FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Load Balancer for Bursting Hadoop-based Network Analysis Jobs

Pedro de Serpa Caiano Rocha Gonçalves

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Professor Ricardo Morla

July 30th, 2015

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MIEEC - MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES | **2014/2015**

A Dissertação intitulada

"A Load Balancer for Bursting Hadoop-based Network Analysis Jobs"

foi aprovada em provas realizadas em 24-07-2015

o júri

Presidente Professor Doutor Artur Manuel Oliveira Andrade de Moura
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

Professor Doutor Tiago José dos Santos Martins da Cuz
Professor Auxiliar do Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Professor Doutor Ricardo Santos Morla
Professor Auxiliar do Departamento de Engenharia Eletrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

O autor declara que a presente dissertação (ou relatório de projeto) é da sua exclusiva autoria e foi escrita sem qualquer apoio externo não explicitamente autorizado. Os resultados, ideias, parágrafos, ou outros extratos tomados de ou inspirados em trabalhos de outros autores, e demais referências bibliográficas usadas, são corretamente citados.

Autor - Pedro de Serpa Caiano Rocha Gonçalves

Faculdade de Engenharia da Universidade do Porto

Abstract

In the age of the Internet of Things, all sorts of devices are turning into connected gadgets that can share information to the Internet and improve our lives. Today there are millions of these devices generating unabating amounts of data that need to be processed and analyzed. These large sets of data are also known as Big Data. The ability to take value out of Big Data can be seen as a major asset for competing companies. However, the processing of Big Data is unsuitable for traditional computing systems. Apache Hadoop is a framework that is designed to handle these large data sets. Though, as the number of connected devices grows, so does the amount of streamed data to be processed. Existing infrastructures might not be capable of handling the increased demand for processing power, leading to unwanted delays.

Our study introduces High Availability techniques that can be deployed in a Hadoop cluster to reduce downtime and increase throughput. More importantly, it focuses on a specific technique called Cloud Bursting to enhance throughput and avoid server overloads, by leveraging a local private infrastructure through the usage of additional on-demand resources on a public Cloud. This hybrid cloud model is becoming a trend, as it offers scalability when the need arises. We present a Cloud Bursting solution for a Hadoop cluster that processes and analyzes network traffic through the use of the Packet Capture (PCAP) application programming interface (API). We introduce our map-intensive network analysis MapReduce job and present a simple model used to estimate its behavior. Our solution is powered by a custom inter-cluster Load Balancer that decides when to burst jobs. After a series of tests, our Load Balancer shows that in fact the Cloud Bursting technique is a viable option to improve the availability by minimizing job delays and maximizing cluster utilization when peaks of network traffic occur.

Resumo

Na era da Internet das Coisas, todo o tipo de dispositivos estão a tornar-se em aparelhos capazes de se ligarem à Internet para partilhar informação e melhorar as nossas vidas. Hoje, existem milhões destes dispositivos a gerar quantidades exorbitantes de dados que necessitam de ser processados e analisados. Estas quantidades enormes de dados são hoje conhecidas como Big Data, sendo que a capacidade de lhe tirar proveito pode ser vista como um trunfo entre empresas concorrentes. No entanto, o processamento de Big Data não é adequado para sistemas tradicionais de computação. Neste sentido, o Apache Hadoop é uma ferramenta desenhada para lidar com estas quantidades exorbitantes de dados. Contudo, à medida que o número de dispositivos capazes de se ligarem à Internet aumenta, também aumenta a quantidade de dados para processar. É possível que as infrastructuras existentes não sejam capazes de lidar com o aumento excessivo de dados e a consequente necessidade de poder de processamento, provocando atrasos indesejados.

Esta dissertação discute técnicas de alta disponibilidade que podem ser aplicadas num cluster Hadoop, por forma a reduzir o tempo de indisponibilidade e aumentar o rendimento. Particularmente, é dado especial ênfase a uma técnica conhecida por Cloud Bursting para aumentar o rendimento e evitar sobrecargas, que se baseia no princípio de aumentar a capacidade de um cluster local privado através da utilização de recursos adicionais numa Cloud pública. Este modelo híbrido de Clouds está a tornar-se numa tendência devido à sua escalabilidade quando a necessidade surge. Apresentamos uma solução baseada na técnica Cloud Bursting para um cluster Hadoop dedicado ao processamento e análise de tráfego de rede, suportado pela interface de programação de aplicações (API) de captura de pacotes (PCAP). Introduzimos um trabalho MapReduce para análise de tráfego de redes cujo processamento se foca na fase de map, e apresentamos um modelo simples para estimar o comportamento do trabalho. A nossa solução é baseada num balanceador de carga que decide fazer o bursting de trabalhos com base num conjunto de regras. Verificamos após uma série de testes que de facto o Cloud Bursting é uma técnica viável para melhorar a disponibilidade, na medida em que minimiza os atrasos dos trabalhos e maximiza a utilização do cluster local na ocorrência de picos de tráfego de rede.

Acknowledgments

I take this opportunity to express my gratitude to my supervisor Professor Ricardo Morla for all the help and support. Professor Ricardo's insight, expertise, guidance, constructive criticism and friendly advice were crucial to the development of this dissertation and I could not be more grateful for his commitment.

I would also like to show my gratitude to Paulo Vaz, the computer networks laboratories technician, for his support and assistance with the hardware in the network labs.

I also thank my fellow colleagues and friends at FEUP for their contributions and comments, which undoubtedly helped in the development of this dissertation.

Finally, a thank you to my family and close friends for their support and love throughout this hard working semester.

Pedro Rocha Gonçalves

Para o meu grande Avô Zé.

Contents

Abbreviations

API	Application Programming Interface
AWS	Amazon Web Services
CPU	Central Processing Unit
EMR	Elastic MapReduce
FIFO	First In First Out
HA	High Availability
HDFS	Hadoop Distributed File System
IDE	Integrated Development Environment
IT	Information Technology
InvIndex	Inverted Index
JSON	JavaScript Object Notation
MTBF	Mean Time Between Failure
MTTR	Mean Time To Recover or Mean Time To Repair
MWC	Multi Word Count
Mbps	Megabits per second
NAS	Network Attached Storage
NIDS	Network Intrusion Detection System
PCAP	Packet Capture
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
REST	Representational State Transfer
SLA	Service Level Agreement
SPOF	Single Point Of Failure
SSH	Secure Shell
URI	Uniform Resource Identifier
VM	Virtual Machine
WC100	Word Count 100
WC200	Word Count 200
YARN	Yet Another Resource Negotiator

Chapter 1

Introduction

Most people believe the Internet is always available, has no off-switch, is always on and always accessible. Services and web pages such as Google or YouTube never appear to be down and are expected to answer almost instantly. But is it really so that these services never experience downtime? Do they always reply within milliseconds or do they struggle from time to time? Unfortunately failure is inevitable and sometimes servers fail to cope with insane amounts of load generated by unforeseen levels of requests.

In the age of the Internet of Things, almost every networked device gathers and communicates data that requires processing and analyzing. Companies collect huge amounts of data from their customers and their supply chain elements, which need to be stored and processed. The valuable information that one can extract out of these ever-growing large data sets, also known as Big Data, is becoming a distinguishing factor for competing big companies such as Facebook, Twitter and Google. However, traditional computing paradigms and storage methods are not efficient for the handling of Big Data. Apache Hadoop is an open-source framework used to process such large data sets and is used by the big players to extract crucial information needed to better suit their customers' needs. This framework was designed to work on commodity hardware, which might not cope well with spikes of demand for computing power, specially in a multi-tenant environment. Investing in highly available and more powerful hardware to deal with these unexpected spikes is costly and usually leads to underutilisation. The Cloud is becoming an attractive model for many businesses, which need to upgrade their data center, but cannot afford the investment in new hardware. However, even though cheaper than buying new hardware, migrating to the Cloud is still expensive and could compromise critical and sensitive data. Thus, a hybrid model has emerged, providing the best of both worlds. In this scenario, companies with local data centers take advantage of the pay-as-go model of the Clouds to leverage their local infrastructure using the Cloud's resources only when needed. This model enables the bursting of applications from the local data center to the Cloud, when the local infrastructure is overloaded. This technique, also known as Cloud Bursting, is a means of improving the availability and is becoming increasingly popular among many applications.

Our comprehensive study aims to implement High Availability techniques in a private data

center running a Hadoop cluster to analyze network traffic collected by the Packet Capture (PCAP) application programming interface (API). The main focus is the implementation of a Cloud Bursting technique to increase the availability of the Hadoop cluster and understand how beneficial this technique can be when the private data center faces increased demand for computing power. Our proposed Cloud Bursting solution consists of an inter-cluster Load Balancer responsible for bursting workload to a public cloud when the local data center is overloaded.

1.1 Structure

This study is divided into several chapters. Chapter 2 introduces Big Data and describes the current version of Apache Hadoop. It then explains what being highly available means and presents the state of the art for high availability techniques that can be applied to Hadoop, including a section dedicated to Cloud Bursting. Chapter 3 presents our map-intensive network traffic analysis MapReduce job and describes our simple model to estimate the job's behavior. It also introduces the PCAP API and describes our network topology. Chapter 4 thoroughly describes our Cloud Bursting capable Load Balancer and all of its components. We present the results achieved with our solution in chapter 5 and conclude in chapter 6.

Chapter 2

State of the Art

This chapter starts by introducing Big Data and Hadoop YARN, a framework designed to handle data sets such as Big Data. It then follows by explaining the meaning of High Availability and presents the state of the art of High Availability techniques that can be deployed in a Hadoop YARN cluster to improve availability and throughput. Lastly, it introduces the Cloud Bursting technique and mentions currently available middleware used to implement this technique. This chapter concludes by providing examples where MapReduce and similar processing paradigms are used with local and cloud resources.

2.1 Big Data

The amount of services available on the Internet grows every day and so does people's demand for more features. Over the past years the volume of data has skyrocketed and tends to continue to do so. In 2011, an average of 235 terabytes of stored data was reached for companies with more than 1000 employees in 15 of the US economy's 17 sectors [1]. In the age of the Internet of Things, every networked device, such as smartphones, smart energy meters, cars and so many more, generates and transmits data, contributing to the ever increasing amount of data [2]. Big companies that provide services like Facebook, Yahoo! or Google collect huge amounts of data from their users to provide a more customized user experience. These huge amounts of data are also known as Big Data and may go up to terabytes or even to a few petabytes [3].

The term "Big Data" refers to data sets so large that they are not suitable for typical database software. Though subjective, this definition is future proof in the sense that, as technology evolves over time, so does the size that defines a data set to be considered Big Data [2]. In 2001, Doug Laney, an industry analyst, described his view of the current mainstream definition of Big Data as the three Vs of Big Data: volume, velocity and variety.

The volume of data, as explained before, has been increasing due to many factors. The greatest issue is how to take relevancy from such large amounts of data. The other aspect of Big Data leans towards velocity, the speed at which such data is streamed. Lastly, variety refers to the fact that

data can come in all sorts of formats, such as structured, unstructured or numeric data, or even video, audio or text [4, 5].

SAS Institute Inc. embraces two additional key words when referring to Big Data: variability and complexity. Variability describes the unpredictability of certain data sources, which at times may generate peak data flows. For example, data generated by social networks may peak if something trendy emerges. Finally, complexity refers to the need to address and correlate data flowing from multiple different sources [5].

Big Data is here to stay and the ability to take value out of it should be regarded as a distinguishing asset, especially between competing companies. McKinsey Global Institute found that taking relevancy out of data may enhance "productivity and competitiveness of companies and the public sector" and create "substantial economic surplus for consumers" [2].

The next section introduces the Apache Hadoop YARN framework and provides a detailed view of its internal components. This framework was designed to handle Big Data, while providing an abstraction layer over the low level core details, such as resource management, enabling developers to focus on the logic of their applications.

2.2 Apache Hadoop YARN (Yet Another Resource Negotiator)[1]

Traditional computation systems are not efficient for the handling of large data sets. In this sense and in an attempt to address this issue, Google introduced GFS, which stands for Google File System and represents Google's distributed file system. Its goals are the same as other distributed file systems: performance, reliability, scalability and availability [6]. However, GFS's drawback is the fact that it is not open source, which led to the creation of Apache's Hadoop framework. Hadoop is considered the de facto industry framework [7] and is an Apache top-level project. It is thought from the ground up to be fault tolerant and to run on commodity hardware [8, 9]. Hadoop follows a typical master-slave configuration and its current implementation can be divided into two major components: an open source implementation of a distributed file system (HDFS) and of a resource manager (MapReduce Generation 2) [10].

The file system component of Hadoop (HDFS) stores file system metadata and application data separately. The metadata is stored on the cluster's Master Node, in the NameNode service. In contrast, application data is spread out through the DataNode services, located in the Slave Nodes. Unlike traditional forms of data availability such as RAID, HDFS provides high availability and reliability by replicating the file contents on multiple DataNodes that are scattered in the cluster. When a file is uploaded to HDFS, it gets broken into blocks, which are spread out through the DataNodes in the cluster. By default, a replication factor of three is applied, which means that the same block will exist in three different separate DataNodes. The NameNode does not store any application data. Instead it keeps track of the locations of the blocks throughout the

[1] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

whole cluster. If an application wishes to access a certain file, it will ask the NameNode for the location of the blocks and communicate directly with the closest DataNodes to retrieve the data [11].

MapReduce Generation 2, also known as YARN and whose architecture is depicted in figure 2.1, is essentially based on the idea of separating resource management and coordination of logical execution plans in two layers. More specifically, the Master Node houses the ResourceManager, which is responsible for allocating resources, tracking current resource usage, monitoring slave nodes liveness, and arbitrating contention among tenants. To manage allocation of resources, the ResourceManager communicates through a heartbeat mechanism[2] with a special system daemon called NodeManager, which runs on every slave node. With regard to the slave node in which it resides, the NodeManager's job is to monitor resource availability, report faults, and manage resource lifecycle. Thus, the ResourceManager builds its global view of the cluster's resource availability via communication with the cluster's NodeManagers.

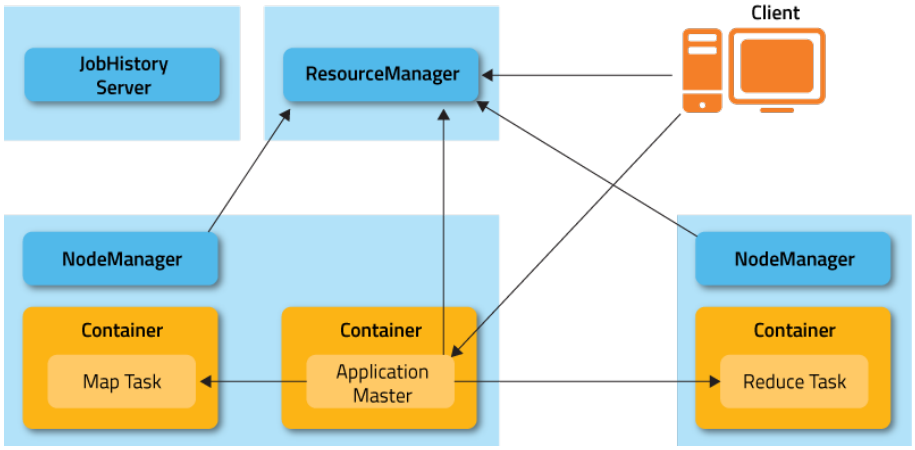

Figure 2.1: Hadoop YARN Architecture[3]

In this context, resources are addressed as containers. A container is a logical bundle of resources, such as 2 CPUs and 4GB of RAM, and is part of a specific (slave) node. Moreover, a per-application ApplicationMaster, which requests resources from the ResourceManager, coordinates the logical plan for the submitted job based on the received resources and monitors the execution of the job. In this scenario, jobs are issued to the ResourceManager, which allocates a container for the ApplicationMaster on a node in the cluster. ApplicationMasters will also need to request resources/containers to be able to complete the issued jobs. To obtain the needed containers, ApplicationMasters communicate with the ResourceManager and specify locality preferences and the properties of the containers. Scheduling policies and resource availability will determine

[2] A heartbeat mechanism is basically a communication system based on a periodic exchange of messages, also called hearbeats.

[3] http://blog.cloudera.com/blog/2013/11/migrating-to-mapreduce-2-on-yarn-for-users/ [Accessed: Jul. 27, 2015]

the resources that the ResourceManager will be able to hand to the ApplicationMasters. When the ApplicationMaster receives an available container for its use, it runs application-specific code to process issued jobs. Even though not mandatory, running containers may communicate with an ApplicationMaster to update their status and eventually receive application-specific commands [9].

One of the most widely used applications is an open-source implementation of MapReduce that is designed for large clusters. It is important to note that, even though there are other applications, this study will focus on Hadoop's MapReduce implementation. MapReduce is a computing paradigm capable of processing many terabytes of data on thousands of computing nodes in a cluster, automatically handling low-level issues such as failure handling, application deployment, aggregation of results and others, thus allowing programmers to focus on the logic of the MapReduce jobs. The flow of a job is illustrated in figure 2.2. Each job can be divided in two operations: a map operation and a reduce operation, hence the name MapReduce. The map operation takes input data and generates key/value pairs. Input data for a map operation is also known as an input split. An input split is a logical representation of the data stored in the HDFS blocks. The need for this representation arises from the fact that HDFS is not aware of the content inside of the blocks, leading to situations where data is divided between two blocks. The input split carries additional information for the location of the divided data. The set of all map operation outputs is the set of intermediate results. These intermediate results are fed to the reduce operations, which merge all intermediate values that match the same key. These operations occur during two main phases: a map phase and a reduce phase. Efficiency is achieved due to the fact that each map operation is independent, thus meaning that all map operations can be executed in parallel on various computers during their respective map phase. The same idea is applied to the reduce operations during their respective reduce phase. During the map phase, the user-specified logic, also known as job, is applied to the input data. When all map operations relative to a certain job are done and the map phase ends, its results, the intermediate results, are used as input to the reduce phase. To be used as input to the reducers, the intermediate results go through two additional phases: a shuffling and a sorting phase. During the shuffling phase, each reducer reaches out to all mappers in the cluster to fetch a portion of the key range of the intermediate results. In the sort phase, intermediate results that share the same key are grouped together. Finally, as soon as these phases end, the reduce phase commences and reducers combine their respective intermediate results and form the final result [8, 12, 13]. In YARN, map and reduce operations are executed on containers, which report to a MapReduce ApplicationMaster. Each user-submitted MapReduce job will spawn its own MapReduce ApplicationMaster that will terminate as soon as the job is done, assuming everything runs as expected [14].

So, simply put, Hadoop MapReduce handles Big Data by taking a huge file and dividing it into small blocks. These small blocks are then replicated and spread out through the DataNodes. The

[4] http://blog.cloudera.com/blog/2014/03/the-truth-about-mapreduce-performance-on-ssds/ [Accessed: Jul. 27, 2015]

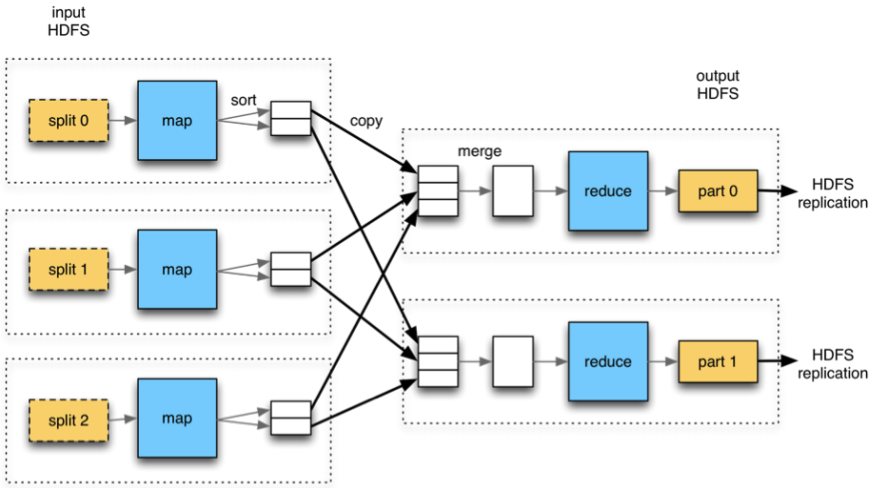

Figure 2.2: MapReduce Job Flow[4]

blocks are then processed in parallel by the Mappers (i.e. containers allocated to run map tasks) and their results are then aggregated by the Reducers (i.e. containers allocated to run reduce tasks) to provide the end result.

One of the key aspects of the Hadoop framework is the fact that it is designed to run on commodity hardware. Commodity hardware does not necessarily translate to low-quality hardware. Instead it is more related to affordability than to quality. However, even high quality hardware has a lifespan and is prone to fail. Moreover, as mentioned in the previous section, data streams suffer from huge variability, causing some clusters to struggle with the increased demand for processing power. The next section gives a brief overview of what being highly available means, while section 2.3.2 provides more details on some techniques that are used to enforce High Availability in a Hadoop cluster.

The initial idea of Hadoop generation one was to address the unprecedented scale required to index the web crawls. Its execution architecture was designed specifically for this use case. Essentially it was built to run MapReduce jobs, so resource management and coordination of MapReduce jobs were tightly coupled in the Master Node. Hadoop became a success among web companies and other organizations. In fact, "it became the place within an organization where engineers and researchers have instantaneous and almost unrestricted access to vast amounts of computational resources and troves of company data." [9, p.1] However, the popularity of Hadoop also led to the discovery of its limitations, specially as developers stretched the MapReduce paradigm beyond the cluster's management layer. A common scenario involved the submission of apparent "MapReduce" jobs that would trigger alternate frameworks. The scheduler interpreted these jobs as map-only jobs with different resource needs. However, these abnormal jobs were not designed according to the assumptions on which the platform was built, which caused poor utilization, even-

tual deadlocks and instability. The need for other programming models was obvious, particularly due to the fact that the MapReduce paradigm is not the perfect candidate for all large-scale computation use cases[9]. Another typical scenario involved the sharing of the same cluster between different users. Data consolidation can prove itself very beneficial as it did to Facebook Engineers when they built their data warehouse. However, performance degraded as the number of simultaneous users grew [15]. The need for multi-tenancy and different application support was clear. To address these issues and others were not covered here, MapReduce Generation 2 was created.

2.3 High Availability

This section defines High Availability and provides an overview of its meaning. It then details several techniques that can be used to enhance the availability of a Hadoop cluster.

2.3.1 Overview

The term "Availability" is well defined and describes the fact that something is readily obtainable and at hand when needed. It provides a measure of the time a system is working as expected and can be described mathematically by the following formula:

$$A = \frac{MTBF}{MTBF + MTTR}$$

MTBF stands for "Mean Time Between Failure" and MTTR means "Mean Time To Recover". So, upon quick inspection of the formula, one can conclude that there are two ways of increasing availability: increase the MTBF and prevent the system from failing or decrease the MTTR and reduce the time it takes to recover from a failure. Since components almost inevitably fail, it is easier for system administrators to decrease the MTTR. Nevertheless, it is always important to consider that some outages will happen such as a simple scheduled maintenance. So the reduction of the time it takes to recover from an outage is a great starting point to improve availability.

However, the term "High Availability" neither has a clear definition, nor a specific value that defines the point as of which a system can be considered highly available. It is common to find the term "High Availability" associated with failures and downtime. However, we believe that it should not be strictly tied to those ideas. We believe High Availability also refers to systems that are capable of efficiently dealing with the function they were designed to fulfill. Systems need not only be up and running, but they also need to cope with fluctuating demand. For example, an e-commerce website might have its web server up and running for the entire year, dealing with its usual number of customers without a hustle. However, when Christmas arrives and the number of customers ramps up, some customers start to experience slowdowns, while others might not even be able to access the web page. The web server is online, but it is struggling to answer the increased demand, degrading the experience for the customers. As the shopping experience worsens, customers start to search for alternatives who offer a better experience.

So, a possible approach to this term is to consider High Availability a design goal. So, when designing a system, the requirements it has for availability should be well defined. At the end, if the system is designed to fulfill or exceed the specified availability requirements, it can be considered highly available [16]. Evan Marcus and Hal Stern, who wrote "Blueprints for High Availability", define High Availability as the following:

"A level of system availability implied by a design that is expected to meet or exceed the business requirements for which the system is implemented." ([16])

Thus, one can conclude that it will always boil down to a trade-off between the cost of having the system down or struggling to answer the requested demand, and the cost to enforce measures and techniques that ultimately avoid system struggles or reduce downtime [16]. Figure 2.3 displays the Availability Index graph as proposed in [16], which highlights the incremental steps that should be taken to achieve a highly available system, as a function of the investment required to implement them.

