seconds")
    mp.ylabel("Shard id")

    mp.title(title)
    mp.grid()

    for line in range(len(tid_us)):
        # if shid[line] < 10:
        #     if shid[line] == 1:
        #         mp.plot([mod_tid_st[line], mod_tid_st[line]+tid_us[line]], [shid[line], shid[line]+1], linewidth=1.5)
        # else:
        #     mp.plot([mod_tid_st[line], mod_tid_st[line]+tid_us[line]], [shid[line], shid[line]], 'g',
        #     linewidth=1.5)

    mp.savefig("Experiment_png/Experiment6/"+str(job_list.
        index(job["jid"]) + 1)+"+="+job["jid"]
```
14.2. DRAW WORKER DISTRIBUTION

```python
#mp.savefig("Experiments/Experiment12/"+str(job_list.index(job["jid"]))+"+
"+job["jid"]+".pdf")

print(title)
mp.gcf().clear()
```
14.3 Draw Job Distribution

```python
from __future__ import division
import matplotlib.pyplot as mp
import json
import matplotlib.patches as mpatches
import numpy as np
import subprocess

with open("Experiments/Experiment5/stats_5.txt") as data_file:
    data = json.load(data_file)

jobs = subprocess.check_output("cat Experiments/Experiment5/stats_5_listjob.txt | awk '{ print $1'} | grep -v JOB", shell=True)
job_list = jobs.split("\n")
mod_tid_st = []

for job in job_list:
    for j in data["job_list"]: 
        if job == j["jid"]:
            mod_tid_st.append([j["Tadm"], j["Tstart_stop"]])

mp.grid()

count = 1
for line in mod_tid_st:
    if count%3 == 0:
        mp.plot([line[0], line[1]], [mod_tid_st.index(line), mod_tid_st.index(line)], 'k', linewidth=1.5)
    else:
        # Line between start and stop times
        mp.plot([x for x in mod_tid_st], [x for x in range(len(mod_tid_st))], 'b', linewidth=0.0)
        mp.plot([x for x in mod_tid_st], [x for x in range(len(mod_tid_st))], 'r', linewidth=2.0)
    count += 1

# Colour a part of the graph
mp.axvspan(mod_tid_st[15][1], mod_tid_st[-1][2], facecolor='#ff8c00', alpha=0.5)

# Predicted line in case
stop_grad = (mod_tid_st[13][2] - mod_tid_st[0][2]) / 13
mp.plot([(x*stop_grad) + mod_tid_st[0][2] for x in range(len(mod_tid_st))], [x for x in range(len(mod_tid_st))], 'g--', linewidth=1.0)

# Legend
red_patch = mpatches.Patch(color='red', label='Corpus A')
```
14.3. DRAW JOB DISTRIBUTION

```python
45 # green_patch = mpatches.Patch(color='green', label='Predicted line
46 # without duplicate worker')
47 mp.legend(handles=[green_patch], bbox_to_anchor=(0., 1.02, 1.,
48 .102), loc=3, ncol=2, mode="expand", borderaxespad=0.)
49
50 xticks = np.arange(0, mod_tid_st[-1][1]+60, 60)
51 mp.xlim(mod_tid_st[-1][1])
52 mp.xticks(xticks)
53
54 mp.title("Experiment with 30 jobs")
55 mp.xlabel("Time in seconds")
56 mp.ylabel("Job id")
57 mp.savefig("Experiment_png/Experiment5/Job_dist")
58 #mp.savefig("Experiments/Experiment12/Job_dist.pdf")
59 mp.show()
```