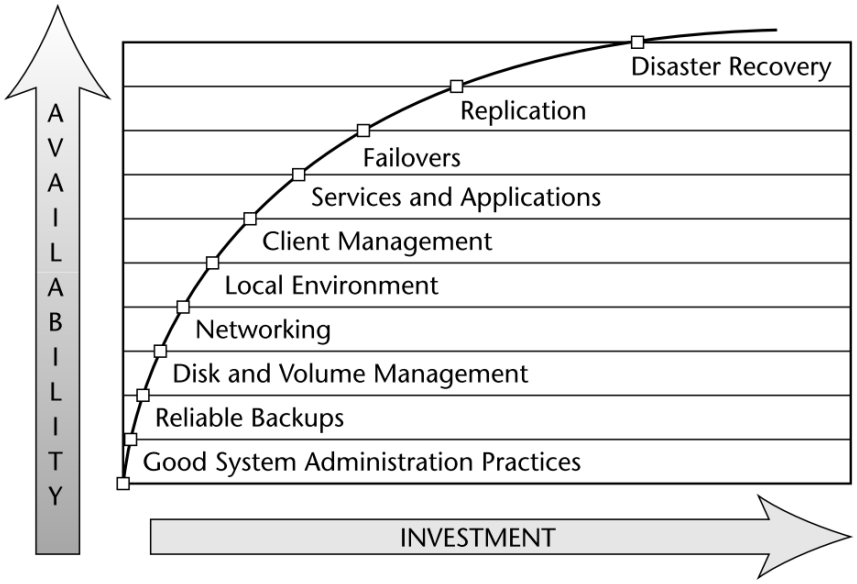

Figure 2.3: The Availability Index as proposed by Evan Marcus and Hal Stern in Blueprints for High Availability[16, p.51]

As expected, figure 2.3 shows that there are many techniques that can be deployed to increase availability. However, as the need for availability rises, so do the cost and the complexity of these techniques. These techniques can go from simple good system administration practices all the way up to full-blown disaster recovery plans. Our study focuses on techniques that can be deployed on Hadoop YARN clusters. More specifically, it narrows down to the idea of Cloud Bursting to enhance the availability and provide additional computing power to increase throughput, should the cluster struggle with peaks of demand.

The next section presents currently available techniques that can be used to increase the availability of a Hadoop cluster, while the section after introduces the Cloud Bursting technique, on which our study will focus.

2.3.2 High Availability in a Hadoop YARN Cluster

Hadoop was designed to be fault-tolerant. This section delivers an overall look into high availability solutions for Hadoop that solve issues ranging from hardware failures up to job scheduling and load unbalance challenges.

2.3.2.1 Node Manager and Resource Manager Hardware Failures

As explained in a previous section, Hadoop follows a typical Master-Slave configuration. HFDS's NameNode and YARN's Resource Manager are housed in Master nodes. If an unplanned event such as a system crash takes the Master node down, the whole cluster becomes inoperable until the Master Node is restarted and finishes booting up. The situation is the same for planned maintenance, such as software or hardware upgrades. Each time a planned maintenance needs to be done, the whole cluster needs to stop, creating time frames of unavailability. Thus, both NameNode and Resource Manager are considered single points of failure (SPOFs), which is undesired for systems that need to be up and available 24/7. Facebook had approximately 150 million files in their HDFS cluster at the time of the writing of [10]. They estimated that a cold-restart of the NameNode took about 45 minutes without the use of a backup node. HDFS's BackupNode provides a faster failover, but it only managed to reduce the cold-restart time to about 20 minutes with the same amount of files. 20 minutes is still too long to be considered as a fast failover and could prove harmful for many businesses. This was the case for Facebook as well, which intended to achieve a NameNode failover within seconds [10].

HDFS High Availability, a feature deployed to solve the issue of fast failover for the NameNode, relies on the idea of running two redundant NameNodes in the same cluster in an Active/Passive setup with a hot standby mode. Only one NameNode can be in the active state at any given point in time, taking responsibility for the client operations in the cluster. The standby NameNode acts as a simple slave and must maintain enough state to ensure a fast failover if necessary. A shared storage between active and passive NameNode is required to keep their states synchronized. This means that the availability of the cluster is limited by the availability of the shared directory. It is important that this shared directory does not become a SPOF, meaning it also requires redundancy. Apache recommends a high-quality dedicated Network Attached Storage (NAS) appliance to ensure high availability [17]. Furthermore, DataNodes must also send block information and heartbeat messages to the standby NameNode. This ensures that the standby NameNode has accurate and up-to-date information regarding the block locations [17].

Failure of the Resource Manager is also undesired, because all containers are terminated and applications restarted, which means jobs have to be re-run. This has a negative impact on the

progress of jobs running before the failure, lowering availability and delaying jobs. A solution similar to HDFS HA is proposed to make the Resource Manager highly available, relying on the idea of adding redundancy in the form of a pair of Active/Passive Resource Managers. This solution allows the possibility of applications resuming their last checkpointed state, meaning that, for example, completed map tasks in MapReduce do not need to be re-run [18, 9].

2.3.2.2 Other YARN Components Failures

YARN's other components don't have high availability requirements like the Resource Manager, which is understandable since containers are supposed to exist in abundant numbers to grant increased availability and throughput. However, there are improvements that can be implemented to further increase the availability and throughput of the whole system. Taking the ApplicationMaster for example, in the event of a crash, the whole application is considered to have failed. By default, the Resource Manager starts a new instance of the ApplicationMaster, meaning the whole application will have to be restarted. This is very inefficient, since already completed work will need to be redone. Generic ApplicationMaster state recovery is not YARN's concern. Instead, the specific frameworks/applications are tasked with the recovery of the ApplicationMaster. In MapReduce, the ApplicationMaster is able to retrieve the state of the tasks, meaning that only uncompleted tasks need to be re-run.

Moreover, the per-node Node Manager's fault recovery isn't a concern of the YARN platform as well. By default, if the Node Manager fails, the Resource Manager marks all of the running containers in the faulty Node Manager as killed and informs their respective ApplicationMasters. This will always have a negative impact in the overall performance, due to the fact that running tasks or jobs on the failed Node will have to be reprocessed. If the fault is transient, however, the Node Manager might be capable of re-syncing with the Resource Manager and continue. Nevertheless, it is the ApplicationMasters' responsibility to react to Node Manager failures and potentially redo work completed by the containers running on the Node during failure.

The failure handling of containers is also a responsibility of the frameworks/applications. By default, the Resource Manager gathers all container exit events from the Node Managers and informs the corresponding ApplicationMasters through heartbeat messages. In MapReduce, the ApplicationMaster listens to these heartbeats and immediately retries the failed map or reduce tasks by requesting new containers from the Resource Manager [9].

2.3.2.3 Scheduling Bottlenecks

Another aspect of Hadoop worth mentioning is its built-in FIFO scheduler for multi-user environments. At Facebook, when groups of Engineers started using Hadoop simultaneously, job response times suffered because of the FIFO scheduler. The disadvantage of FIFO scheduling is the poor response time it introduces for small jobs in the presence of large jobs. Thus, a Fair Scheduler was designed that consists of a technique that assigns resources to applications in a way that, on

average, all applications get the same share of resources over time. Currently, the Fair Scheduler supports all YARN applications, meaning it's not restricted to MapReduce. This scheduler is based on the idea of statistical multiplexing, which means it redistributes processing power that is unused by some jobs to other jobs. So, if by chance there is only one application running in the cluster, that application will use the entire cluster. But when more applications join in, freed up resources are allocated to the newly submitted applications. Eventually every application receives the same amount of computing power. Furthermore, the Fair Scheduler provides user (job) isolation, giving each user (job) the illusion of running its private cluster, allowing users to start jobs within seconds, as opposed to the FIFO Scheduler [15, 19].

However, blunt implementation and usage of the Fair Scheduler compromises data locality, which is specially important for applications such as MapReduce. Data locality refers to the need to place computation near the data, thus increasing throughput. The main reason for the increased throughput is the fact that network bisection bandwidth is a lot smaller in a large cluster than the total bandwidth of the cluster's hard drives. So, efficiency is achieved when jobs run on the node containing the data they require. However, when this is not possible, running on the same rack (where the data is located) is still faster than running off-rack. The problem is neither the Fair Scheduler, nor the FIFO Scheduler guarantee that the job to be scheduled next will be placed on a free node containing the data it needs. To solve this issue and increase the availability of the system, a simple algorithm called Delay Scheduling can be deployed. Using this algorithm, jobs wait for a limited amount of time for an opportunity on a node that has the required data. When a node is freed up, it requests a new task. The scheduler takes the head-of-line job (the job to be scheduled next) and tries to launch a local task. If it can't, it skips it and looks at subsequent jobs. However, there is a threshold that defines the number of times a job can be skipped. If the threshold is exceeded, jobs are authorized to launch non-local tasks, thus avoiding job starvation. Delay scheduling manages to bring locality close to 100% by waiting a very small amount. However, it is important to note that Delay Scheduling is only effective if a large portion of tasks aren't much longer than the average job, and nodes have a reasonable amount of slots. Making tasks short improves fault-tolerance, so it is expected that as clusters grow, developers will make the effort to split their work into shorter tasks. And as technology evolves and number of cores per processor increases, more tasks are supported at once, thus meaning that slot numbers should not be an issue [15, 19].

There is another aspect impacting MapReduce, besides data locality, that poses a challenge for the scheduler and more specifically the Fair Scheduler: the dependence between map and reduce phases. As map functions finish, they generate intermediate results as explained before. These intermediate results are stored on disk and each reduce function copies its fraction of the results. But reducers can only start the user's reduce function when they have all the results from all maps. This is very inefficient and can lead to a "Slot Hoarding" problem, which can affect throughput. The "Slot Hoarding" problem happens in the presence of large jobs, which hold reduce

slots for a long time, leading to resource under-utilization and small job starvation. Reduce tasks are normally launched as soon as the first few maps finish. This allows reducers to begin copying map results while remaining maps continue running. However, if the job is large (i.e. it has a lot of map tasks), the map phase may take a long time to finish. The reduce slots will remain held until all maps finish. Thus, if the cluster is experiencing a low activity period, the submission of a large job will take most or all of the reduce slots. In the meantime, if any other jobs are submitted, they will starve until the large job finishes. To solve this issue and increase throughput, a technique called Copy-Compute Splitting can be applied. The idea is to split the reduce tasks into two logically different tasks: copy tasks and compute tasks. The copy tasks are responsible for gathering and merging map results, which is normally a network-IO-bound operation. Compute tasks execute the user submitted reduce function to the map results. To make this technique work, an admission control step is added to the reduce process before the computing phase begins. So, when a reducer finishes the data retrieval phase, it queries the slave daemon on its node for permission to initiate its compute phase. In addition, the number of reducers computing at the same time is limited by the slave, allowing nodes to run more reducers than they have computing resources for. But it also limits the competition for said resources. Furthermore, the number of reducers in the copy phase on a node for each job is limited to the number of computing reducers, allowing other jobs to use the other remaining slots [15].

The use of both Delay Scheduling and Copy-Compute Splitting techniques provide gains of 2-10x in throughput and response time in a multi-user environment, however, even in a single-user environment, using FIFO, these techniques elevate throughput by a factor of 2 [15].

2.3.2.4 Load Unbalance Bottlenecks

Hadoop MapReduce is most efficient in homogeneous clusters, lacking the same efficiency when in a heterogeneous environment. One of the reasons why MapReduce doesn't behave as well in heterogeneous clusters is related to load unbalance of reduce tasks. There are two fundamental reasons why this load unbalance occurs: the native hash partitioning, which does not allocate fair amounts of input data to the reducers; and the hardware heterogeneity. The former is related to the method that is natively used to partition the input data (the output key/value pairs generated by the mappers) among the reducers. The output pairs are split into several partitions through hashing. The problem is that the amount of partitions that each reducer receives is based only on this static partitioning approach, leading to uneven data distribution among reducers for many applications. The world population statistic application is a great example for this case. Reducers in charge of major countries, such as China, will have a lot more work to do than other reducers tasked with smaller countries.

The latter cause refers to heterogeneous clusters, where nodes' performance may differ. This performance difference can further increase load unbalance, since lower performing nodes will take longer to execute the reduce tasks, delaying the whole job. In addition, if the uneven data distribution among reducers is taken into account, it is possible that lower performing reducers

receive the most amount of work, further exacerbating the uneven data distribution issue. To address these issues, a novel load balancing approach has been developed, which provides fairer work distribution among reducers while taking processing power into consideration. This approach evaluates the performance of reducers through the history of reduce tasks, allowing the estimation of a threshold of work that can be allocated to each reducer. This threshold value is then used by a heterogeneity-aware partitioning approach that adaptively reallocates the input data to the reducers. Through experiments in a cluster with 22 virtual machines, the use of this solution achieves an average speed-up of 11% over the native Hadoop approach, presenting itself as a good improvement in performance for MapReduce in heterogeneous clusters [20]. However, despite the improvements brought by this solution, note that it is not implemented natively in Hadoop YARN.

2.4 Cloud Bursting

When discussing Big Data, one of the most important key words that was mentioned in section 2.1 is variability. As explained, variability refers to the dynamic and sometimes unpredictable peak data flows that certain services or applications generate. Infrastructures that are based on an estimation of these peaks require more hardware than what is actually required to fulfill the vast majority of the needs, leading to systems that are underutilized most of the times. Many companies invest in their private IT data centers to meet most of their needs. However, these private data centers, though sufficient for most needs, might not cope with planned or unplanned workload peaks, leading to delays and lower throughput. Using extra computing power to handle the unpredictable and infrequent peak data workloads is expensive and inefficient, since a portion of the resources will be inactive most of the time. Migrating the whole application to a cloud infrastructure, though possibly cheaper than investing in the private data center because servers are only rented when needed, is still expensive and could compromise private and important data. So, depending on the applications, there are privacy issues that must also be taken into consideration.

The hybrid model offers the best solution to address the needs for elasticity when peak workloads appear, while providing local privacy if needed. In this model, the local IT data center of an enterprise is optimized to fulfill the vast majority of its needs, resorting to additional resources in the cloud when the local data center resources become scarce [21]. This technique is called Cloud Bursting and enables an enterprise to scale out their local infrastructure by bursting their applications to a third-party public cloud seamlessly, if the need arises [22]. To seamlessly transition between the local infrastructure and the public cloud, cloud bursting relies on the concept of virtualization, which allows the encapsulation of applications in virtual machines.

2.4.1 Virtualization Tools

Existing open-source virtualized data center management tools such as OpenStack and OpenNebula already support cloud bursting. Their initial focus was to provide an abstraction layer for the low level details of transitioning VMs (Virtual Machines) between data centers [21]. However, throughout the latest years, continuous improvements have been made to the amount of provided

features and configurable parameters to better suit the users' needs. Today, many of the available solutions, such as OpenNebula, Seagull and the OPTIMIS Project offer scheduling policies that determine if and when to burst to the cloud.

Seagull's scheduler goes as far as to consider if it is cheaper and faster to move one or more lighter applications to the public cloud and assign the freed local resources to an overloaded application, as opposed to bursting the stressed application to the public cloud. This is achieved thanks to its greedy heuristic, which determines which applications should be relocated in order to minimize cost. It also intelligently precopies snapshots of the virtual disk's image belonging to an overload-prone application to the public cloud, greatly reducing bursting times. Its primary focus is the optimal placement of applications to reduce cost, rather than handling the division of data between the public and private data centers. The placement algorithm has proven to reduce the total cost of cloud bursting in response to data center overload by 45%, while its precopying strategy manages to burst applications to the cloud with 95% saving in data transfer [21].

The OPTIMIS Project, on the other hand, aims to create an entire cloud ecosystem. Its objective is to implement an open, scalable, dependable and virtualized cloud ecosystem that will improve the delivery of reliable, secure, elastic, auditable and sustainable IT services. The goal is to provide automatic and seamless transition of services and applications from private local data centers to trustworthy and auditable cloud services. It provides mechanisms to monitor the internal cloud resource usage to determine if the provisioned resources are still fulfilling the established service level agreements (SLA) and other requirements, such as energy consumption. Furthermore, it provides a mechanism to detect the nature of the applications running in the internal cloud. These are extremely useful to help decide whether or not it makes sense to burst a specific application to the external public cloud. For example, if the allocated resources for a critical service containing private data are struggling to meet the specified SLA, a good approach would be to force the bursting of a non-critical service whose SLA requirements are being met [22].

2.4.2 Data Splitting

It is important to note, however, that Cloud Bursting does not necessarily need to be tied to the use of additional computing power through an external public cloud. As the world embraces the ever-growing paradigm of Big Data, Cloud Bursting can also be used in the context where cloud resources are used to store additional data if local resources become scarce. In fact, it has been proven that the use of Cloud Bursting for data-intensive computing is not only feasible, but scalable as well. [23] presents a middleware that supports a custom MapReduce type API and enables data-intensive computing with Cloud Bursting in a scenario where the data is split across a local private data center and a public cloud data center. The processing of the data is performed using computing power from both the local and public cloud data centers. Furthermore, data on one end can be processed with computing power from the other end, lowering the overall execution time of jobs. To evaluate the performance of said middleware, three data-intensive applications

have been chosen to run in two different scenarios: A centralized scenario, in which all of the data is stored at one location and processed using the available computing power at the same location; a hybrid scenario, in which the same data set is divided by both the local and public data center, and processed using the same amount of computing power also divided by both data centers. The data distribution is varied to test unevenness situations. As the data is pushed towards the cloud, the cost of remote data retrieval increases, which is expected given that fact that the data needs to traverse the Internet.

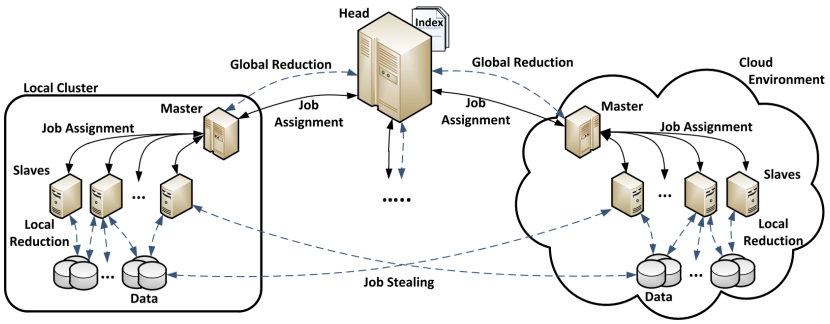

Figure 2.4: System Architecture for the Generalized Reduction Cloud Bursting scenario as presented in [23]

One of the most interesting features of this custom implementation is how it enables fair load balancing. This implementation introduces yet another node type, called head node, which essentially coordinates the inter-cluster communication and manages job assignments. Master nodes are responsible for the communication with the head node and for distributing the jobs among slave nodes. Since slave nodes request jobs in an on-demand logic, the slaves with higher computing power will receive and process more tasks than others. Master nodes also request jobs from the head node on demand, meaning clusters with more computing power perform more processing. Moreover, when a master node requests jobs, the head node takes data locality into consideration and sends suitable jobs to the requesting master. It is interesting to see how this implementation neither requires a technique such as the novel load balancing approach as detailed in section 2.3.2.4 to address the heterogeneity of the clusters, nor does it require the Delay Scheduler as mentioned in section 2.3.2.3 to achieve data locality.

Overall, [23] concludes that the average slowdown ratio when using the Cloud Bursting data processing solution over a pure centralized solution is only 15.55%, yet the system scales with an average of 81% when computing resources are doubled. Thus, the use of a hybrid cloud to perform Cloud Bursting on data-intensive computing is in fact feasible, providing a flexible solution to extend the local limited resources through pay-as-you-go cloud solutions [23].

However, rather than using Hadoop's MapReduce framework, [23] uses a custom data processing framework called Generalized Reduction. The Generalized Reduction framework also has

two phases, though, instead of having a map and a reduce phase, this framework has a local reduction phase and a global reduction phase. The former couples mapping, combining and reducing operations together into the same phase. In this phase, all data elements are locally processed and reduced immediately. After that, the global reduction phase kicks in and all reduced objects from the local reducers are merged using a user defined function or a collective operation. [23] states that the advantage brought by this framework is the avoidance of intermediate overheads created by intermediate steps, such as sorting and shuffling. Furthermore, it states that this is critical for a cloud bursting scenario in order to reduce the inter-cluster communication.

2.4.3 Streaming

[24] presents a framework that enables Cloud Bursting for Hadoop's MapReduce running on a Hadoop YARN cluster. This study points out two challenges that difficult the leveraging of a Hadoop MapReduce cluster using Cloud Bursting: The fact that Hadoop is a batch-processing system and the tight coupling of the shuffle phase with reducers. The former implies that Hadoop can only start computing a job as soon as it has the entire input data to process it. Since inter-cloud bandwidth is far lower than the intra-cloud bandwidth, using the Hadoop's batch-processing system in the public cloud will lead to high start up delays, because Hadoop needs to wait for the entire input data to be uploaded and can only start computing it after. The latter delays job completion times and causes resource underutilisation within reducers. This happens due to the shuffle phase only commencing after reducers have started. Shuffling involves all-all node communication, thus the shuffling of data from the public cloud to the reducers in the private local data center takes longer, due to lower inter-cloud bandwidth compared to intra-cloud bandwidth.

[24] proposes BStream to implement the cloud bursting technique in a Hadoop YARN cluster, while addressing the above difficulties. This framework is based on the usage of a stream processing engine called Storm, that is deployed in the public cloud, and the Hadoop YARN framework, that is deployed in the local private data center. Storm enables the execution of both map and reduce tasks on an incoming stream of data as and when it arrives in the public external cloud. Thus, essentially, this allows the overlapping of input data transfer and its processing in the public cloud, minimizing startup delays. Furthermore, Figure 2.5 shows four possible approaches to implementing inter-cloud MapReduce. BStream's approach is illustrated in 2.5d), in which the output of a MapReduce job in the external cloud is directly downloaded to the reducers in the local data center. So, as explicit in the figure, this framework also executes reduce operations in the external cloud, reducing download sizes to the private data center. Shuffling in the external cloud is also decoupled from the local YARN reducers, which means that reducers can start much later in the job lifecycle. Furthermore, it uses checkpointing techniques to send intermittent reduce outputs from the public cloud to the reducers in the local private data center. So, these features enable pipelined uploading, processing and downloading of data. In addition, BStream is able to decide what parts of a MapReduce job to burst to the cloud, when to burst and even when to start the reducers to meet the job deadline.

BStream framework is built upon the idea that the input data is initially located in the private local data center, where the Hadoop YARN cluster and HDFS are deployed, which is the scenario which makes more sense for companies that use their local data center for normal operation and extend to the public cloud when demand for computing power exceeds the available local resources. However, it is relevant to note that when jobs are burst to the public cloud, they are not stored on a parallel independent instance of HDFS. Instead they are stored in a Kafka queue, which worker nodes in the Storm topology access in parallel to retrieve the input data.

The stream processing engine has undergone thorough testing in the external public cloud and [24] concludes that for jobs which require notable amounts of inter-cloud data transfer, the stream processing engine is a far better solution than the batch-processing system in terms of job completion time, using the same computing power. In fact, Storm achieved job completion times 23% lower than Hadoop in a particular situation where computation time per map and inter-cloud bandwidth is fixed, but the input data size is varied.

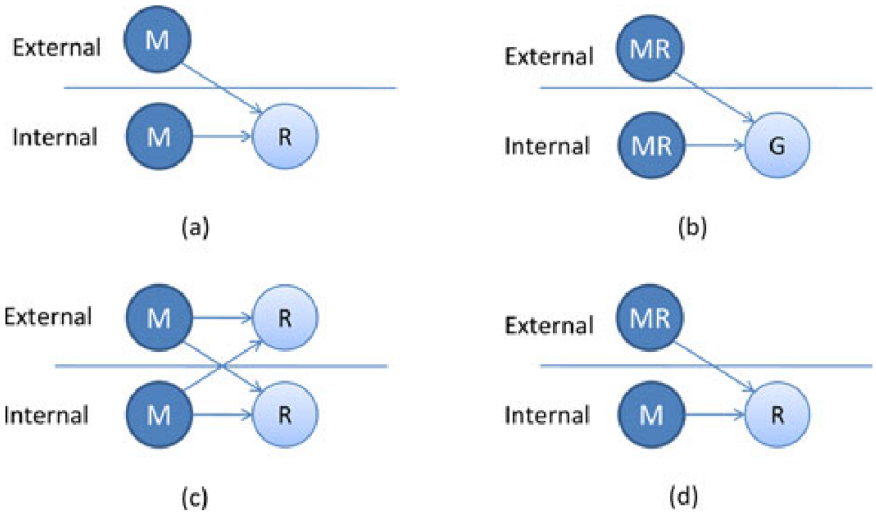

Figure 2.5: Inter-Cloud MapReduce approaches [24]

BStream outperforms several other multi-cloud MapReduce implementations similarly presented in related works. For testing purposes, the authors in [24] implemented a task-stealing mechanism similar to the one introduced in [23], with one of the differences being the fact that the stealing of tasks has no regulation, or in other words, pending map tasks are launched in the external public cloud whenever there are free slots. They called this implementation TaskSteal. Furthermore, [24] also implemented a similar solution to the one proposed in [25], by deploying Hadoop MapReduce in the external cloud with map-aware push and shuffle-aware map optimizations. These optimizations enable Mappers to overlap data pushing with map execution (similar to what BStream achieves with the stream processing engine), while providing faster shuffle and reduce execution through a map task scheduler that is based on the estimated shuffle cost from each mapper. This implementation is referred to as MaSa in [24].

To compare these similar implementations with BStream, the test bench for the similar implementations comprises a single Hadoop cluster spanning across the local private data center and the public cloud, in which reducers only run in the local private data center. This means that local reducers must shuffle intermediate results from both the public cloud and the local data center. In addition, HDFS is only present in the local cluster, meaning that input data is only stored locally.

The authors of [24] conclude that BStream clearly outperforms the similar frameworks in jobs with high output to input ratio (such as MultiWordCount (MWC), InvertedIndex (InvIndex), WordCount100 (WC100) and WordCount200 (WC200)). BStream manages to achieve 8 to 15% better speedups, except for jobs such as Sort and WordCount (WC). Sort's compute to data ratio is low for maps, but it generates highly sized outputs and heavy reduce computation requirements. Thus, overall, Sort does not benefit from Cloud Bursting for all implementations. WC's output to input ratio is very low, so the gained speedup is very similar for all frameworks [24].

2.5 Conclusion

This chapter presented a decent amount of techniques that can be used to improve the availability of Hadoop clusters. Existing solutions range from managing the Master Nodes' hardware failures to provide a quick and seamless failover, all the way up to dealing with scheduling bottlenecks and load unbalance issues to improve throughput. Currently, one of the hot topics in cloud computing is the Cloud Bursting technique, which can ideally lead to zero downtime. This technique, which has yet to mature, has proven to be a feasible and beneficial technique to improve the availability of data intensive applications. By leveraging a public pay-as-you-go cloud, companies no longer need to invest as much in private data centers to meet occasional and sometimes unpredictable spikes in demand for computing power.

The Cloud Bursting capabilities offered by the type of virtualization tools presented in 2.4.1 are adequate for bursting vious types of virtualized applications and services. They offer various scheduling policies and provide an abstraction layer for the low level details of transitioning VMs between data centers. They are, however, not intended to burst Hadoop jobs, as they are not virtualized applications *per se*. In this sense, we do not take advantage of the features and facilities that they offer.

We conclude that the approach seen in [23] is focused on a static Cloud Bursting technique, in which the data is split between local and public cloud resources. While the distribution of data is varied, cloud resources are not allocated dynamically and specifically when needed, being previously specified and allocated instead. Moreover, the Cloud Bursting technique used in [23] is applied to a proprietary, custom version of MapReduce, which albeit similar to Hadoop's MapReduce, is not the same open-source implementation. [24]'s approach is more closely related to what our study aims to achieve. [24] proposes BStream, a Cloud Bursting implementation which uses Hadoop YARN with MapReduce in the local data center, but uses a stream processing engine called Storm in the cloud to process MapReduce jobs, instead of using YARN. BStream only uses

the cloud resources if it estimates that the local data center cannot handle the job deadline, which is similar to the approach we intend to take.

We realize that Cloud Bursting has yet to be deployed as a technique to enhance the availability of a local Hadoop YARN cluster running Hadoop MapReduce jobs to process and analyze network traffic. Our focus is to develop a Cloud Bursting technique for a network analysis Hadoop cluster that will burst jobs to a public cloud when the local cluster is overloaded with its workload. The next chapter presents our network traffic analysis job models and the chapter afterwards presents our Cloud Bursting solution.

Chapter 3

Map-Intensive Network Analysis Job

This chapter introduces and details the map-intensive network analysis job intended for use with our integrated solution. It also covers the network topology in our network laboratory and explains where our integrated solution should be deployed. In addition, it explains how network traffic is captured and discusses various types of network analysis.

3.1 Network Topology

Our study was conducted in the network laboratories at the Faculty of Engineering of the University of Porto (FEUP). Figure 3.1 depicts the network topology of our network labs.

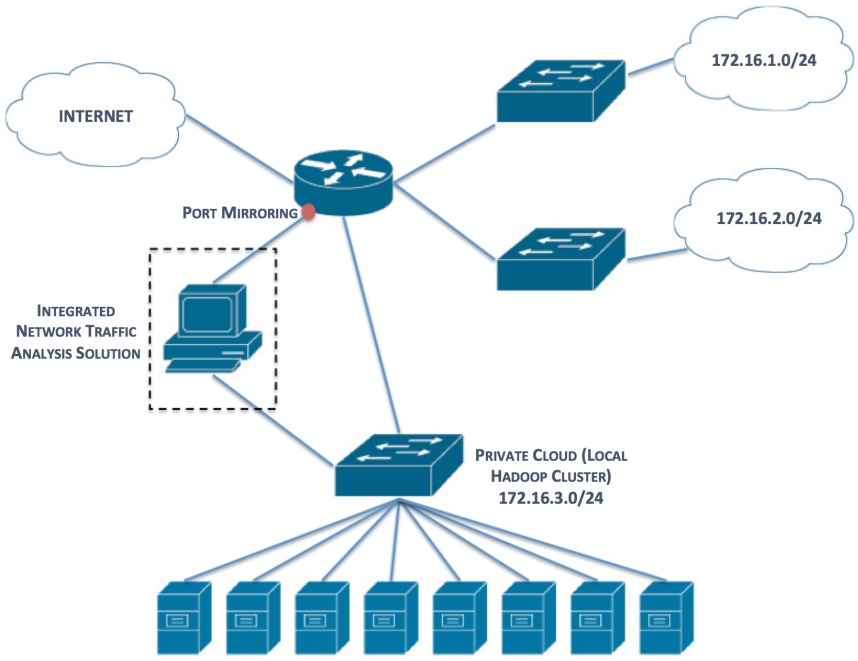

Figure 3.1: The network topology where our Integrated Solution was tested

There are two network laboratories (two separate rooms) and each laboratory has its own switch and own network. Both of the lab switches are connected via FastEthernet links to a router. Our private Cloud (our local Hadoop cluster) is located in one of the labs and consists of eight Hadoop nodes connected to another switch via FastEthernet links, which is also connected to the router via a FastEthernet link. The router is connected to the Internet. There are over 24 hosts capable of generating network traffic in each laboratory.

To analyze the generated outgoing and incoming network traffic in this topology we propose our Integrated Network Traffic Analysis Solution. To setup our solution, an interface in the router must be configured with port mirroring to forward all traffic that goes through the router to that interface. Our solution must have an interface connected to the interface with port mirroring on the router, and another interface connected to the switch dedicated to the private Cloud. The network traffic forwarded through the port mirroring enabled interface on the router will be captured by our solution, stored in PCAP files, and either sent to the local Hadoop cluster or burst to the public Cloud, depending on the local cluster's workload.

3.2 Packet Capture (PCAP)

The monitoring of network traffic is a crucial task of every network administrator. It provides a means to detect network bottlenecks, maintain the efficiency of the network, solve network problems, and determine if security policies are being followed.

Packet Capture (PCAP) is an application programming interface (API) used to capture network traffic that is traversing over a network. It is implemented in the libcap library, which is also used by many open source and commercial tools such as packet sniffers, network monitors and network intrusion detection systems. PCAP allows the collection of all packets on a network and supports storage of captured packets for later analysis [26, 27].

[28] distinguishes three types of network traffic analysis: real-time analysis, batched analysis, and forensics analysis. In real-time analysis, continuous streams of data are analyzed and processed as they arrive. Alternatively, small batches, also known as buffers, can be used to more efficiently analyze the incoming stream of data. This type of analysis requires a decent amount of computational resources and provides low response time[1] due to the reduced delay in receiving the data and the fact that real-time analysis is fully automated. In batched analysis, data is aggregated in batches and analyzed periodically. Overall, response times are higher, though computational resource requirements are lower. Finally, forensics analysis are analysis that only occur when special events are triggered, for example, when an intrusion is detected.

In our study, we intend to use our Hadoop YARN cluster and the MapReduce programming paradigm to process PCAP files and perform batched network analysis. The use of Hadoop and MapReduce for networking analysis is not a new topic. [29] proposes a traffic measurement and

[1] In this context, response time is the elapsed time between the occurrence of a specific event and the processing or detection of that event.

analysis system that uses versions of Hadoop prior to YARN, and does not use Cloud Bursting to improve throughput and availability. Due to the lack of an automatic middleware for resource provisioning based on the traffic load, [29] focuses on an offline traffic collection and analysis system. In addition, in [29] file sizes for the submitted MapReduce network analysis jobs range from 1 TB to 5 TB, which produce large jobs.

We implement a traffic monitoring system with batched analysis through the submission of small MapReduce jobs. Our traffic gatherer captures small batches of data and immediately submits a MapReduce job to analyze the batch of network data and extract useful network information. The next section details the analysis we intend to perform in our study and explains the model we use to describe the job behavior.

3.3 Network Analysis MapReduce Job

As noted in the section before, the goal of network analysis is to provide the Network Administrator with useful information about the monitored network. There are many issues that can be detected and solved through network analysis. Our aim is to provide the Network Administrator with a means to detect network intrusions through packet payload inspection.

Network traffic is captured and saved in the PCAP format. Hadoop does not support this format by default however, because it is mainly designed to work with text files. Nevertheless, there are already studies that successfully developed support for PCAP files. [29] presents traffic analysis MapReduce jobs that perform IP, TCP, HTTP and NetFlow analysis. To achieve this, the authors in [29] developed the PCAP input format, providing Hadoop with the ability to read PCAP files. Alternatively, RIPE[2] also offers a packet library similar to the one presented in [29]. Our study takes advantage of the PCAP input format developed by [29] due to performance and scalability reasons. As noted by [29], the library developed by RIPE is not prepared for parallel processing of packet records, meaning it does not scale. Furthermore, [29] also mentions its inefficiency in recovering against task failures.

To use Hadoop as a Network Intrusion Detection System (NIDS), we envision a deep packet inspection mechanism that is essentially based on the idea of string comparisons. With the PCAP input format, Hadoop is able to find the packets in the PCAP file, thus enabling the inspection of their payloads to search for patterns that suggest an intrusion or some sort of malware. Figure 3.2 depicts the string comparison algorithm for the MapReduce job, where each packet's payload is inspected searching for a specific pattern. This particular algorithm is intended to run during the map phase. The aggregation of the results, that is, the aggregation of the occurrences where a pattern was found is done during the reduce phase. The resulting MapReduce job is a map-intensive job as most of the processing is done in the map phase.

The mechanism based on the idea of string comparisons is not a new topic and is already used by many network intrusion detection systems (NIDS), such as Snort. In fact, this type of intrusion

[2] https://labs.ripe.net/Members/wnagele/large-scale-pcap-data-analysis-using-apache-hadoop [Accessed: Jul. 27, 2015]

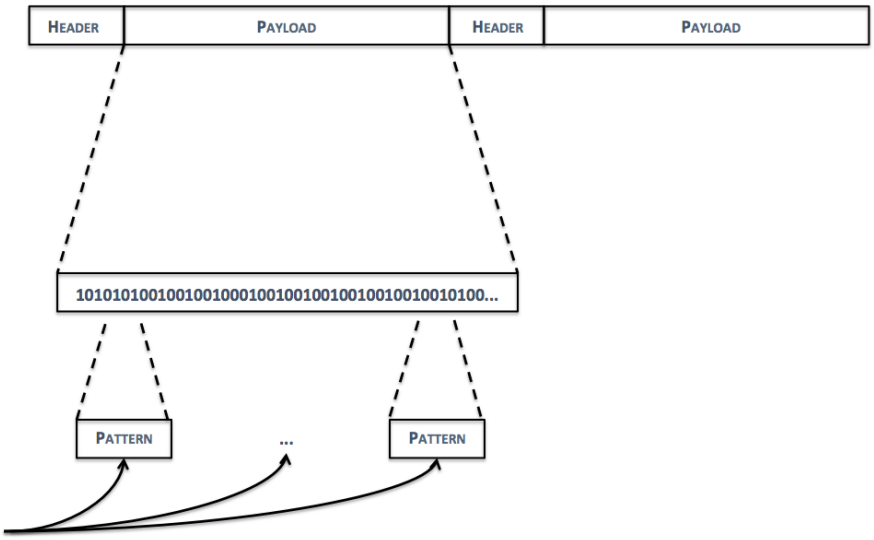

Figure 3.2: Algorithm for the intrusion and malware detection job

detection is also known as network-based signature matching and is part of a broader group of detection systems known as misuse detection systems. The term signature refers to a pattern or raw byte sequence, whereas the term misuse describes the fact the these detection systems search for explicit details that suggest attacks [30]. What makes Hadoop so interesting for this task is that it scales. This allows the analysis of big PCAP files in an efficient way, as can be seen in [29], where network analysis MapReduce jobs for one TB libpcap files had an increase in throughput of up to eight times when more nodes were added to the cluster.

We use two map-intensive MapReduce jobs in our experiments: one that simulates processing and sleeps a certain number of milliseconds on each packet, and another that actually compares packet payload with a set of 200 strings from the Snort community rules. While the first has no real use, it was useful in our experiments with synthetic traffic to understand the behavior of our system. The source code for both jobs can be seen in A.1. Since our focus in this dissertation is the bursting algorithm, we do not explore the detection performance of the packet payload job.

3.4 Job Model

After a study of the simulation job's behavior, we came up with a simple model to describe the job's behavior. This model is important due to the Load Balancer's need to predict job completion times to aid the decision process. The model we present is based on the concept of map waves.

The number of map tasks that a job launches depends on the input file size (the size of the PCAP file in our case). Particularly, the number of map tasks for a given job is equal to the number of input splits. In our setup, the input split size is configured to be equal to the block

size, 128 MB. Thus, it is possible to estimate the number of map tasks that a job will launch by dividing the input file size (the whole file size) by the input split size. So, for example, we can estimate that a 891 MB sized PCAP file will require seven map tasks to process the whole file (seven containers, since each map task is executed in a container)[3]. Besides containers for map tasks, we also need to take the ApplicationMaster into consideration, which requires an additional container and runs throughout the entire job's life cycle. We do not consider the reduce tasks for this estimation because, as mentioned earlier, this job is mainly a map-intensive job with a very short reduce phase. Our local Hadoop cluster consists of eight nodes, one master and seven workers, which translates into 14 containers (each worker node runs two containers at most).

So, if the number of estimated required containers for a job is lower than or equal to our cluster's available 14 containers, that means that the job will be a single map wave job, since all of the job's required containers fit into one wave of 14 containers. Since map tasks run in parallel and almost all of them take approximately the same time to execute in a homogeneous cluster, single map wave jobs have approximately the same duration regardless of how many containers they require within the single wave. The reason for the approximate execution time among map tasks is related to the fact that almost all map tasks process input splits with 128 MB of size. As for the jobs, we will be considering that the map execution time is proportional to the split size. Since the input file size is not necessarily a multiple of 128 MB, the last input split can be smaller than 128 MB. The map task that gets this smaller input split will finish earlier.

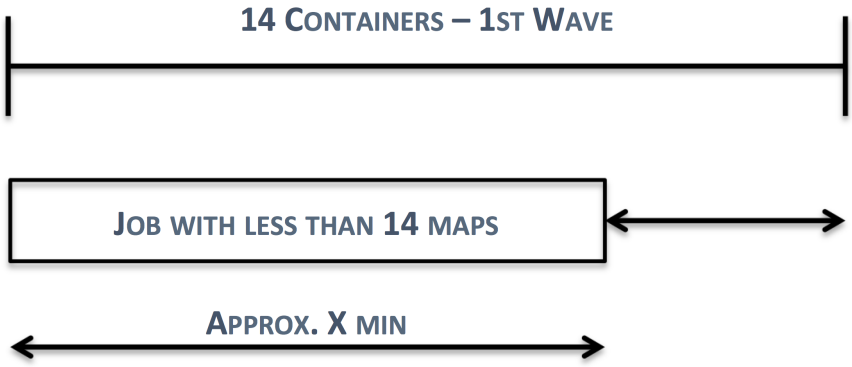

Figure 3.3: The duration of a single map wave job

Figure 3.3 illustrates an example of a single map wave job. Regardless of the amount of required containers for map tasks, the execution time is always the same, as long as the total amount of estimated required containers is lower than or equal to the single map wave container limit. The single map wave container limit is 14 containers or 13 if the ApplicationMaster is taken into consideration. An approximation of the typical behavior of a single map wave job in terms of the number of containers in use during execution time can be seen in figure 3.4. The drop in

[3] Actually, if we divide 891 by 128 we get approximately 6.961, but we can only use an integer number of containers.

container usage at the end of the job's execution reflects the end of the map phase and the start of the reduce phase. The reduce phase is only started at the end of the execution of the job, because we have manually set it to do so. This is particularly important if we want to be able to execute more than one job simultaneously, since we don't want the reducers to occupy available containers while they wait for the mappers to finish.

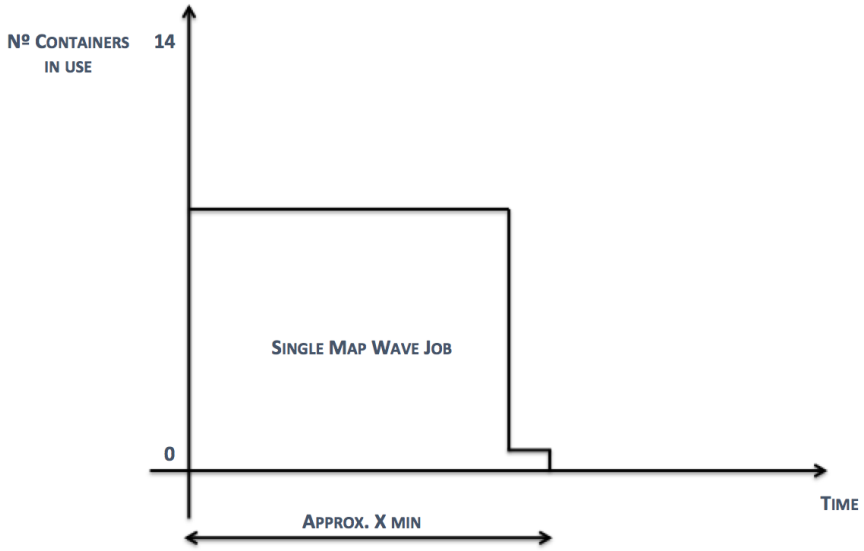

Figure 3.4: The behavior of a single map wave job

Figure 3.4 also helps understand why single map wave jobs have the same duration regardless of the number of containers required for map tasks within the single map wave container limit. All map tasks are started at the same moment and since almost all take approximately the same time, all single map wave jobs have approximately the same duration.

If the number of estimated required containers for a job is greater than the number of containers in our cluster, then the job will be a double or multi map wave job. For example, a 1810 MB sized PCAP File requires 15 containers for map tasks. Since only 14 containers can be fit into our cluster and the ApplicationMaster needs one container throughout the execution of the job, we get 13 containers for maps in the first wave and two containers for maps in the second wave plus two at the end of the execution for the reducers. The job is therefore considered a double map wave job. Double map wave jobs take approximately twice as much the time to execute as it takes for a single map wave job, because the remaining map tasks that run in the second wave are only started when the maps in the first wave finish. Similar to the single map wave job, the double map wave job has approximately the same duration regardless of the number of required containers within the double map wave container limit. The limit is 26 containers, the double of the limit for the single map wave.

Figures 3.5 and 3.6 depict two examples of differently sized double map wave jobs, illustrating

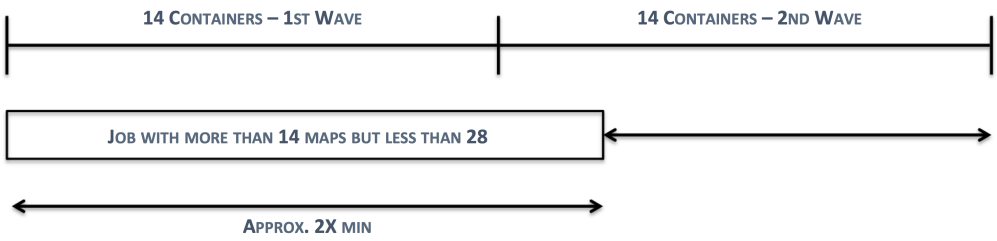

Figure 3.5: The duration of a double map wave job

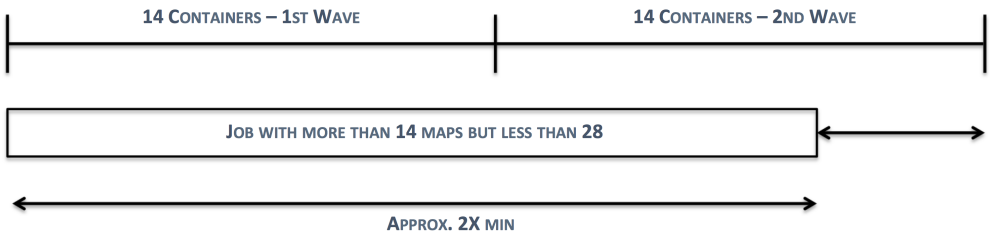

Figure 3.6: The duration of a slightly bigger double map wave job

the explanation regarding the execution time given above. Naturally, the same logic can also be applied to jobs who span more than two map waves. Jobs with three waves will take three times as much to execute as a single map wave job and so forth. The exception to the rule is the situation in which the last wave only has one map and its designated input split is smaller than 128 MB. Map tasks who process less than 128 MB will logically finish faster than those that do. Thus, when this situation occurs, double and multi map wave jobs finish faster than their usual duration. Figure 3.7 depicts the typical behavior of a double map wave job in terms of number of containers in use during its execution. Figure 3.7 makes it is easy to understand why double map wave jobs take twice as much to execute as single map wave jobs. The remaining map tasks that cannot fit into the first wave can only start when the first wave has finished and freed up the containers in use.

The typical behavior shown in figure 3.7 displays a second map wave with less than half the maps found in the first wave, but it's important to understand that this isn't always the case. In fact, the second wave can have as many maps as the first wave, although never more since that would mean that an additional wave would be necessary.

The behavior of multi map wave jobs is very similar to the one found in double map wave jobs. The only difference lies in the fact that the container usage found in the first of wave of the double map wave job is seen multiple times (in multiple waves) in the multi map wave job. All of the remaining characteristics can be extrapolated and applied to jobs with more than two waves.

The focus of this chapter was to introduce in detail our map-intensive network analysis MapReduce job. In order to better understand the context of our network analysis job, this chapter also

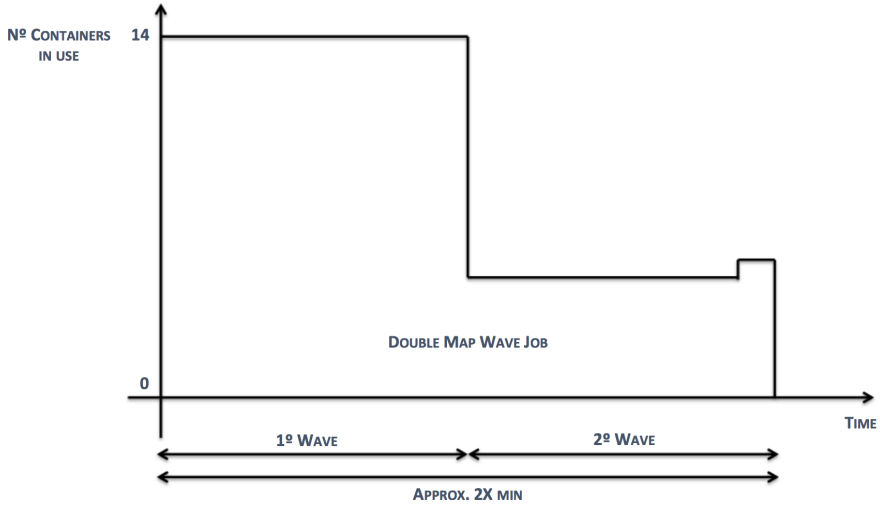

Figure 3.7: The behavior of a double map wave job

covered important background information regarding the physical network topology, how network traffic is captured, what PCAP is, and what types of network analysis exist. The next chapter provides a detailed overview of the architecture of our proposed solution to tackle the implementation of a traffic monitoring system for batched analysis of network traffic, featuring a Cloud Bursting capable Load Balancer.

Chapter 4

Cloud Bursting for Apache Hadoop YARN

This chapter introduces our proposed Cloud Bursting solution for a local private data center running Apache Hadoop YARN to analyze network traffic. It discusses our integrated online monitoring system with batched network analysis, where our Cloud Bursting solution is implemented. It starts by describing the system as a whole and continues by detailing individuals module.

4.1 Introduction

As explained in the previous chapter, the implementation of an online traffic monitoring system with real-time analysis usually requires more powerful hardware to handle the real-time processing and analyzing of data. Our study implements a batched traffic monitoring system using MapReduce jobs in a Hadoop YARN cluster. We use a Network Traffic Gatherer that continuously gathers batches of network traffic and feeds them to the Hadoop cluster. The idea is to launch a small MapReduce job each time a batch of network data is collected. Thus, the size of the launched jobs at a given moment fully depends on the amount of generated network traffic at that moment. If the amount of generated network traffic grows beyond the capacity of our local cluster, analysis of the traffic will be slowed down. So, to guarantee that our infrastructure can handle peaks of network traffic without delaying the analyses, we propose a Cloud Bursting capable infrastructure that automatically bursts workload when the local cluster is overloaded or cannot launch further jobs without delaying them. Figure 4.1 depicts the logical overview of our Cloud Bursting capable network traffic analysis solution.

Our solution collects varying amounts of network traffic and stores them as PCAP files. The PCAP files are then fed to the inter-cluster Load Balancer. The inter-cluster Load Balancer is the key element that powers our Cloud Bursting solution. As batches of network data arrive, the Load Balancer has to decide which cluster is suitable to process the arriving batches. To make this decision, it has to rely on pre-specified criteria or metrics that are configured by an administrator and have threshold values. There are a wide variety of metrics that can be used to determine when

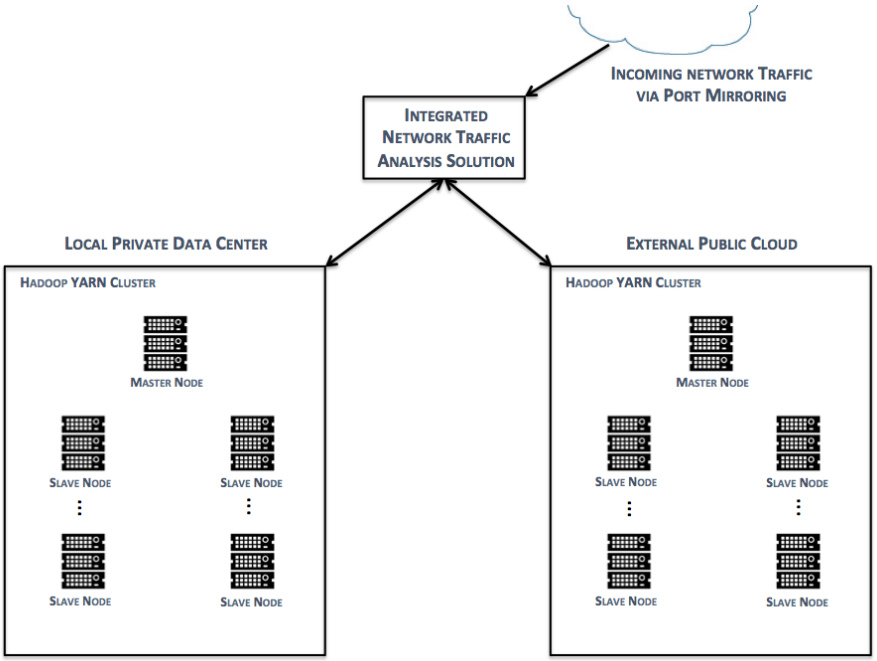

Figure 4.1: Logical overview of the proposed solution

to trigger a burst to the public cloud. The most common involve system-level metrics such as CPU utilization, RAM usage, and disk/network utilization, or application-level metrics such as application response time. For our scenario, our bursting algorithm will focus on four specific metrics: cost of bursting in the sense that bursting to the cloud is more expensive than running the analysis of a batch of PCAP data locally, resource utilization, batch size, and more importantly the guarantee that jobs are not delayed by the simultaneous execution of other jobs.

The fundamental criterion is to try that jobs execute as if the whole cluster was dedicated to them. To ensure that jobs are not delayed, other metrics need to be taken into account. Naturally, resource utilization and network traffic batch size are both fundamental metrics to monitor, as these allow the Load Balancer to estimate if the local launch of the job will not delayed it or if additional remote resources are required. Usage of additional remote resources brings about another important metric: the cost of using remote resources. In this sense, it is crucial to note that usage of local resources should always be preferred, as it incurs no extra cost. The allocation of resources in a public cloud such as Amazon AWS, however, does incur extra costs. Thus, if the execution of a job can be performed without delays using local resources only, the use of resources in a public cloud should be avoided.

In the end, our bursting mechanism will need to consider these metrics and attempt to yield lower delays and fewer number of bursts, thus yielding better local resource usage and lower extra costs.

The next section focuses on detailing the modules that are at the foundation of our solution.

4.2 Integrated Solution

The integrated solution that we developed in this dissertation consists of many crucial modules that will be detailed in the following subsections. Figure 4.2 depicts the whole architecture of our integrated solution.

The Network Traffic Gatherer module uses tcpdump to capture batches of network traffic from a pre-specified network interface and generate PCAP files. Once ready, the PCAP files are copied to a folder where the PCAP File Processor module determines whether or not the PCAP file is new before delivering it to the Load Balancer. Using the cluster resource utilization information provided by the Resource Monitor, the Load Balancer decides whether to launch the job locally or to burst it. In either case, the PCAP file is first uploaded to HDFS on the destination cluster through the HDFS Uploader module. After the upload is complete, the Job Launcher connects to the ResourceManager to launch the map-intensive network analysis job. The following subsections provide a detailed look at each module.

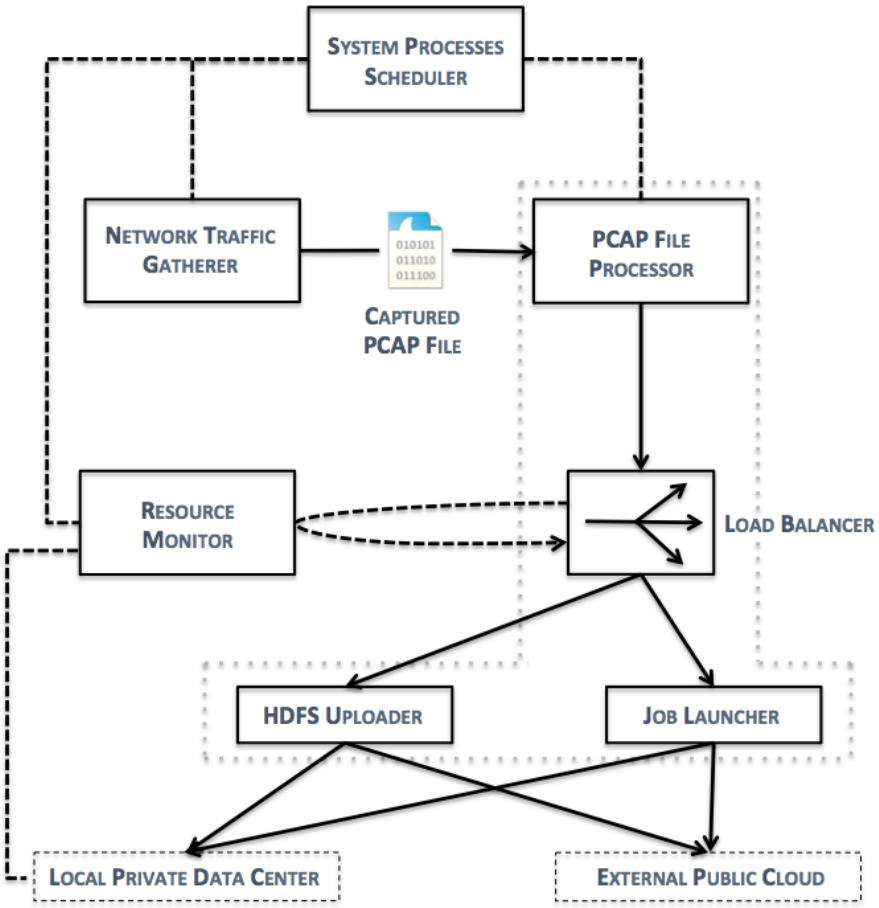

Figure 4.2: Architecture of the developed Integrated Solution

4.3 System Modules

4.3.1 System Processes Scheduler

The System Processes Scheduler is the engine behind our solution. It guarantees that every important function in the system is executed in their respective time frames. The Scheduler is also responsible for launching the system processes (the main functions) in separate threads.

When our Integrated Solution is initialized, it's the System Processes Scheduler that takes over control and starts all other modules. The scheduled system processes are the Resource Monitor, the Network Traffic Gatherer and the PCAP File Processor. The Resource Monitor is executed every second, the Network Traffic Gatherer every 10 seconds, and PCAP File Processor every 15 seconds.

4.3.2 Network Traffic Gatherer

The Network Traffic Gatherer consists of two sub-modules: the TCPDumpStarter and the TCPDumpTransporter. The TCPDumpStarter is a sub-module that is launched in a separate thread to start the tcpdump utility. Tcpdump is a powerful command-line packet analyzer that uses the previously introduced libpcap library to capture packets on a network[31]. Tcpdump can be executed with a variety of different configurable parameters. In our scenario, Tcpdump is executed with the parameters found in table 4.1.

Table 4.1: Configured Tcpdump parameters

Parameter	**Description**
-i *interface*	Interface to listen to. We set ours to en5, which is the ThunderBolt Ethernet on the Mac we used.
-s *snaplen*	Snapshot-length, the number of bytes of data to snarf from each packet. We set ours to zero, defaulting to 65535 B.
-B *buffer_size*	Operating system capture buffer size. Ours is set to 524288 KB (Mac OS X) as indicated by Apple[1].
-G *rotate_seconds*	Rotate the dump file every X seconds. We set ours to rotate every 600 seconds (10 minutes).
-w *file*	Used to specify dump file name. For our solution, we used the following name: `dump_%y%m%d%H%M%S.pcap`. The syntax `%y%m%d%H%M%S` allows for tcpdump to write the precise date in the name, e.g. `dump_150417181706.pcap` (17/04/15 18:17:06)

[1] https://support.apple.com/en-us/HT202013 [Accessed: Mar. 7, 2015]

These parameters allow our solution to capture traffic from a specified network interface and generate PCAP files with names that follow the syntax `dump_%y%m%d%H%M%S.pcap` every *rotate_seconds*. The use of this syntax for the dump file names allows Tcpdump to write on the file name the precise date at which it started the capture. So, for example, a capture that was started on the 17th of April 2015 at 18:17:06 will be stored in a PCAP file with the name `dump_150417181706.pcap`. Our solution is designed to work with this naming scheme. The TCPDumpStarter sub-module launches Tcpdump in a specified directory, where Tcpdump dumps its captures in the PCAP format. These generated PCAP files represent our batches of network data, which are fed to the Load Balancer.

The TCPDumpTransporter sub-module is responsible for taking the newly generated PCAP file and sending it over to the PCAP File Processor, which will then deliver the PCAP file to the Load Balancer. The need for this sub-module has to do with the fact that Tcpdump does not store the capture in progress in a temporary (different) directory, meaning the only way to know that a capture is finished and thus a batch of data is ready to be analyzed is to check if Tcpdump has already created a second PCAP file. The second PCAP file indicates that the first capture is complete. So, the TCPDumpTransporter periodically checks the directory where Tcpdump is dumping its captures to verify if there are two PCAP files. If it doesn't find a second PCAP file, it does nothing and returns. However, if it does find a second file, it takes the file with the oldest date on the file name, moves it to the PCAP File Processor directory and prints a message to the console informing about the success of the move.

4.3.3 PCAP File Processor

The PCAP File Processor is where the main execution flow starts. It's the connector between the Network Traffic Gatherer and the Load Balancer. Figure 4.2 pictures a dashed gray line encompassing not only the PCAP File Processor, but also the Load Balancer, the HDFS Uploader, and the Job Launcher because they are all part of the same execution flow and are executed sequentially in the same thread. The PCAP File Processor has its own directory (to where the TCPDumpTransporter sub-module moves the ready to be analyzed PCAP dump files), which it checks every 15 seconds in order to verify if new PCAP files are ready to be handed over to the Load Balancer. If it doesn't find any file, it prints a message to the console informing that no new files were found. If it does find a file (or more than one), it checks if that file (or the oldest one if more than one is found) has already been uploaded or is being uploaded. If it has been uploaded or is currently being uploaded, the PCAP File Processor prints out a message indicating that the found file is older and that no new files have been found. If it hasn't been uploaded yet, the PCAP File Processor saves the date of the new PCAP file and gets its size. Knowing the size enables the estimation of the number of maps that will be required to process the file, as explained in the previous chapter. An estimation of the number of required containers for this file is calculated and stored. The PCAP File Processor hands a list containing the PCAP file name, size and estimated required number of a containers of the new capture to the Load Balancer. A message is printed to the console informing that a new PCAP File is ready to be uploaded to the HFDS in the local or public data centers.

4.3.4 Resource Monitor

The Resource Monitor is the module responsible for two tasks: monitoring of the local data center's resource utilization, and answering of the Load Balancer's resource utilization queries. It needs to be up to date to guarantee that the Load Balancer makes its decisions based on the most recent cluster resource usage information. The System Processes Scheduler makes sure it gets updated every second.

The Resource Monitor takes advantage of Hadoop's ResourceManager Representational State Transfer (REST) API to get resource usage information. With the REST API, the Resource Monitor is able to retrieve information regarding the number of containers in use, the number of applications in the running state, the number of pending applications, the total number of containers that the cluster has to offer, the progress of running jobs, among other useful information. The following is an example of a request from the Resource Monitor and the respective answer from the ResourceManager:

Listing 4.1: ResourceManager REST API Example

```
$ curl "http://resourceManagerAddress/ws/v1/cluster/metrics"
$ {
              "clusterMetrics":
              {
              "activeNodes": 7,
                  "allocatedMB": 0,
                      "appsCompleted": 4,
                      "appsFailed": 0,
                      "appsKilled": 0,
                      "appsPending": 0,
                      "appsRunning": 0,
                      "appsSubmitted": 4,
                      "availableMB": 28672,
                      "containersAllocated": 0,
                      "containersPending": 0,
                      "containersReserved": 0,
                      "decommissionedNodes": 0,
                      "lostNodes": 0,
                      "rebootedNodes": 0,
                      "reservedMB": 0,
                      "totalMB": 28672,
                      "totalNodes": 7,
                      "unhealthyNodes": 0
              }
      }
```

The answer is formatted using the JavaScript Object Notation (JSON), which is very convenient because it's straightforward to parse in Python. We can see from this example that the cluster has seven active nodes with a total of 28672 MB of available memory, that no applications are pending, no applications are currently running and thus no containers are allocated, meaning that the cluster is completely free.

So, the Resource Monitor sends different requests to retrieve the information it needs, parses the answers and stores the most useful values. These useful values are then available to the Load Balancer to aid it in the decision process.

4.3.5 HDFS Uploader

The HDFS Uploader is the module that uploads a newly captured PCAP file to the HDFS in one of the Hadoop clusters, the private local data center or the public cloud. This module is summoned by the Load Balancer as soon as it has made its decision about whether to burst the job or to launch it locally. The HDFS Uploader takes the newly captured PCAP file, updates a "UploadInProgress" flag, gets the size of the PCAP file and calculates an estimation of the upload time to the HDFS. The estimation is based on the fact that we have prior knowledge of our infrastructure and its limitations, which allows us to have an estimated value for the upload speed. The resulting estimated upload time is presented in the console and the HDFS Uploader starts building the command to start the upload. To upload files to a remote HDFS, our solution uses the REST API provided by HDFS, which supports the complete FileSystem through standard HTTP operations. The following shows the command build process:

Listing 4.2: Source Code for the building of the CURL command to upload a PCAP file to HDFS

```
# Build CURL command to upload file
command = 'curl -i -# -X PUT -L "http://' + cloudAddress +
    ':50070/webhdfs/v1/user/hadoop/pcapFiles/' + pcapFileName +
    '?op=CREATE&user.name=hadoop" -T ' + pcapFileName
```

The variable cloudAddress is then replaced with the address of the destination's Resource-Manager and the pcapFileName variable is replaced with the full name and extension of the PCAP file to be uploaded. The following is an example of a resulting command (after the variables are replaced by real values) and HDFS's answer:

Listing 4.3: Example of a CURL command to upload a PCAP file to HDFS

```
$ [localhost] local: curl -i -# -X PUT -L "http://172.16.3.53:50070/webhdfs/v1/user/
    hadoop/pcapFiles/dump_150409095002.pcap?op=CREATE&user.name=hadoop" -T
    dump_150409095002.pcap
```

Listing 4.4: Example of an answer from HDFS to the CURL command to upload a PCAP file to HDFS

```
1 HTTP/1.1 100 Continue
2
3 HTTP/1.1 307 TEMPORARY_REDIRECT
4 Cache-Control: no-cache
5 Expires: Thu, 01-Jan-1970 00:00:00 GMT
6 Date: Mon, 11 May 2015 09:55:55 GMT
7 Pragma: no-cache
8 Date: Mon, 11 May 2015 09:55:55 GMT
9 Pragma: no-cache
```

```
10 Content-Type: application/octet-stream
11 Set-Cookie: hadoop.auth="u=hadoop&p=hadoop&t=simple&e=1431374155228&s=Mg3/J/
       AhujBiXLXO1WoiSa76Uu4=";Path=/
12 Location: http://LocalDataCenter-Worker4GBRAM2CPUWithLog-003.novalocal:50075/webhdfs/
       v1/user/hadoop/pcapFiles/dump_150409095002.pcap?op=CREATE&user.name=hadoop&
       namenoderpcaddress=LocalDataCenter-Master4GBRAM2CPUWithLog-001:9000&overwrite=
       false
13 Content-Length: 0
14 Server: Jetty(6.1.26)
15
16 HTTP/1.1 100 Continue
17
18 HTTP/1.1 201 Created
19 Cache-Control: no-cache
20 Expires: Mon, 11 May 2015 09:56:00 GMT
21 Date: Mon, 11 May 2015 09:56:00 GMT
22 Pragma: no-cache
23 Expires: Mon, 11 May 2015 09:56:00 GMT
24 Date: Mon, 11 May 2015 09:56:00 GMT
25 Pragma: no-cache
26 Content-Type: application/octet-stream
27 Location: webhdfs://0.0.0.0:50070/user/hadoop/pcapFiles/dump_150409095002.pcap
28 Content-Length: 0
29 Server: Jetty(6.1.26)
```

From line one to 16 we see the request to upload a file get redirected from the ResourceManager to the DataNode where the file data is to be written. When the upload is complete, HDFS answers with the `201 Created` response with the content length equal to zero and the URI (Uniform Resource Identifier) of the uploaded file in the `Location` header, seen in lines 18 to 29[1].

The HDFS Uploader registers the upload start time and upload finish time to keep the estimated upload speed updated and closer to the real value. Each time an upload is successful, the upload speed is updated to reflect the most recent obtained upload speeds. Also, when an upload is successful, the "UploadInProgress" flag is set to zero and the newly captured PCAP file is removed from the local FileSystem since it is no longer needed there, saving space. Finally, the HDFS Uploader prints a success message to the console to indicate that the upload was successful.

4.3.6 Job Launcher

The Job Launcher is the module responsible for establishing Secure Shell (SSH) connections to the ResourceManagers in the local or public data center and submitting the network traffic analysis job. The establishment of SSH connections is considered a better alternative to the use of the ResourceManager's REST API for the submission of jobs, because the use of the command prompt to submit the jobs is more intuitive and simpler than the construction of the corresponding REST command. The only downside is the fact that the job itself (the JAR file) needs to be copied first into the ResourceManager's FileSystem. We believe, however, that this downside is overcome by the fact that it can be easily automated if the need exists.

[1] https://hadoop.apache.org/docs/r1.0.4/webhdfs.html#CREATE [Accessed: Jun. 3, 2015]

This module is summoned by the Load Balancer as soon as the HDFS Uploader is finished uploading the newly captured PCAP file. The Load Balancer specifies the PCAP file name and whether to burst the job to the public Cloud or not. The Job Launcher prepares the SSH properties such as the remote user, the path to the public key for authentication, and the address of the ResourceManager. The Job Launcher then builds the bash command string to launch the MapReduce network analysis job. We wish to terminate the SSH connection as soon as the job has been launched and detach the terminal session in which the execution of the job is running so it can continue to run regardless of the fact that the SSH connection is terminated. After the command is built, the SSH connection is established, the command is executed and the SSH connection is terminated. The Job Launcher registers the job submission time and uses the map wave logic presented in the previous chapter to predict the job completion time. The ability to provide a prediction of the job finish time is also based on prior knowledge of the typical behavior of the job, hence the creation of the map wave model, which allows our solution to estimate the completion time in advance.

The predicted finish time, the information about the number of map waves and number of containers required in the last wave are stored and displayed in the console.

4.4 Load Balancer

The Load Balancer is where the decision process takes place. It's the module responsible for taking advantage of the resource usage information provided by the Resource Monitor, the information about the estimated number of required containers provided by the PCAP File Processor, and the completion time and map wave information provided by the Job Launcher module, to decide whether the analysis of a new PCAP file should be launched locally or burst to the public Cloud.

To develop the Load Balancer, a series of possible real use cases were considered. The use cases are based on the previously explained idea that the Network Traffic Gatherer collects batches of network traffic in fixed time windows, of which the Load Balancer schedules the analyses. The following subsections describe and illustrate how the Load Balancer design started with a simple container usage based algorithm and evolved to a more advanced version, capable of better predicting job completion times and scheduling the execution of more than one simultaneous jobs.

4.4.1 Simple Container Usage based Load Balancer

Figure 4.3 depicts a simple example to better understand the described scenario.

The capture time is the amount of time that the Network Traffic Gatherer spends capturing network traffic before rotating to a new capture file. It is user-configurable and depends on what the Network Administrator believes is adequate for his network and infrastructure. When Tcpdump finishes its current capture, the capture is uploaded and the analysis job is submitted. The total time for the analysis of a batch of data is the sum of the time to upload the PCAP data to HDFS and of the time the job takes to execute. The example displayed in figure 4.3 is also the simplest use case, in which the analyses of the captured network traffic do not exceed their respective capture

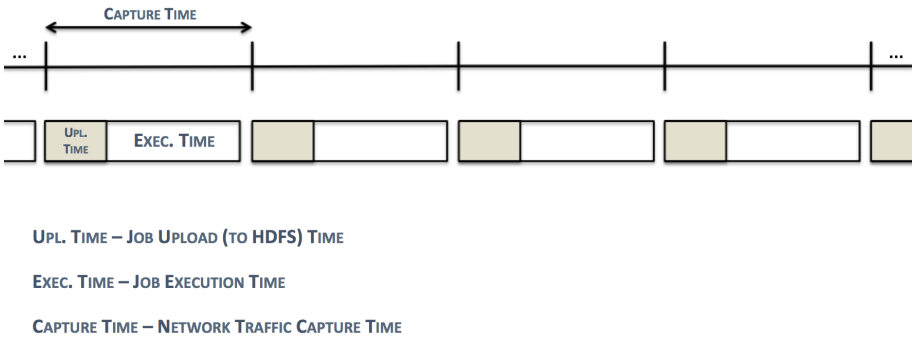

Figure 4.3: Use case 0

time windows. This is the case where no peaks of network traffic are present and all PCAP files are processed in the local data center. There would be no need for a Load Balancer if this always happened. Let us consider the case where a peak in network traffic generates a bigger PCAP file and thus a longer job. Figure 4.4 illustrates what would happen if no Load Balancer was present in this situation.

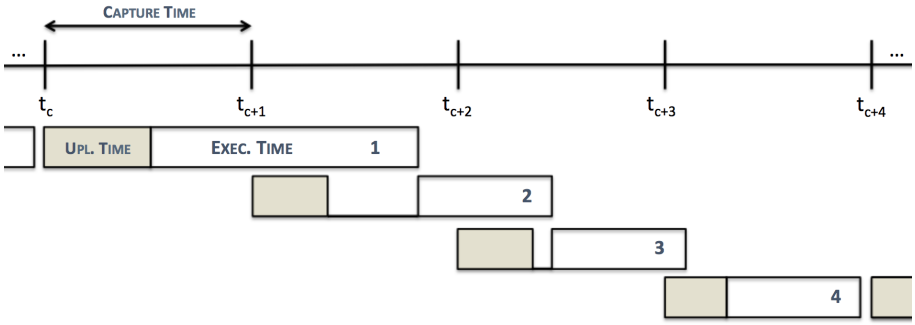

Figure 4.4: Use case 1 without the Cloud Bursting based Load Balancer

It is clear that the execution of job two has to be delayed, because job one is still executing. Job two is uploaded to HDFS but has to wait for available containers to start its execution. This implies that the executing job is using all of the containers in the cluster, which can happen and is the worst scenario. But even in a brighter scenario in which the executing job (job one) is not hogging up all of the containers, job two can still suffer a considerate delay.

So, the first version of our Load Balancer considers the cluster's container utilization before deciding where to upload and process the captured network traffic. When the PCAP File Processor hands a new PCAP file to the Load Balancer, the Load Balancer takes the estimated required containers for the new PCAP file (provided by the PCAP File Processor), the total amount of available containers in the cluster, and checks the average used containers in the last 15 seconds to decide whether to send the new job to the local or public cluster. If the number of free containers in the local cluster is higher than or equal to the number of estimated required containers for the new job, then the new PCAP file is uploaded to the local cluster and the job is launched locally.

If there aren't enough free containers for the new job, then the job is burst to the public cloud. Implementation of this algorithm introduces a problem: double or multi map wave jobs will never be launched locally, even if the cluster is idling. This happens because the estimated required containers for those types of jobs always exceeds the available containers in the cluster. The Load Balancer needs to check if the cluster is idling (no jobs are running) before calculating if it has enough containers for a new job. If the cluster is idling, the job is launched locally regardless of the estimated required amount of containers (which is equivalent to saying regardless of the PCAP file's size). If it's not idling, then the Load Balancer needs to check if the new job can fit in the available amount of containers, that is, if the estimated required containers for the new job is lower than or equal to the number of free containers in the cluster. If the new job fits, then it's launched locally and will execute simultaneously with the one already running in the cluster. If it cannot fit in the available amount of free containers, then the job is burst to the public Cloud.

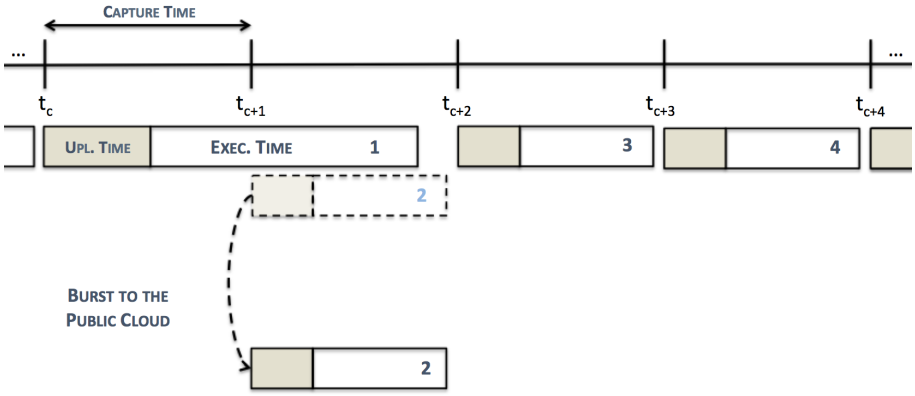

Figure 4.5: Use case 1 with the container usage based Load Balancer

Figure 4.5 depicts use case one with the above described Load Balancer. We now see that job two was burst to the public Cloud to avoid a delay that not only would affect itself (job two), but also job three. As job two arrived, the Load Balancer verified if jobs were running in the local cluster. Since there was already a job running (job one), the Load Balancer checked if it could fit job two into the available free containers. Since the amount of free containers was insufficient or non-existent at the moment of decision, job two was burst. Job three was not delayed as a consequence of the burst.

4.4.2 Adding Job Completion and Upload Time Predictors

Now let us consider the case where the upload of a job takes longer than expected (which could happen, for example, if the network is shared with other services or the captured network traffic is bigger). As seen in figure 4.6, job one is still executing as the instant t_{c+1} is reached (the moment job two is ready to be scheduled). The Load Balancer will see that job one is running in the local cluster and calculate if job two fits in the available free containers. If by chance it does not fit, the Load Balancer will burst it to the public Cloud. It becomes clear from figure 4.6 however, that

bursting job two in this situation does not make any sense, since job one finishes during the upload period of job two. Thus, the need for a way to provide an estimation of the duration of a job and of the duration of the upload time is evident. With the ability to predict job completion and upload times, the Load Balancer is capable of estimating if a running job will finish during the upload period of a new yet to be scheduled job. The estimation of the job completion time is possible due to our prior knowledge of its behavior. In fact, the map wave model described in the previous chapter is the key element that allows the Load Balancer to estimate the duration of a job. So, the estimate for the job completion time can be described as follows:

$$t_{finish} = t_{submit} + N_{waves} \times T_{singlewave}$$

The finish time of a job that has been submitted to the local cluster is equal to the instant at which the job was submitted, plus the average duration of a single map wave multiplied by the number of map waves that the job has. The average duration of a single wave is a static value that we can know in advance after having studied the behavior of the job.

The estimation of the upload duration is achieved by dividing the PCAP file size by the average upload speed. Since the upload speed can vary depending on the utilization of the network, we use an average that is updated with each upload. So, the upload duration can be estimated as follows:

$$t_{upload} = \frac{s_{file}}{v_{upload}}$$

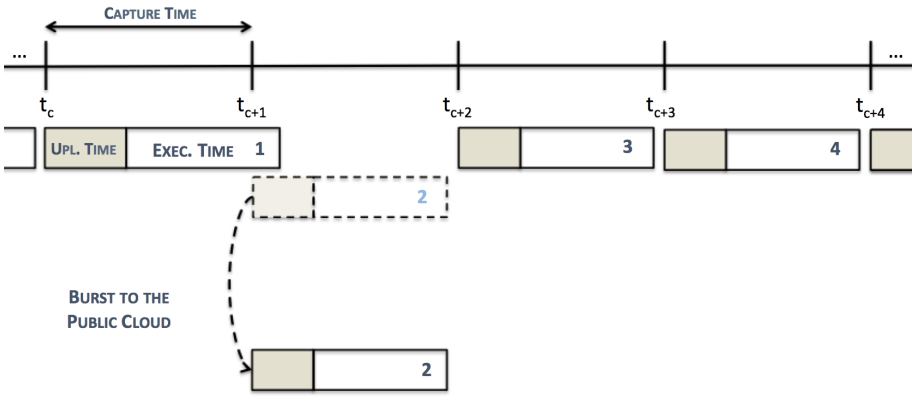

Figure 4.6: Use case 2 with the container usage based Load Balancer

The use of both of these estimators enables the Load Balancer to avoid bursting a new job to the public Cloud in situations where the job that is running in the cluster finishes during the upload period of the new yet to be scheduled job. When a new job arrives, the duration of its upload is estimated. If the Load Balancer verifies that a job is already running in the cluster and the new yet to be scheduled job requires more containers than those that are available, the Load Balancer checks if the job that is running in the cluster ends within the upload period of the new yet to be scheduled job. If it does, the new PCAP file is uploaded to the local HDFS and the new job is scheduled to the local data center. If it doesn't, the new job is burst to the public Cloud. Figure 4.7 depicts use case two with the use of the above mentioned estimators.

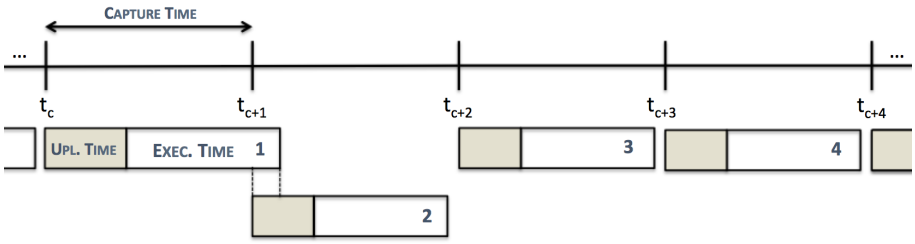

Figure 4.7: Use case 2 with the container usage based Load Balancer with Upload and Job Completion Times Estimators

Notice that job two is not burst to the public Cloud because job one finishes within the upload period of job two, meaning that job two can be scheduled to the local cluster without suffering delays. Furthermore, it is important to note that we have defined the threshold value to 70 seconds after the end of the upload period. Setting the threshold after the upload period has proven to avoid bursts in specific situations. We will present some of these situations in the next chapter. This advantage comes at the cost of a small delay in the job to be scheduled that can go up to 70 seconds, but is usually less. We believe this delay is a reasonable trade-off, considering its maximum size when compared to the full duration of a job. Nevertheless, as illustrated in figure 4.8, the threshold is user configurable and can be extended or reduced.

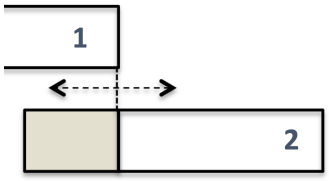

Figure 4.8: Upload Period Tolerance

4.4.3 Scheduling a Single Map Wave Job with a Double/Multi Map Wave Job

We will now consider the case where a double or a multi map wave job does not use all of the containers in the second wave. Figure 4.9 depicts this case with a double map wave job (job one) that is followed by a single map wave job (job two). We can see that the first wave of job one finishes within the upload period of job two, releasing several containers, since the second wave has less maps. The Load Balancer bursts job two because the last wave finishes after the upload period. However, it becomes clear from figure 4.9 that there might be cases where it is possible to execute two jobs in simultaneous. In situations similar to the one presented in 4.9 where job two is a single map wave job and requires as many or less containers as the number of containers that are free during the execution of the last wave of job one, job two should be scheduled to execute simultaneously with job one. Particularly, it makes sense to execute them simultaneously because

job two will not be delayed by the presence of job one, as it has enough free containers to keep it a single map wave job.

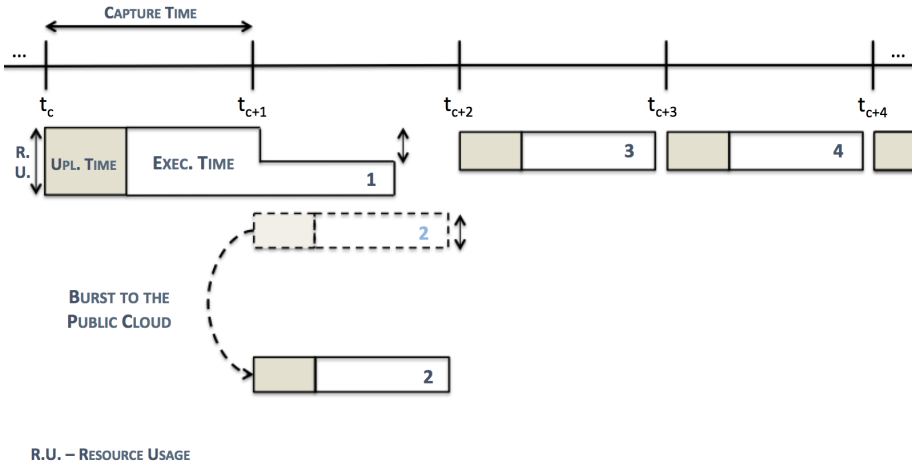

Figure 4.9: Use case 3 with the container usage based Load Balancer with Upload and Job Completion Times Estimators[2]

To avoid the situation presented in figure 4.9, the Load Balancer checks if the last wave of job one ends within the upload period of job two. If it doesn't, it verifies if the wave before the last one (of job one) ends within the upload period (of job two) and simultaneously if the container requirements of job two fit into the amount of free containers during the last wave of job one. If both of the verifications happen, job two is scheduled to the local cluster. The PCAP file is uploaded to the local HDFS while job one is still executing, and eventually job two is launched and executes in parallel with job one. Figure 4.10 depicts the use case found in figure 4.9 with the new feature explained above.

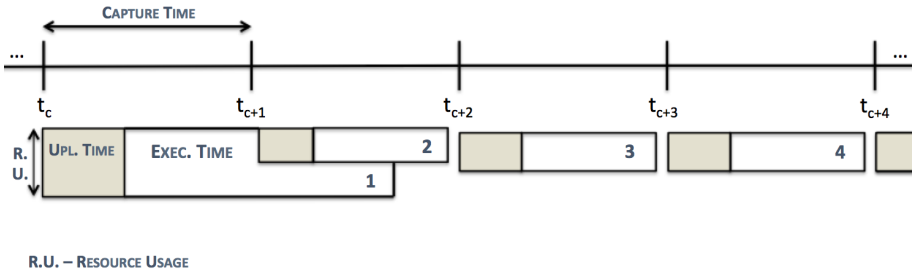

Figure 4.10: Use case 3 with the container usage based Load Balancer with Upload and Job Completion Times Estimators updated for the execution of concurrent jobs

If the upload of job one takes longer than usual and the first wave is pushed passed the upload period of job two, or job one processes a bigger PCAP file and requires more containers in the second wave, leaving insufficient free containers for job two, then the Load Balancer bursts job

[2]The resource usage information is only applied to the execution of the job and does not apply to the upload portion displayed in the figure.

two to the public Cloud to avoid the delay that would be introduced if it were launched locally. Figure 4.11 illustrates the case where the second wave of job one does not free enough containers for job two, forcing job two to be burst.

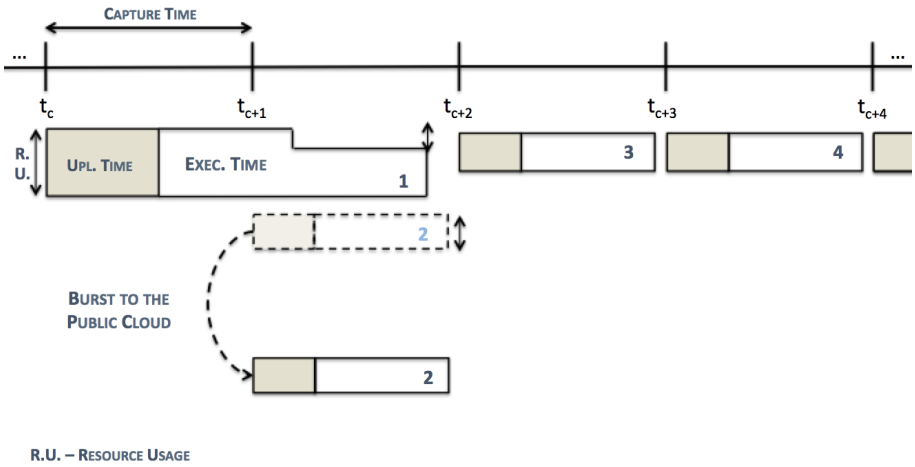

Figure 4.11: Use case 4 with the container usage based Load Balancer with Upload and Job Completion Times Estimators updated for the execution of concurrent jobs

4.4.4 Scheduling a Double/Multi Map Wave Job with another Double/Multi Map Wave Job

Let us now consider the case where a peak of network traffic spans two capture time windows, generating two sequential double map wave jobs, as seen in figure 4.12.

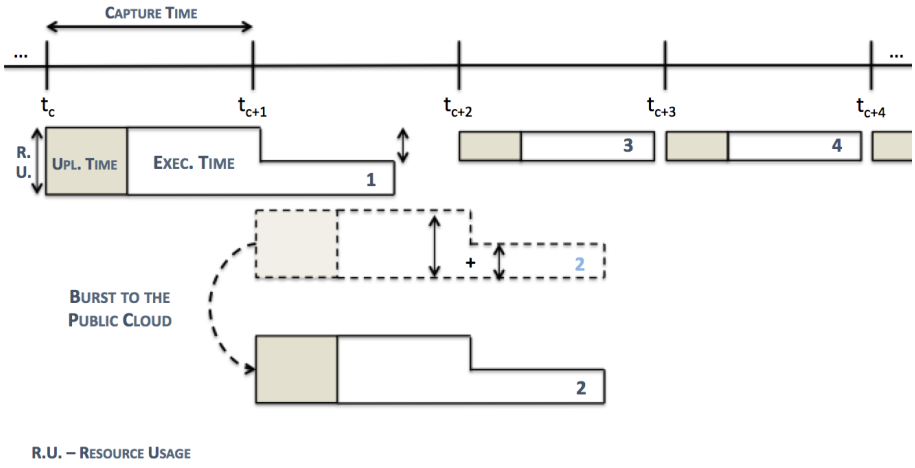

Figure 4.12: Use case 5 with the container usage based Load Balancer with Upload and Job Completion Times Estimators updated for the execution of concurrent jobs

When job two is ready to be scheduled, the Load Balancer looks at job one and checks if the wave before the last one ends within the upload period. If it does, it checks if the estimated amount

of required containers for job two is equal to or lower than the free containers during the second wave of job one. Since job two is a double map wave, its amount of estimated required containers will always be higher than the free containers during the second wave of job one. Thus, job two is burst. However, we can see from figure 4.12 that the last wave of job two requires less than or the same amount of containers that are free during the last wave of job one. This means that it is possible to fit the last wave of job two in the last wave of job one, meaning job two can be launched locally without delaying it, since it still remains a double map wave job according to our simple map wave model. So, when a double or multi map wave job is running in the cluster and another double or multi map wave job arrives, the Load Balancer needs to look at the running job and check not only if the wave before the last one ends within the upload period of the job to be scheduled, but also if the last wave of the yet to be scheduled job requires less than or the same amount of containers that are available during the last wave of the running job.

Figure 4.13 illustrates how the Load Balancer schedules job two with the features mentioned above.

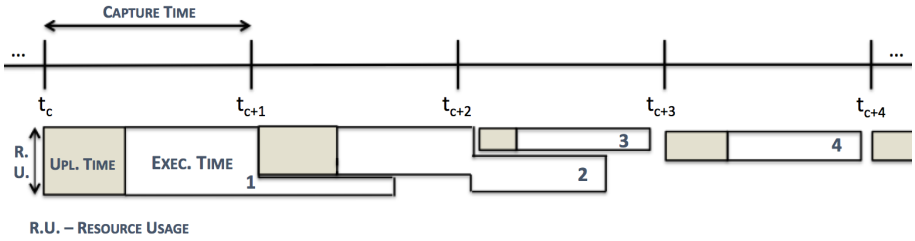

Figure 4.13: Use case 5 with the container usage based Load Balancer with Upload and Job Completion Times Estimators updated for the execution of concurrent single and multi map wave jobs

As seen in figure 4.13, it is possible to launch job two without delaying it, thus avoiding an unnecessary burst to the public Cloud. It is important to note, however, that there are differences between the map wave model's predicted behavior and the real behavior of job two for this specific case. Figure 4.14 depicts both versions of the behavior of job two.

For this situation, the map wave model provides an approximation of the real behavior, where the predicted last wave is modeled after the peak in the last wave of the real behavior. Despite the difference, the map wave model's prediction provides a reasonable indication regarding whether or not the next yet to be scheduled job will be delayed or not. Essentially, if the next yet to be scheduled job is a single map wave job, two conditions need to be checked: the wave before the last wave of the running job needs to end within the upload period of the yet to be scheduled job, and the amount of free containers during the predicted last wave of the running job needs to be equal to or more than the amount of required containers for the yet to be scheduled job. If these conditions are met, then the single map wave job that follows can be launched locally without delays.

If the job that follows is a double or multi map wave job, however, then to launch the job locally without delays, the same conditions need to be verified, except for the fact that the amount

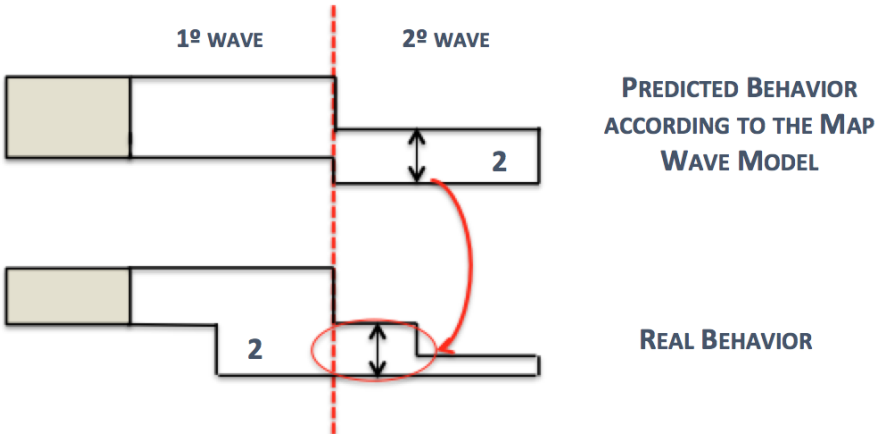

Figure 4.14: Difference between the map wave model's behavior prediction and the real behavior for the situation presented in use case 5

of free containers during the predicted last wave of the running job need to be equal to or more than the amount of required containers in the last wave of the double or multi map job that follows. If these conditions are not met, the Load Balancer bursts the job to avoid the possibility of delaying the job.

4.4.5 Dealing with Uploads that exceed their Capture Time Window

Finally, we consider the case where a large peak of network traffic is captured and the upload period exceeds its capture time window as a consequence. Figure 4.15 depicts the case where the network traffic captured at instant t_c exceeds its capture time window and enters the capture time window started at t_{c+1}, generating a large job with three map waves (job one).

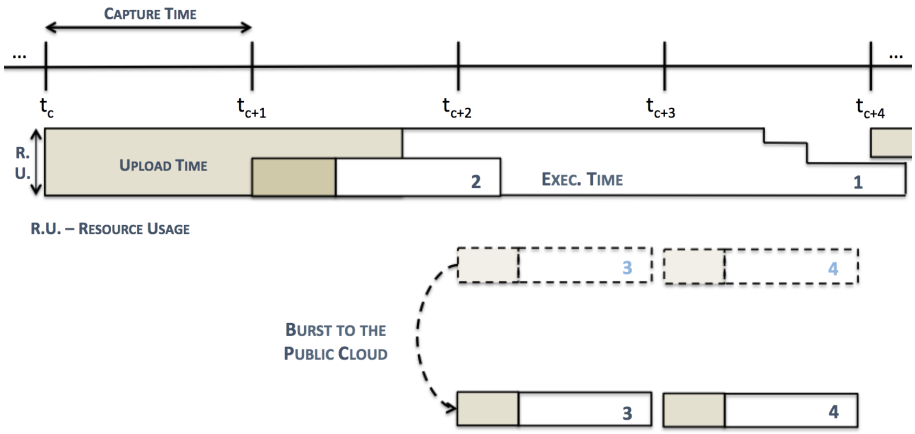

Figure 4.15: Use case 6 with the container usage based Load Balancer with Upload and Job Completion Times Estimators updated for the execution of concurrent single and multi map wave jobs

Since job one is in a pending state at the instant t_{c+1}, waiting for the upload of its PCAP file to the local HDFS, the Load Balancer launches job two locally. The problem that follows, however, is the fact that job two starts during the upload period of job one, hogging up a portion of the containers and delaying job one. Also, the upload of both jobs is delayed as well, because the available bandwidth is shared between both. To avoid this situation, the Load Balancer needs to check if there are uploads in progress when it finds that there are no jobs running in the cluster. If there are no uploads in progress, the Load Balancer launches the job locally. If, on the other hand, there are uploads in progress, then the Load Balancer bursts the job to the public Cloud. Figure 4.16 displays the use case found in figure 4.15 with the feature regarding the verification of uploads in progress mentioned above.

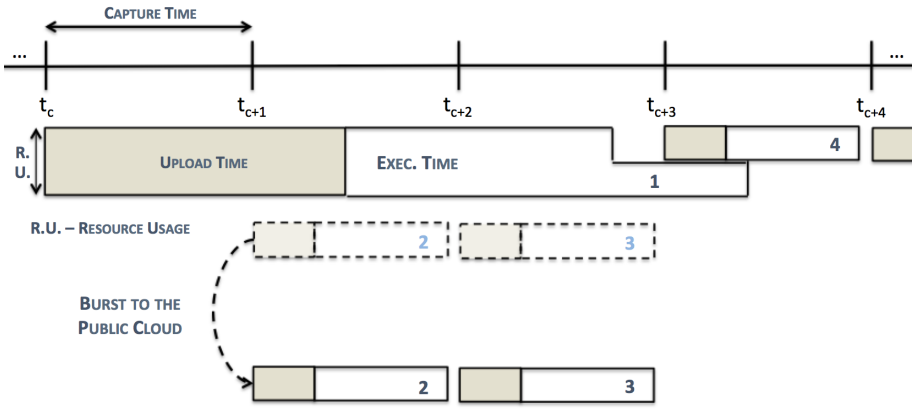

Figure 4.16: Use case 6 with the container usage based Load Balancer with Upload and Job Completion Times Estimators updated for the execution of concurrent single and multi map wave jobs, and with uploads in progress verification

Even though the amount of bursts remains the same when compared to the prior version of the

Load Balancer, notice how neither job one, nor the upload of job one and two are delayed .

This chapter provided a detailed overview of how our proposed solution is built and how it should work for several possible use cases. The next chapter focuses on demonstrating how our solution performs on our local Hadoop cluster with different types of captured network traffic, which include traffic generated with iPerf and traffic captured on the main router of the network labs at FEUP.

Chapter 5

Results

The focus of this chapter is to demonstrate the performance of our proposed solution for a set of predefined use cases. It starts by introducing the template for the presentation of results, followed by a brief description of the cluster we use for the benchmarks. Afterwards, we illustrate a possible outcome when a large peak of traffic is captured and the local infrastructure does not have a Cloud Bursting capable Load Balancer to handle the scheduling of jobs to avoid delays. Finally, we compare the simplest version of our Load Balancer to the more advanced one, by running the predefined uses cases with both versions.

The first set of performance tests uses the job that simulates processing on network traffic generated by Iperf, an application typically used to measure the maximum achievable throughput on IP networks. Since Iperf[1] supports tuning of various parameters, we are able to generate UDP streams with throughput values of our choice, allowing us to easily create different test cases. The second set uses the signature matching job on real network traffic, captured at the router in our network labs. Finally, the last set uses a heterogeneous cluster instead of a homogeneous one.

5.1 Template for the Presentation of Results

In order to better understand the results that will be presented, we begin by introducing the template we use to the display the results. Figure 5.1 depicts the template.

While most of the content in the template is already familiar from the previous chapter, there are a few details that need to be mentioned. The capture time window (explained in the previous chapter) is represented by the top horizontal double arrow. Its duration, in minutes, is given by the variable T and is user configurable. For our use cases, we set the duration of the capture time window to 10 minutes. During each capture time window, network traffic average throughput is given in Megabits per second (Mbps) located next to the boundaries of the time windows. In figure 5.1, the throughput values for each capture time window are X, Y and Z Mbps.

[1] https://iperf.fr [Accessed: Jun. 4, 2015]; http://software.es.net/iperf/index.html [Accessed: Jun. 4, 2015]

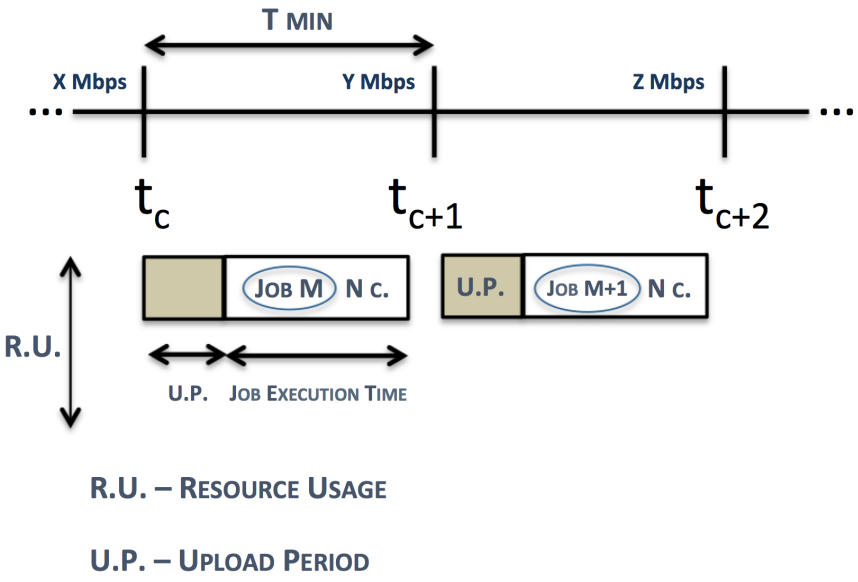

Figure 5.1: Template for the presentation of the results

Notice that the analysis job that is submitted at t_c refers to the network traffic captured during the previous capture time window, started at t_{c-1} (not visible in the figure). So, for example, the traffic generated with X Mbps of throughput during the time window starting at t_{c-1} is uploaded to HDFS and analyzed during the capture time window starting at t_c. Furthermore, the variable N next to the job number M indicates the amount of required containers for that specific job.

The performance tests that follow were performed in one hour time intervals, translating to six 10 minute capture time windows. In addition, we assume that our network has a dedicated line to the public Cloud, which is at least as powerful as the local cluster. In other words, we assume that the jobs that are burst to the public Cloud never suffer delays.

Before introducing the outcome of the performance tests, the next section provides some details on the integration of the map-intensive network analysis job with our local Hadoop cluster.

5.2 The Map-Intensive Network Analysis Job in our Local Hadoop Cluster

The physical machines which house the Hadoop nodes share the same hardware specifications, classifying our cluster as homogeneous. Each machine has four GB of RAM, a quad-core processor, and a 500 GB hard drive. Eight nodes allows us to have seven worker/slave nodes and one master node. We configured the YARN and MapReduce memory settings according to the guide provided by Hortonworks[2], according to which our worker nodes should run a maximum of two

[2] http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.0.6.0/bk_installing_manually_book/content/rpm-chap1-11.html [Accessed: March. 16, 2015]

containers with a maximum of two GB of memory each. Thus, our local cluster has a total of 14 available containers, which translates into 13 containers as the limit for a single map wave job. If a job requires more than 13 containers for its maps, then the job is either a double map wave job (if it needs less than or exactly 26 containers for maps) or a multi map wave job (if it needs more than 26 containers for maps). So, assuming the cluster's network is not shared, our cluster is in steady state, i.e. there are no overlapping jobs, if the captured PCAP files generate jobs that need 13 or less containers for maps. In other words, in ideal conditions and to maintain steady state, the PCAP file size should not exceed the input split size (128 MB) times 13, which equates to 1664 MB.

Moreover, after a number of executed jobs, we concluded that the average duration of a map wave is six minutes and 15 seconds. Single map wave jobs finish on average after six minutes and 15 seconds, double map wave jobs finish after 12 minutes and 30 seconds, and so forth. We have also observed that the mean absolute error of our estimation is 24.30 seconds, which is a reasonable value considering the duration of the jobs. Finally, we measured the upload speed and determined that the starting value [3] should be 36 Mbps. While this value might seem low considering the FastEthernet links, note that each block is replicated three times, meaning that the upload speed is not necessarily indicated by the speed of the physical network interfaces. Also, the upload speed can vary depending on network utilization, execution of other jobs, and whether or not other files are being uploaded. For these reasons, the upload speed is updated every time an uploaded is successful.

The next section illustrates the outcome of the case where a high peak of network traffic occurs and the local cluster is not equipped with the cloud bursting capable Load Balancer.

5.3 Peak without Load Balancer

We now present the case where our local Hadoop cluster is processing a steady 12 Mbps stream and an unexpected peak of 58 Mpbs occurs during the capture time window started at t_c. Figure 5.2 illustrates this situation using the template mentioned in the previous section with a slight difference.

Job one, which processes the traffic captured from t_{c-1} to t_c, is a single map wave job with eight containers, whose upload and processing occurs during the capture time window started at t_c. Thus, since job one occurs during its respective capture time window, it is not affected by the peak that follows at t_{c+1} and finishes in time. At t_{c+1}, the 58 Mbps stream of data is ready to be analyzed (job two) and is thus uploaded to HDFS. A 10 minute 58 Mpbs stream creates a PCAP file with approximately 4320 MB, which in our cluster has shown to take between 13 to 17 minutes to upload. As depicted in figure 5.2, the upload of the data for job two exceeds its respective capture time window, entering the next one (t_{c+2}), where the data for job three (captured during the capture time window started at t_{c+1}) is ready to be uploaded. As a consequence, the upload of the data for

[3] The value our solution uses as a reference when it is started

job three is overlapped by the upload of the data for job two, delaying both uploads. Job three's upload finishes earlier than that of job two. Thus, job three, which requires eight containers, is submitted first and uses the available resources it needs for the ApplicationMaster (AM) and the maps. Approximately three minutes and a half later, job two is finally submitted and takes the remaining resources to begin processing its 35 maps. With a 4320 MB sized PCAP file, job two is a three map wave job with nine maps in the last wave. Note that according to the logic explained in the previous chapter, 10 containers (nine maps and one AM) do not fit in the remaining six containers left by job three during its execution. Hence, job two is delayed approximately two minutes and 55 seconds. Likewise, job three is also delayed circa three minutes and 30 seconds due to the presence of job two, which hogs up the containers that job three needs to start its short reduce phase.

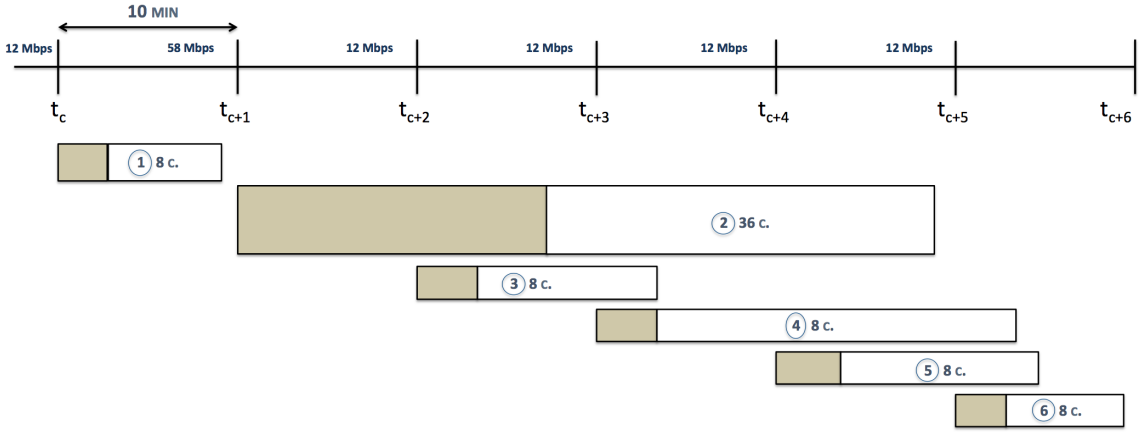

Figure 5.2: Example of the delay that can be caused by a peak in network traffic in our Hadoop cluster when no Load Balancer is present

At t_{c+3} the upload of the PCAP file for job four is started, while job two and three are executing. Despite the fact that job three finishes nearly at the same time as the submission of job four, job two is still executing and is using all of the available containers as job four is submitted. As a consequence, job four is delayed about 13 minutes and 55 seconds, which is a massive delay considering that job four is a single map wave job. At the next capture time window (starting at t_{c+4}) job four and two are both still executing and are using all available containers when job five is submitted. Job two starts releasing containers as its maps are finishing and eventually finishes during job five's capture time window, making room for job five to execute. Nevertheless, job five still suffers a considerate delay of about five minutes and 46 seconds. Finally, at t_{c+5} job four and five are reaching the end of their execution, freeing up enough containers for job six to execute normally and without delays.

It becomes clear from this example that in order to avoid the possibility of delaying jobs, the need for a Cloud Bursting capable Load Balancer is evident. The following sections compare the performance of our more advanced version of the Load Balancer to our simplest version.

Table 5.1: Job details for figure 5.2

Job Number	Average Throughput During Capture	Upload Start Time	Job Submit Time	Predicted Job Finish Time	Job Finish Time	Average Duration / Effective Duration
Job 1	12 Mbps	13:50:26	13:53:09	13:59:24	13:59:39	6 min. 15 sec. / 6 min. 20 sec.
Job 2	58 Mbps	14:00:13	14:17:16	14:36:01	14:38:56	18 min. 45 sec. / 21 min. 26 sec.
Job 3	12 Mbps	14:10:15	14:13:41	14:19:56	14:23:39	6 min. 15 sec. / 9 min. 58 sec.
Job 4	12 Mbps	14:20:17	14:23:40	14:29:55	14:43:50	6 min. 15 sec. / 13 min. 38 sec.
Job 5	12 Mbps	14:30:19	14:33:55	14:40:10	14:45:56	6 min. 15 sec. / 8 min. 53 sec.
Job 6	12 Mbps	14:40:22	14:43:15	14:49:30	14:49:59	6 min. 15 sec. / 6 min. 31 sec.

5.4 Simple Load Balancer vs. Advanced Load Balancer

The purpose of this section is to compare the performance of the simple version of the Load Balancer with the more advanced version. The simple version is based on the availability of containers at the beginning of each capture time window, which is when the captures are ready to be uploaded. The more advanced version has all of the features explained in the previous chapter. The results of the performance tests are presented in the subsections that follow.

5.4.1 Synthetic Traffic Analysis with the Simulation Job

The first set of performance tests uses the job that simulates processing in the map phase on synthetic traffic generated by Iperf.

5.4.1.1 Test Case 1

In this test case, illustrated by figures 5.3, 5.4 and their respective detail tables 5.2, 5.3, the local cluster starts off in steady state capturing and analyzing a stream of 12 Mbps, when at t_c the throughput is increased to 20 Mbps. Notice that the increase in throughput still generates a single map wave job. The difference lies in the upload period, which is longer in the 20 Mbps case. This test case illustrates the use case presented in the previous chapter in figures 4.6, 4.7.

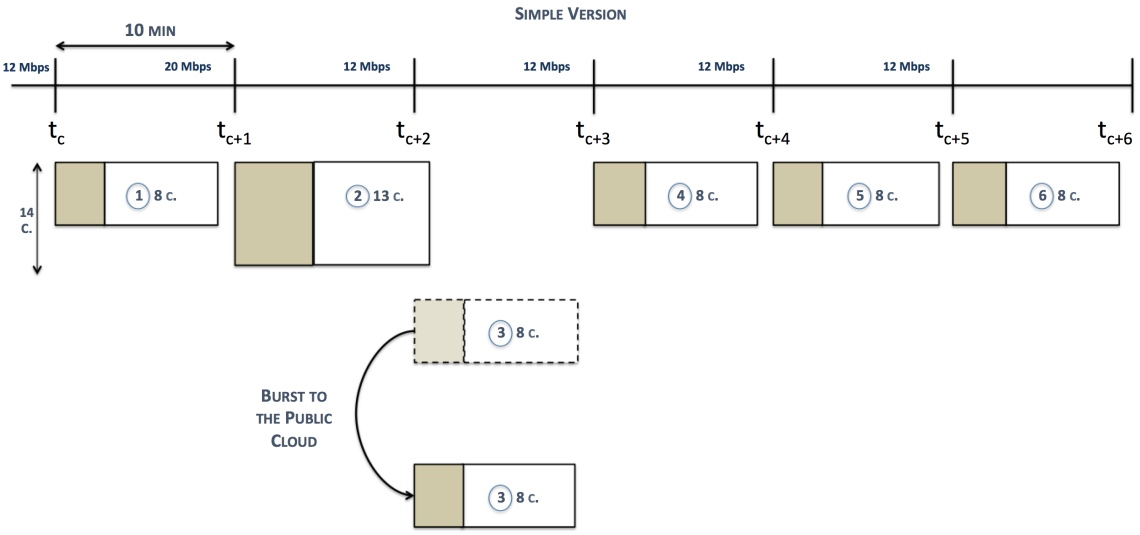

Figure 5.3: Test case 1 with the simple version of the Load Balancer

Table 5.2: Job details for figure 5.3

Job Number	**Average Through-put During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	12 Mbps	14:06:55	14:09:41	14:15:56	14:16:09
Job 2	20 Mbps	14:16:42	14:21:05	14:27:20	14:27:10
Job 3	12 Mbps	14:26:47	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 4	12 Mbps	14:36:47	14:39:41	14:45:56	14:46:13
Job 5	12 Mbps	14:46:49	14:49:26	14:56:41	14:55:53
Job 6	12 Mbps	14:56:52	14:59:28	15:05:43	15:06:12

The simple version of the Load Balancer bursts job three to the public Cloud because there is only one free container at the moment (t_{c+2}) it needs to decide where to upload and launch the yet to be scheduled job three. We easily conclude from figure 5.3 that it does not make sense to burst job three, since the execution of job two ends within the upload period of job three, leaving the execution of job three unaffected. Notice how the more advanced version of the Load Balancer launches job three in the local cluster. It is able to to do so, because it is capable of predicting

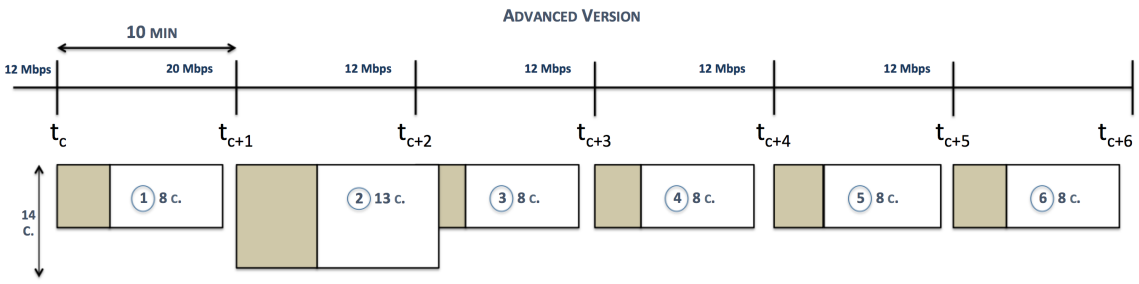

Figure 5.4: Test case 1 with the advanced version of the Load Balancer

Table 5.3: Job details for figure 5.4

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	12 Mbps	15:24:39	15:27:34	15:33:49	15:34:15
Job 2	20 Mbps	15:34:26	15:38:56	15:45:11	15:45:39
Job 3	12 Mbps	15:44:29	15:47:35	15:53:50	15:54:11
Job 4	12 Mbps	15:54:31	15:57:11	16:03:26	16:03:42
Job 5	12 Mbps	16:04:33	16:07:20	16:13:35	16:13:24
Job 6	12 Mbps	16:14:35	16:17:34	16:23:49	16:24:02

the finish time of job two to check if it ends within the predicted upload period of the yet to be scheduled job three.

The more advanced version clearly outperforms the simple version in this test case, avoiding a needless burst to the public Cloud.

5.4.1.2 Test Case 2

In this test case, illustrated by figures 5.5, 5.6 and their respective detail tables 5.4, 5.5, the local cluster starts off in steady state capturing and analyzing a stream of 12 Mbps, when at t_c a peak of 25 Mbps occurs, generating a double map wave job that is submitted during the capture time window started at t_{c+1}. This test case illustrates the use case presented in the previous chapter in figures 4.9, 4.10.

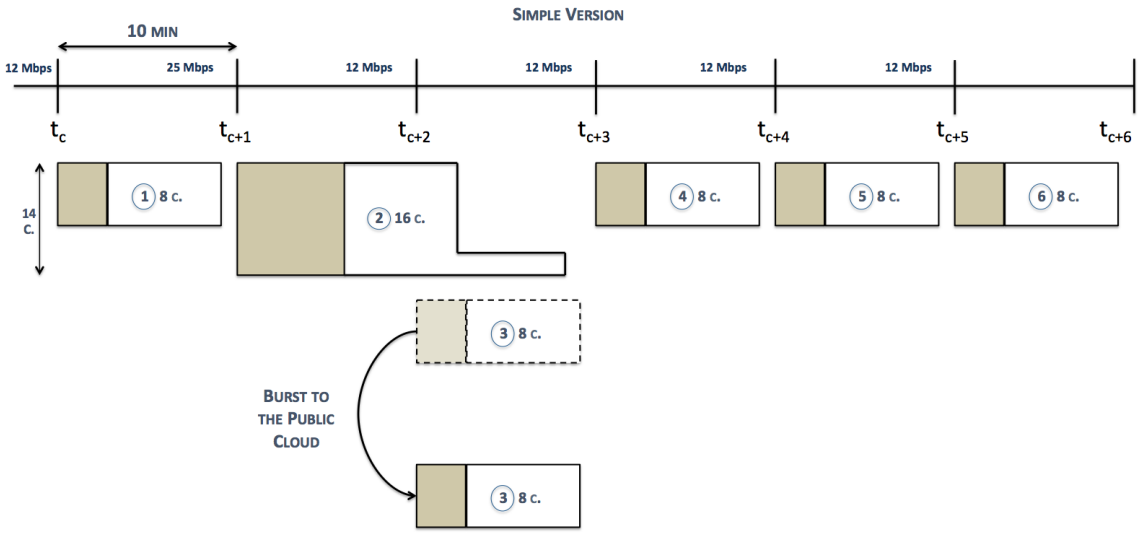

Figure 5.5: Test case 2 with the simple version of the Load Balancer

Table 5.4: Job details for figure 5.5

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	12 Mbps	09:39:17	09:41:56	09:48:11	09:47:54
Job 2	25 Mbps	09:49:04	09:54:52	10:07:22	10:06:40
Job 3	12 Mbps	09:59:10	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 4	12 Mbps	10:09:09	10:11:52	10:18:07	10:18:19
Job 5	12 Mbps	10:19:11	10:21:47	10:28:02	10:27:50
Job 6	12 Mbps	10:29:13	10:31:52	10:38:07	10:38:43

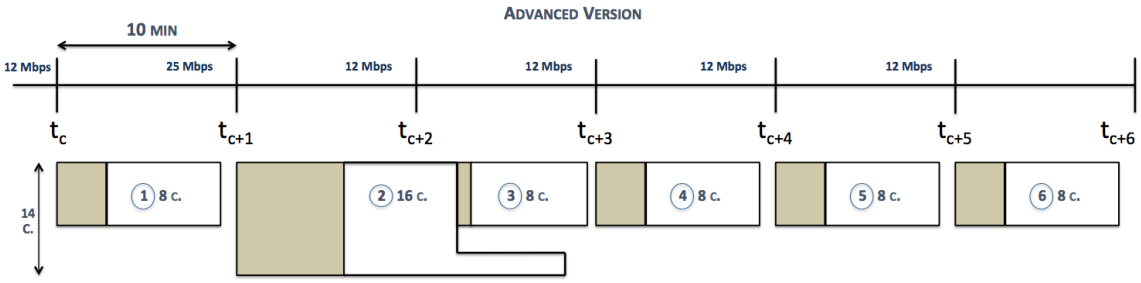

Figure 5.6: Test case 2 with the advanced version of the Load Balancer

Table 5.5: Job details for figure 5.6

Job Number	**Average Through-put During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	12 Mbps	10:59:20	11:02:02	11:08:17	11:07:58
Job 2	25 Mbps	11:09:08	11:14:47	11:27:17	11:27:18
Job 3	12 Mbps	11:19:11	11:22:19	11:28:34	11:28:39
Job 4	12 Mbps	11:29:13	11:31:47	11:38:02	11:38:49
Job 5	12 Mbps	11:39:16	11:41:59	11:48:14	11:49:31
Job 6	12 Mbps	11:49:18	11:52:02	11:58:17	11:58:24

Notice how the simple version of the Load Balancer bursts job three to the public Cloud, even though there would have been enough free containers at the moment job three would be submitted if it were launched locally. Since all of the containers are being used at t_{c+2}, the simple Load Balancer decides to burst job three. On the other hand, the more advanced version verifies that the first wave of job two ends within the predicted upload period of job three, and that the amount of free containers in the last wave of job two is enough to guarantee that job three is not delayed.

We conclude that test case two is another situation in which the more advanced version outperforms the simple version.

5.4.1.3 Test Case 3

Test case three covers the case where the local Hadoop cluster is in steady state and two peaks of network traffic are captured during two consecutive capture time windows (t_c and t_{c+1}), spawning two consecutive double map wave jobs. Figures 5.7 and 5.8 alongside with the respective details tables 5.6, 5.7 depict this situation. This test case refers to the use case explained in the previous chapter and illustrated by figures 4.12, 4.13.

Once more, we see that the simple version of the Load Balancer bursts job three to the public Cloud because it verifies that all containers are being used at t_{c+2}. As explained in the previous chapter, there is no need to burst job three in this situation because, as seen in figure 5.7, the amount of free containers during the second wave of job two is enough to fit the second wave of

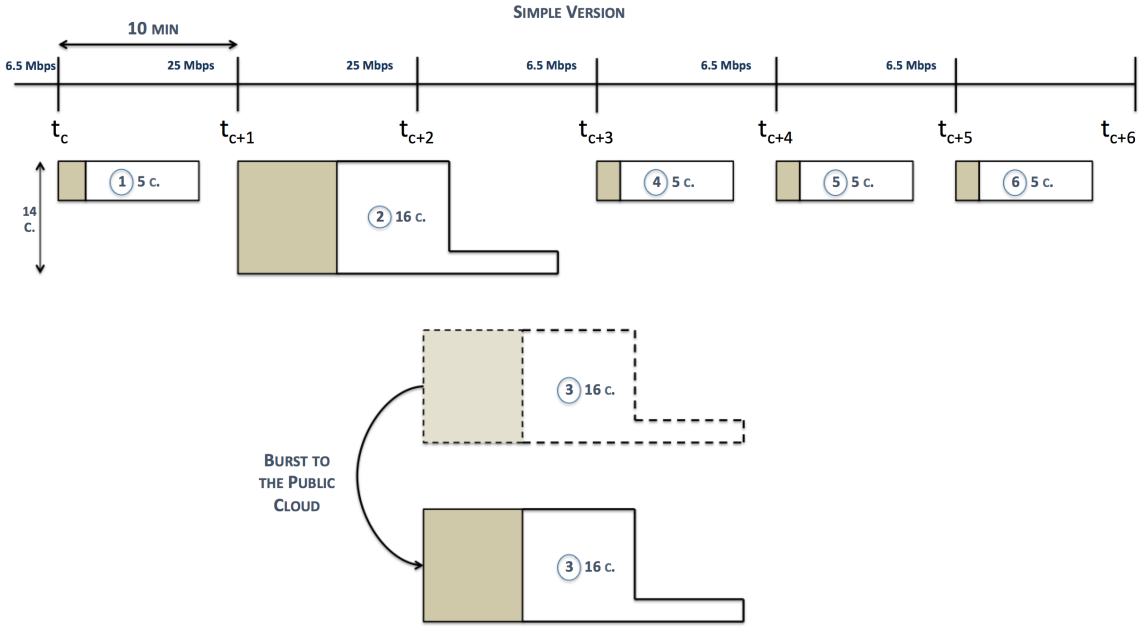

Figure 5.7: Test case 3 with the simple version of the Load Balancer

Table 5.6: Job details for figure 5.7

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	6.5 Mbps	10:00:01	10:01:29	10:07:44	10:07:29
Job 2	25 Mbps	10:09:48	10:15:14	10:27:44	10:27:32
Job 3	25 Mbps	10:19:50	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 4	6.5 Mbps	10:29:53	10:31:15	10:37:30	10:37:48
Job 5	6.5 Mbps	10:39:55	10:41:20	10:47:35	10:47:20
Job 6	6.5 Mbps	10:49:58	10:51:27	10:57:42	10:58:24

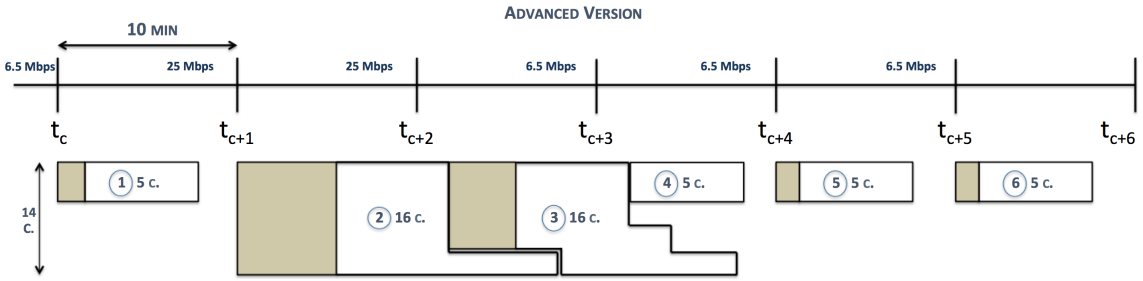

Figure 5.8: Test case 3 with the advanced version of the Load Balancer

job three, which guarantees that job three does not get delayed. Similar to the previous test case, the more advanced version is capable of verifying if the penultimate wave of job two ends within

Table 5.7: Job details for figure 5.8

Job Number	Average Throughput During Capture	Upload Start Time	Job Submit Time	Predicted Job Finish Time	Job Finish Time
Job 1	6.5 Mbps	11:59:56	12:01:29	12:07:44	12:08:02
Job 2	25 Mbps	12:09:44	12:15:12	12:27:42	12:27:45
Job 3	25 Mbps	12:19:46	12:25:35	12:38:05	12:37:57
Job 4	6.5 Mbps	12:29:48	12:31:41	12:37:56	12:38:17
Job 5	6.5 Mbps	12:39:51	12:41:21	12:47:36	12:47:57
Job 6	6.5 Mbps	12:49:53	12:51:22	12:57:37	12:58:18

the upload period of job three, and if the amount of free containers during the last wave of job two is enough to fit the last wave of job three. Since this is the case, job three can be scheduled locally without delays. In addition, job four is also scheduled locally due to the fact that the threshold is set after the upload period. In this case, there is no added delay to its execution[4].

Once again, test case three is another situation in which the more advanced version performs better than the simple version of the Load Balancer.

5.4.1.4 Test Case 4

Test case four, depicted in figures 5.9 and 5.10, is similar to test case three, with the difference being the network traffic throughput (12 Mbps) before and after the peaks processed at t_{c+1} and t_{c+2}.

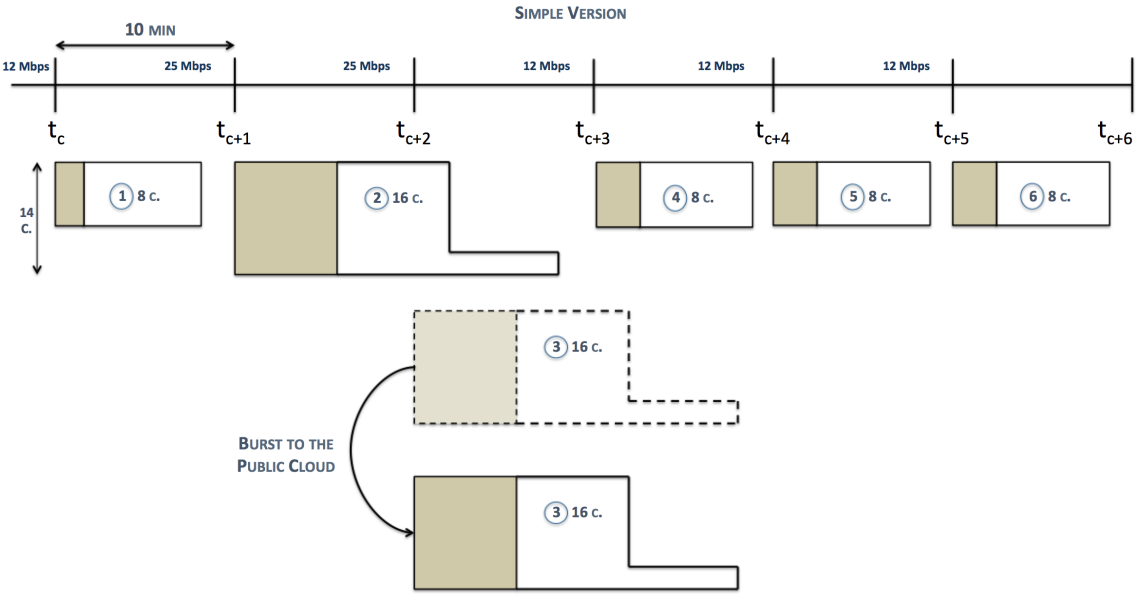

Figure 5.9: Test case 4 with the simple version of the Load Balancer

[4]The time difference between the predicted job finish time and real job finish time is due to the precision of the prediction algorithm

Table 5.8: Job details for figure 5.9

Job Number	Average Throughput During Capture	Upload Start Time	Job Submit Time	Predicted Job Finish Time	Job Finish Time
Job 1	12 Mbps	13:58:22	14:00:04	14:06:19	14:06:30
Job 2	25 Mbps	14:08:09	14:13:46	14:26:16	14:26:05
Job 3	25 Mbps	14:18:12	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 4	12 Mbps	14:28:14	14:30:50	14:37:05	14:37:20
Job 5	12 Mbps	14:38:17	14:41:02	14:47:17	14:47:30
Job 6	12 Mbps	14:48:20	14:51:08	14:57:23	14:57:35

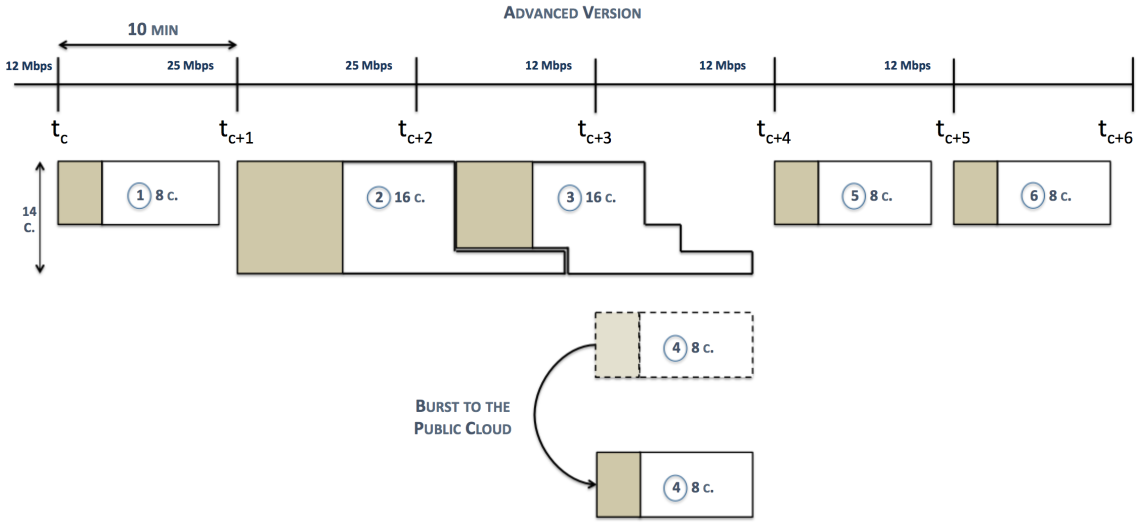

Figure 5.10: Test case 4 with the advanced version of the Load Balancer

Table 5.9: Job details for figure 5.10

Job Number	Average Throughput During Capture	Upload Start Time	Job Submit Time	Predicted Job Finish Time	Job Finish Time
Job 1	12 Mbps	12:15:18	12:27:49	12:34:04	12:34:26
Job 2	25 Mbps	12:35:06	12:41:40	12:54:09	12:54:06
Job 3	25 Mbps	12:45:10	12:52:04	13:04:34	13:04:27
Job 4	12 Mbps	12:55:14	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 5	12 Mbps	13:05:19	13:07:54	13:14:09	13:14:24
Job 6	12 Mbps	13:15:08	13:17:41	13:23:56	13:24:11

While the outcome of the simple version remains unaffected, the more advanced version is forced to burst job four to avoid it being delayed by the last wave of job three. Even though both

versions are forced to burst jobs to the public Cloud, note that the more advanced version bursts the job with the least execution time. In the situation where a public Cloud with a pay-as-you-go model is used, the more advanced version is capable of minimizing costs.

5.4.1.5 Test Case 5

The scenario depicted in section 5.3 is now presented using both versions of the Load Balancer. This test case illustrates the use case found in figures 4.15 and 4.16.

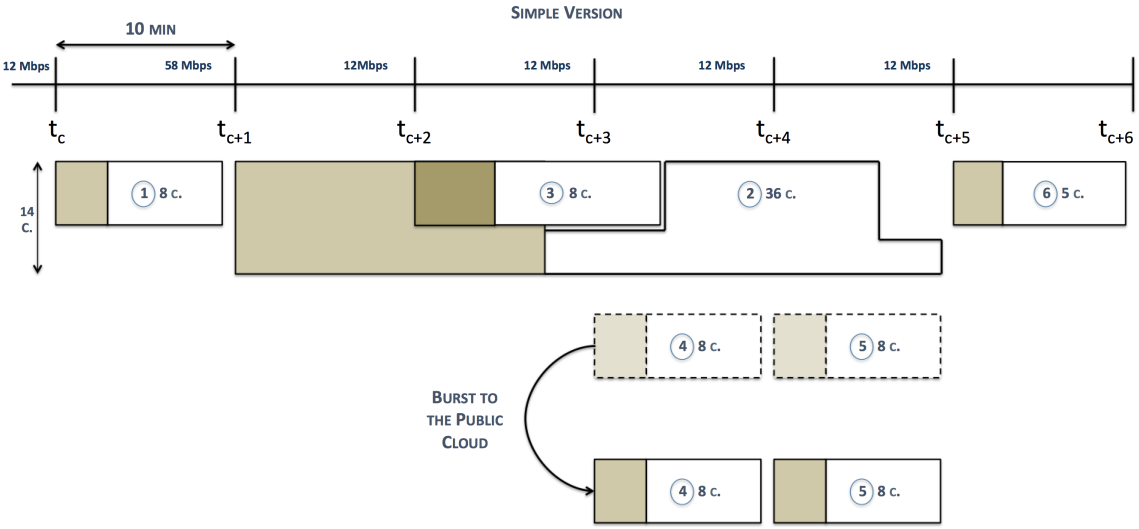

Figure 5.11: Test case 5 with the simple version of the Load Balancer

Table 5.10: Job details for figure 5.11

Job Number	Average Throughput During Capture	Upload Start Time	Job Submit Time	Predicted Job Finish Time	Job Finish Time
Job 1	12 Mbps	17:48:43	17:51:43	17:57:58	17:58:29
Job 2	58 Mbps	17:58:30	18:15:32	18:53:02	18:37:38
Job 3	12 Mbps	18:08:32	18:13:00	18:19:15	18:22:57
Job 4	12 Mbps	18:18: 34	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 5	12 Mbps	18:28:30	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 6	12 Mbps	18:38:38	18:41:26	18:47:41	18:48:19

The simple version, unable to detect uploads in progress, schedules job three to the local data center because it verifies at t_{c+2} that there are no jobs running in the cluster and thus all containers are free. Approximately two minutes and 30 seconds after job three is submitted, job two is also submitted and executes in parallel, competing for the resources. As a consequence, both jobs are delayed, but for different reasons. The presence of job three causes an additional wave at the end of

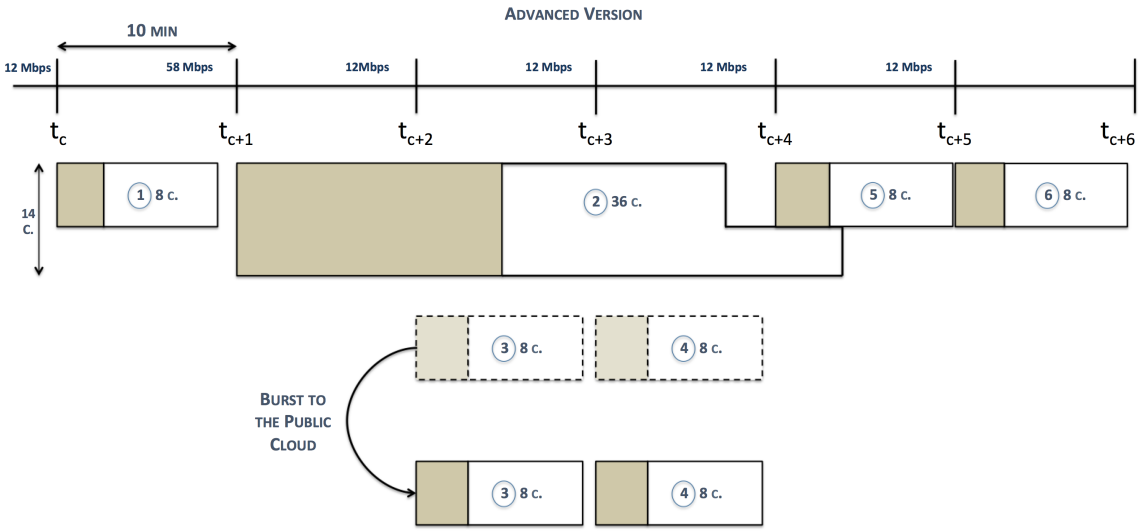

Figure 5.12: Test case 5 with the advanced version of the Load Balancer

Table 5.11: Job details for figure 5.12

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	12 Mbps	16:20:15	16:22:57	16:29:12	16:30:00
Job 2	58 Mbps	16:30:02	16:44:45	17:03:30	17:03:50
Job 3	12 Mbps	16:40:07	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 4	12 Mbps	16:50:10	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 5	12 Mbps	17:00:08	17:03:22	17:09:37	17:10:38
Job 6	12 Mbps	17:10:10	17:14:07	17:20:22	17:21:21

job two, reflecting the maps that should have started right at the beginning but were pushed back because the resources were occupied (by job three). This additional wave adds approximately three minutes to the execution of job two. The delay seen in job three occurs because job two hogs up all of the free resources at the moment job three releases the containers it requires for its maps. Hence, job three's otherwise fast reduce phase is sent into a pending state, waiting for free containers to finish the job. Lastly, job four and five are burst to the public Cloud since all of the containers are in use at t_{c+3} and t_{c+4}.

The more advanced version, on the other hand, bursts incoming jobs if it detects that there is an upload in progress. Thus, at t_{c+2} job three is burst to the public cloud to avoid possible delays. At t_{c+3} job four is also burst, because job two's penultimate wave ends way ahead of job four's predicted upload period (even with the added tolerance). Job five, however, is launched locally because the last wave of job two ends within job five's predicted upload period (with the added tolerance). Even though job five executes normally, table 5.9 shows almost a one minute difference

between the predicted finish time and the real finish time. This difference occurred mainly for two reasons: the first reason is the fact that job five took approximately 17 seconds longer to execute than the average, which is a normal behavior since the execution time and conditions are not static and constant values; the second reason is the circa 29 seconds that it took for job five to start executing after being submitted. We believe these 29 seconds were caused by overhead in the ResourceManager, since job two was releasing its containers and finishing its execution, while job five was being submitted and requesting its AM to start executing. There is also a difference of a few seconds between the submission of a job on our solution and the submission on Hadoop. We see a similar situation in job six's case, where the predicted finish time and the real finish time also have nearly a one minute difference. However, in this case, job six's execution started immediately after being submitted. The reasons for the difference are the fact that job six took more or less 50 seconds more to execute than normal, and the small difference in submission time between our solution and Hadoop. Again, we want to emphasize the fact that the execution time of a job is not a constant value. This explains the nearly one minute difference seen in job five and six's cases.

Note that, despite the fact that both versions burst two jobs to the public Cloud, the more advanced version is capable of minimizing overall job delays by bursting the job that the simple version launches locally while an upload is in progress. Once more, we conclude that the more advanced version provides better results than the simple version. Overall, not only is it able to burst less jobs, but also minimize delays and bursting times.

5.4.2 Real Traffic Analysis with Signature Matching

The following performance tests use the signature matching job on real traffic captured on our network labs' router. These tests were conducted with an upgraded network, where all links are GigabitEthernet. In addition, the number of worker nodes was reduced to four, allowing for a total of eight containers. Furthermore, all network traffic captures generate jobs that exceed the cluster's steady state maximum, which is seven containers per capture window, equating to approximately 896 MB or about 12 Mbps throughput during the 10 minute capture time window.

The signature matching job's behavior is very similar to the behavior observed with the simulation job. The average duration of a wave with signature matching is approximately five minutes and 35 seconds, which is close to the six minutes and 15 seconds seen in the simulation job. In the following tests, we have kept the setting for the average duration of a single map wave in our solution with the value for the simulation job. This is a user configurable setting, which upon adjustment may or may not provide better results.

Notice that the difference in the predicted finish time and the real finish time in the following test cases is due to the fact that we did not adapt the average single map duration configuration for the signature matching job. Regardless, the prediction is still able to provide reasonable predictions considering the full duration of the jobs, with the closest estimate being circa 44 seconds off and the average about one minute and 14 seconds off.

5.4.2.1 Test Case 1

In this test case, a peak of 23 Mbps is followed by four larger peaks of about 32.7 Mpbs. After the peaks, network traffic throughput lowers back to the initial peak of 23 Mbps. Figures 5.13 and 5.14 depict the outcome with the simple and the more advanced version respectively. The tables containing the job details for 5.13 and 5.14 are 5.12 and 5.13 respectively.

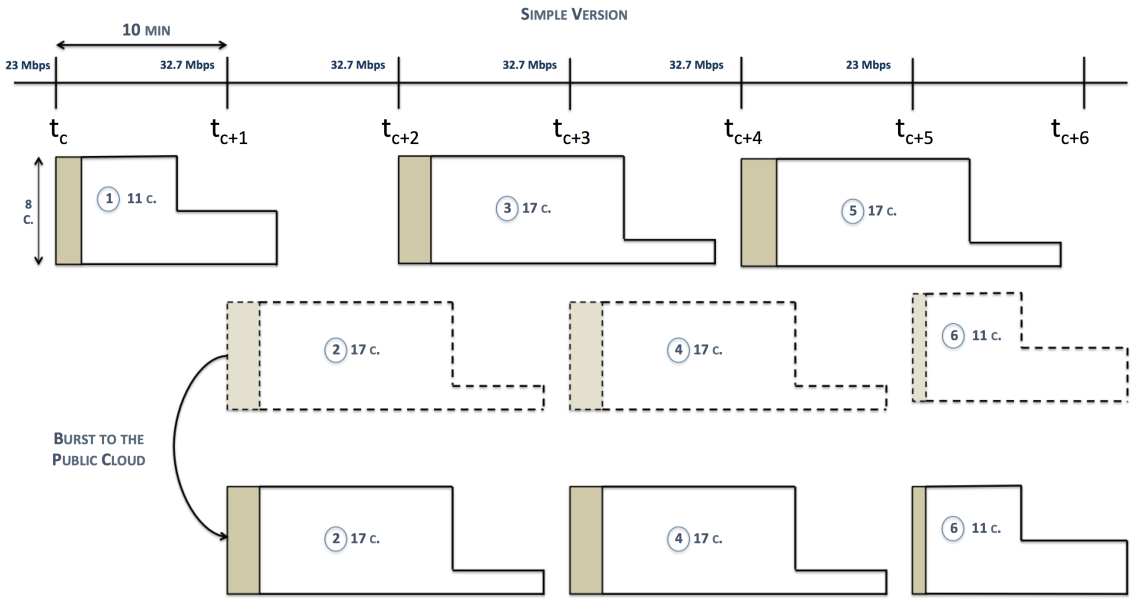

Figure 5.13: Test case 1 with the simple version of the Load Balancer

Table 5.12: Job details for figure 5.13

Job Number	Average Throughput During Capture	Upload Start Time	Job Submit Time	Predicted Job Finish Time	Job Finish Time
Job 1	23 Mbps	15:32:19	15:33:56	15:46:26	15:45:29
Job 2	32.8 Mbps	15:42:06	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 3	32.8 Mbps	15:52:08	15:54:05	16:12:50	16:11:09
Job 4	32.8 Mbps	16:02:11	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 5	32.8 Mbps	16:12:13	16:14:19	16:33:04	16:30:57
Job 6	23 Mbps	16:22:17	N/A (BURST)	N/A (BURST)	N/A (BURST)

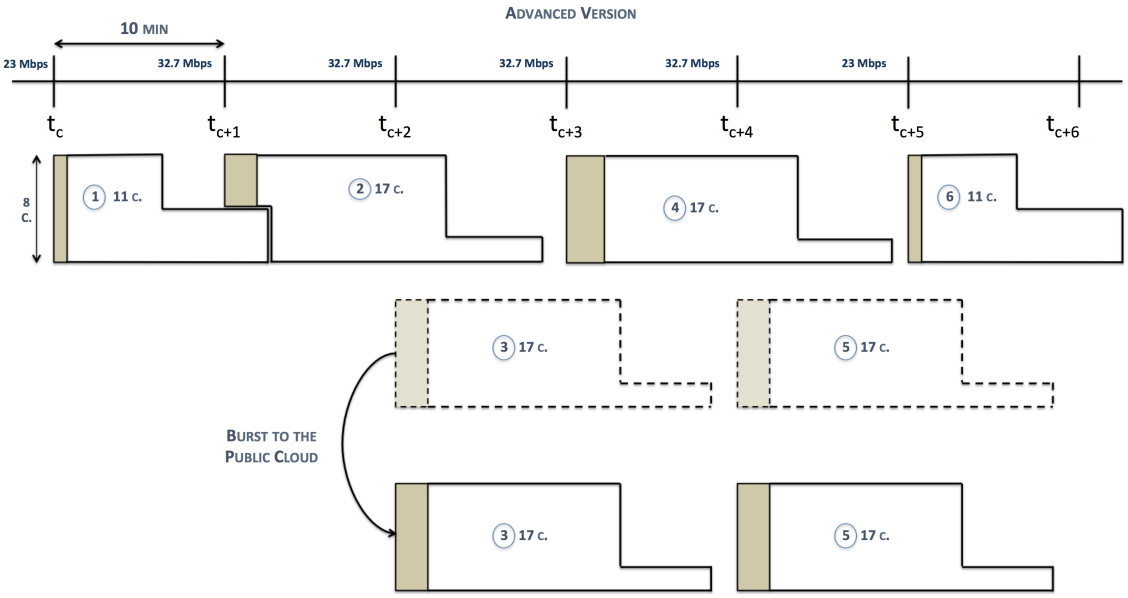

Figure 5.14: Test case 1 with the advanced version of the Load Balancer

As expected, the more advanced version is capable of bursting less jobs than the simple version. The simple version, unable to predict job completion and upload times, bursts job two because job one is still executing at t_{c+1}. Additionally, it bursts job four and six for the same reason. The more advanced version, on the other hand, uses the predictors and concludes that it can launch job two in the local cluster without it being delayed. Job three and five are burst because the predictors produce estimates that lead the Load Balancer to conclude that both jobs would be delayed if launched locally. We see once more that the more advanced version is able to burst less jobs than the simple version.

Table 5.13: Job details for figure 5.14

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	23 Mbps	14:07:04	14:07:49	14:20:19	14:19:33
Job 2	32.8 Mbps	14:16:51	14:18:45	14:37:30	14:35:40
Job 3	32.8 Mbps	14:26:53	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 4	32.8 Mbps	14:36:46	14:39:07	14:57:52	14:56:26
Job 5	32.8 Mbps	14:46:58	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 6	23 Mbps	14:57:00	14:58:10	15:10:39	15:09:55

5.4.2.2 Test Case 2

Test case two features a stream of network traffic with peaks that go up and down from 23 to 32.7 Mbps, which stabilize at 23 Mbps in the last two capture time windows. The result of this test case is illustrated in figures 5.15 and 5.16, alongside their respective detail tables 5.14 and 5.15.

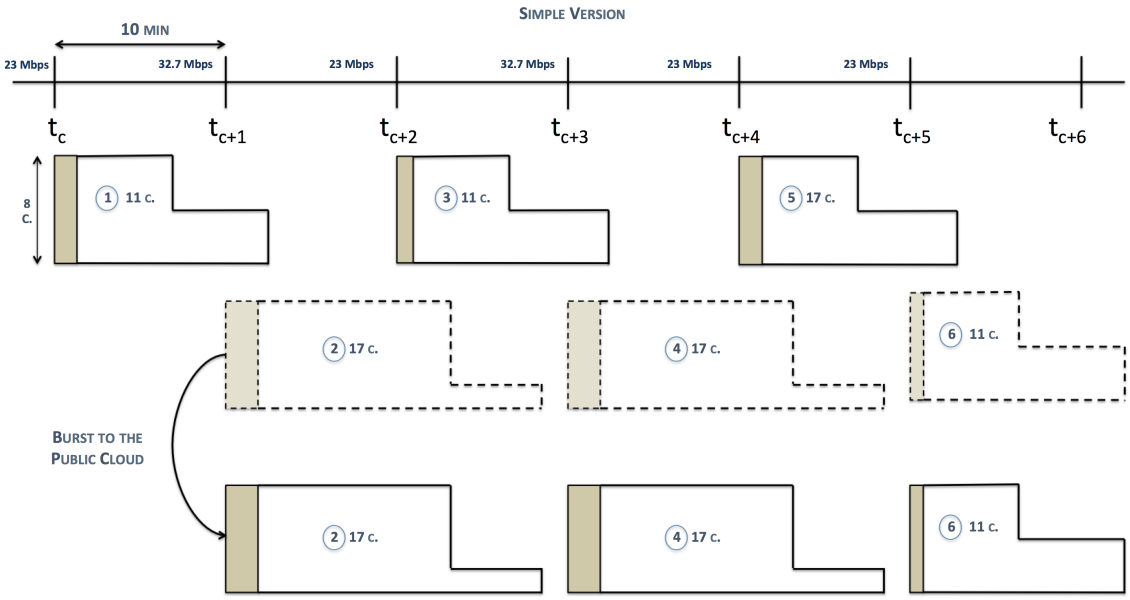

Figure 5.15: Test case 2 with the simple version of the Load Balancer

Table 5.14: Job details for figure 5.15

Job Number	Average Throughput During Capture	Upload Start Time	Job Submit Time	Predicted Job Finish Time	Job Finish Time
Job 1	23 Mbps	17:01:25	17:02:51	17:15:21	17:14:08
Job 2	32.8 Mbps	17:11:14	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 3	23 Mbps	17:21:16	17:22:24	17:34:44	17:33:38
Job 4	32.8 Mbps	17:31:19	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 5	23 Mbps	17:41:21	17:42:41	17:55:11	17:54:06
Job 6	23 Mbps	17:51:24	N/A (BURST)	N/A (BURST)	N/A (BURST)

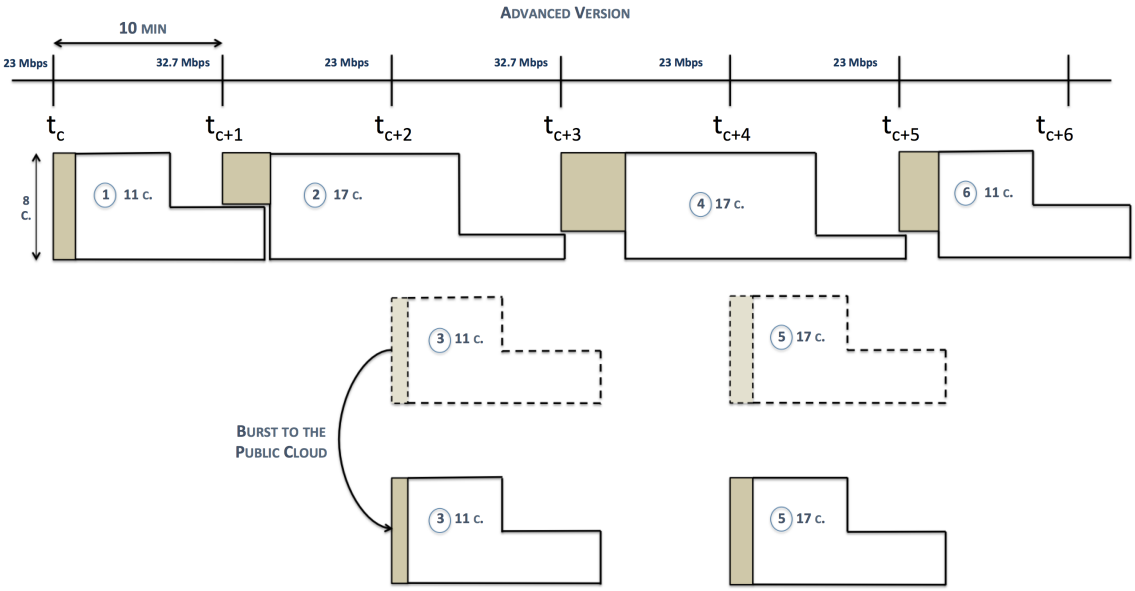

Figure 5.16: Test case 2 with the advanced version of the Load Balancer

Table 5.15: Job details for figure 5.16

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	23 Mbps	09:02:40	09:03:44	09:16:14	09:15:22
Job 2	32.8 Mbps	09:12:27	09:15:17	09:34:02	09:32:39
Job 3	23 Mbps	09:22:29	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 4	32.8 Mbps	09:32:32	09:36:20	09:55:05	09:54:06
Job 5	23 Mbps	09:42:34	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 6	23 Mbps	09:52:36	09:54:52	10:07:22	10:06:17

Similar to test case one in 5.4.2.1, the simple Load Balancer bursts job two, even though job two would not have been delayed if it had been launched locally. The more advanced version, on the other hand, manages to launch job two locally because the Load Balancer estimates that the number of required containers in the last wave of job two fits in the last wave of job one. Again we see that the more advanced version is capable of reducing the number of necessary bursts.

5.4.2.3 Test Case 3

Finally, we present test case three, in which the network traffic throughput continuously peaks at an average of 23 Mbps during the whole test case. 23 Mbps exceeds the steady state maximum, generating jobs that span two map waves. Figures 5.17 and 5.18 alongside tables 5.16 and 5.17 describe test case three.

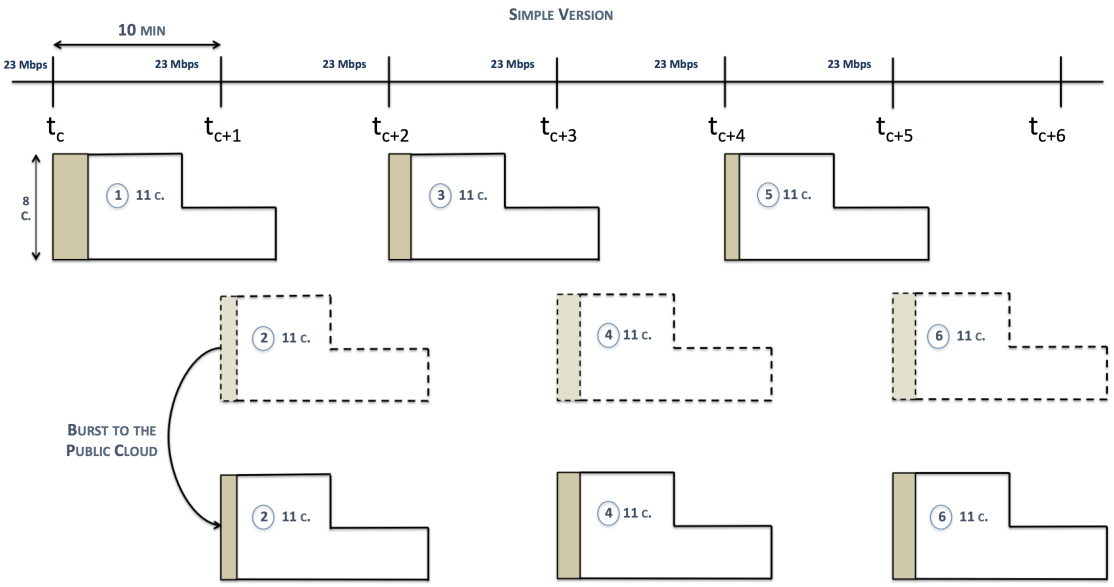

Figure 5.17: Test case 3 with the simple version of the Load Balancer

Table 5.16: Job details for figure 5.17

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	23 Mbps	11:29:27	11:31:43	11:43:11	11:45:01
Job 2	23 Mbps	11:39:14	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 3	23 Mbps	11:49:17	11:50:38	12:03:08	12:01:58
Job 4	23 Mbps	11:59:20	N/A (BURST)	N/A (BURST)	N/A (BURST)
Job 5	23 Mbps	12:09:22	12:10:14	12:22:4	12:21:33
Job 6	23 Mbps	12:19:25	N/A (BURST)	N/A (BURST)	N/A (BURST)

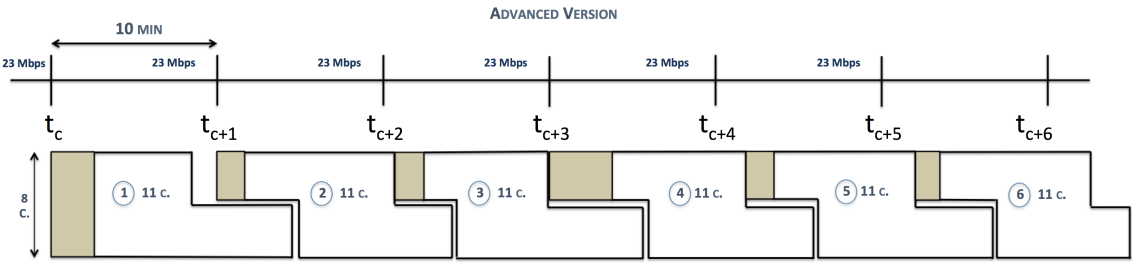

Figure 5.18: Test case 3 with the advanced version of the Load Balancer

Table 5.17: Job details for figure 5.18

Job Number	**Average Throughput During Capture**	**Upload Start Time**	**Job Submit Time**	**Predicted Job Finish Time**	**Job Finish Time**
Job 1	23 Mbps	10:15:00	10:17:43	10:30:13	10:29:59
Job 2	23 Mbps	10:24:47	10:26:28	10:38:58	10:39:10
Job 3	23 Mbps	10:34:50	10:37:18	10:49:48	10:50:36
Job 4	23 Mbps	10:44:52	10:48:35	11:01:05	11:00:50
Job 5	23 Mbps	10:54:54	10:57:28	11:10:57	11:12:05
Job 6	23 Mbps	11:04:56	11:08:34	11:21:04	11:20:18

The simple container usage based Load Balancer decides to burst jobs two, four and six because the jobs preceding these are still executing, occupying a portion of the containers, at the moment the PCAP files for jobs two, four and six are ready to be scheduled. The more advanced version stands out by being able to schedule all jobs, including the ones the simple version burst, locally. The advanced version does not burst any job. While we noticed that some jobs took longer to execute than expected (circa two minutes in the worst case), we have not yet found the explanation for the prolonged execution. All jobs should have executed normally according to our map wave model, as seen in other test cases. This is a subject that should be further explored. According to our map wave model, the more advanced version performed as expected and produced the results we desired.

5.5 Real Traffic Analysis with Signature Matching in a Heterogeneous Cluster

Thus far, all tests were performed on a homogeneous cluster, where all nodes are equally powerful. Since Hadoop is thought from the ground up to be used with commodity hardware, it is not uncommon for companies to build Hadoop clusters out of different unused hardware and continue adding nodes as technology evolves and other computers are replaced. While our solution is not built with heterogeneity in mind, we believe it should also be tested on a heterogeneous cluster to study the outcome.

The following performance test is done on a four node heterogeneous Hadoop cluster. Each node has four GB of RAM, two virtual CPUs and a maximum of two containers. Similar to the homogeneous cluster in the previous section, four nodes gives us a total of eight containers. The difference lies in the different physical CPUs in which the containers execute. While the number of vCPUs per node is the same, their specifications and speed are not. With this in mind, we expect the maps that run on faster nodes to execute faster than the ones that run on the slower nodes. The specifications for the physical CPUs, retrieved through the linux `lscpu` command, can be seen in table 5.18.

Table 5.18: Heterogeneous Cluster CPU Specifications

Node Nº	**Nº of CPUs**	**CPU Model Name**	**Cache (KB)**	**Responsible for Containers**	**Rank**
1	8	Intel(R) Core(TM) i7-3770 CPU @ 3.40 GHz	**L1d:** 32 **L1i:** 32 **L2:** 256 **L3:** 8192	1 and 2	1st (Fastest)
2	8	Intel(R) Core(TM) i7 CPU 950 @ 3.07 GHz	**L1d:** 32 **L1i:** 32 **L2:** 256 **L3:** 8192	3 and 4	2nd (Second Fastest)
3	4	Intel(R) Xeon(R) CPU E5504 @ 2.00 GHz	**L1d:** 32 **L1i:** 32 **L2:** 256 **L3:** 4096	5 and 6	3rd (Slowest)
4	8	Intel(R) Core(TM) i7 CPU 950 @ 3.07 GHz	**L1d:** 32 **L1i:** 32 **L2:** 256 **L3:** 8192	7 and 8	2nd (Second Fastest, tied with node two)

Despite the similarities between all four CPUs, the difference in the amount of cache and the CPU model is enough to give us a notion of what to expect in terms of performance. Thus, given table 5.18, we expect the maps running on node one to execute the fastest, followed by a tie between node two and four, and lastly node three.

We have taken all three test cases from the previous section and combined them into a single continuous test case, where test case one (5.4.2.1) is directly followed by test case two (5.4.2.2) and three (5.4.2.3). Our test covers three different scenarios: no Load Balancer, using the simple version of the Load Balancer and finally using the more advanced version. We immediately conclude from our three different scenarios that our expectations regarding the map execution speed on differently powered CPUs are confirmed. In fact, we see that in 44 jobs out of the 46 that were launched locally the maps that execute in containers five and six (node three) are the slowest. Also, in almost all cases, the maps running in containers one and two take the least to execute. Lastly, maps running in containers three, four, seven and eight are close in execution speed, standing be-

hind containers one and two, but ahead of five and six. The speed difference in the execution of the maps can be an issue if the difference is such that it might void the logic of the map waves. Consider the case where the Load Balancer launches a double map wave job with seven maps in the second wave. Imagine that some of the containers are much faster than the others. As a consequence, some of the maps that were predicted to execute in the second wave, actually execute in the first wave, leaving the second wave with less maps to execute. Since the Load Balancer is not built with this in mind, it is possible that some jobs get incorrectly burst because of a mistaken prediction.

Let us examine how our solution performs in our heterogeneous cluster. Before starting our test, we launched the job with all the different PCAP files, one at a time, to build estimations regarding job completion times when the jobs are executing by themselves in the cluster. So, delays are measured according to the time it would take that job to execute if it had executed by itself. In the first scenario (no Load Balancer), all jobs are scheduled blindly to the local heterogeneous cluster. We detect a total of 10 delayed jobs in this situation. Some of those delays are small enough to be discarded (less than one minute), because as explained previously, job execution time is not a constant value. However, we highlight the following four delays:

Table 5.19: Longest delays in the no Load Balancer scenario

Job Number	**Average Throughput During Capture**	**Job Start Time**	**Job Finish Time**	**Average Execution Time**	**Approximate Delay**
Job 51	32.8 Mbps	22:12:31	22:28:30	13:48	02:11
Job 54	23 Mbps	22:45:07	22:57:11	07:52	04:12
Job 55	23 Mbps	22:52:45	23:04:34	07:52	03:57
Job 57	23 Mbps	23:11:42	23:26:03	07:52	06:29

If the peaks of network traffic grow even bigger, the number of delayed jobs and/or the size of the delays will be worse. Thus, we apply our simple Load Balancer to study how it deals with the same test. Immediately, the number of delays is reduced from 10 to four, out of which three can be discarded for being too small. The only delay worth mentioning (circa one minute and 44 seconds) is caused by a rack-local map. The reduction in the amount of delays is achieved by the bursting of five jobs. Our more advanced version, on the other hand, is capable of reducing the number of bursts to three, while keeping the delays similar to the simple version, as expected. Out of the three delays seen in the outcome of the more advanced version, two are small enough to be neglected. The remaining delay is approximately five minutes long, but is caused by a rack-local map task, which is not under the control of our Load Balancer.

We conclude that both versions of the Load Balancer are able to reduce the amount of delays in our heterogeneous cluster, even when the Load Balancer is not adapted to the consequences of the hardware heterogeneity. Once again we see that the more advanced version performs better

than the simple version by being capable of reducing the amount of bursts and maximizing the local resource usage.

Chapter 6

Conclusions

In this dissertation, we introduced the importance of embracing Big Data and presented Apache Hadoop, a framework capable of handling such large data sets. We defined High Availability and presented the state of the art for high availability techniques that can be implemented in Hadoop clusters. We then followed by describing the problem with investing in highly available and more powerful hardware to improve the availability of private data centers and deal with unexpected peaks of computing demand. Not only is investing in local resources expensive, but it also usually leads to underutilization. The hybrid cloud model and Cloud Bursting are introduced as solutions capable of providing the required scalability for peaks of demand, while minimizing investment costs and maximizing resource utilization.

We presented a Cloud Bursting capable solution for a Hadoop cluster running MapReduce jobs to process and analyze network traffic. The analysis of the captured network traffic was performed by our map-intensive signature matching job, which is based on a set of rules from the Snort community and is able to detect intrusions or similar attacks. Our Cloud Bursting capable solution is an inter-cluster Load Balancer that bursts jobs when the local cluster is overloaded and cannot process further jobs without delaying them. To achieve this, the Load Balancer closely monitors the local cluster's resource utilization, captured network traffic batch size, and the possibility of delays to decide whether or not to burst a job. To develop our Load Balancer, we designed a model for the behavior of a map-intensive network traffic analysis job, based on the concept of map waves. This model, though simple, is of high importance to the Load Balancer to grant it the ability to estimate wave and job completion times to aid the decision process. We thoroughly described our solution and detailed the development of the Load Balancer.

Finally, we submitted our solution to a series of test cases and present its results. A simple version of the Load Balancer, where only container utilization and execution of other jobs is considered, was tested against a more advanced version with all of the features described in 4.4. We used synthetic traffic generated by Iperf to prove the concept and the development steps of our Cloud Bursting capable Load Balancer. It becomes clear that the need for the Load Balancer is real, as the occurrence of peaks in network traffic can lead to severe delays in the execution of the jobs that follow. Our Load Balancer underwent testing with the signature matching job together

with real network traffic captured at the router in our network labs. Lastly, we tested our Load Balancer in a heterogeneous cluster with real traffic. While using the simple Load Balancer is always better than not using any Load Balancer at all, we conclude that the more advanced version provides the best results by minimizing the number of bursts and maximizing local resource usage.

Thus, we believe that the Cloud Bursting technique is in fact a viable option to improve the availability of Hadoop clusters aimed at analyzing network traffic with our map-intensive signature matching MapReduce job. We are convinced that this is a subject that should be further studied and that our solution should be further developed to better suit clusters with different specifications. Particularly, our solution could be upgraded to a full application, capable of providing system administrators with a graphical user interface and powerful tools to automatically adapt our solution to the cluster's specifications upon installation. It should be able to determine the cluster's steady state limit, inform whether the cluster needs an upgrade or not considering the amount of bursts, provide visual and detailed alerts when attacks occur, display cluster utilization statistics, among other features.

In conclusion, this dissertation reflected our study of the viability of the Cloud Bursting technique as a means to improve the availability of a Hadoop cluster aimed at performing network traffic analysis through a map-intensive MapReduce job. The problem was thoroughly analyzed and a solution was presented and detailed. After a series of tests, we concluded that our solution does in fact improve the availability and is a subject with potential that should be further explored.

Appendix A

Map-Intensive Network Analysis Job

A.1 Network Analysis MapReduce Job

Listing A.1: Source Code for both map-intensive MapReduce Jobs

```
public void map(LongWritable key, BytesWritable value,
    OutputCollector<Text, Text> output, Reporter reporter) throws
     IOException
{
        // per packet delay
        long sleepTime = MAPms*1000000L; // convert to
            nanoseconds
        long startTime = System.nanoTime();

        while ((System.nanoTime() - startTime) < sleepTime) {}

        // or matching of virus and application signatures
        byte[] value_bytes = value.getBytes();
        System.arraycopy(value_bytes, PcapRec.POS_IP_BYTES, bc,
            0, 2);
        Long ibc = Bytes.toLong(bc);
        String[] s0_array = new String[] { "StoogR", "/scripts/
            CGImail.exe", "/scripts/proxy/w3proxy.dll", // (...)
            };

        int found_any = 0;
        for (int j = 0; j < s0_array.length; j++)
        {
```

```
                int pos = matchString(s0_array[j], value_bytes,
                    ibc);

                if (pos != -1)
                {
                        out = out + "␣" + j + "␣" + pos + "␣";
                        found_any = 1;
                }
        }

        if (found_any != 0) output.collect(new Text(strTuple),
            new Text(out));
}


public int matchString(String s0, byte[] value_bytes, long
    ip_len_bytes)
{

        byte[] str0 = s0.getBytes();
        byte[] str = new byte[str0.length];
        int len = str0.length;

        for (int i = PcapRec.POS_IP_VER + 40; i < PcapRec.
            POS_IP_VER + ip_len_bytes; i++)
        {
                if (i + len >= value_bytes.length )
                        break;

                System.arraycopy(value_bytes, i, str, 0, len);

                if (Arrays.equals(str0, str))
                {
                        return i - PcapRec.POS_IP_VER - 40;
                }
        }

        return -1;
}
```

References

[1] Brad Brown, Michael Chui, and James Manyika. Are you ready for the era of ‘ big data ’? *Library*, October(October):1–12, 2011. URL: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Are+you+ready+for+the+era+of+?+big+data+??#1, arXiv:67227835.

[2] Angela Hung Byers James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh. Big data: The next frontier for innovation, competition, and productivity, 2011. URL: http://scholar.google.com/scholar.bib?q=info:kkCtazs1Q6wJ:scholar.google.com/&output=citation&hl=en&as_sdt=0,47&ct=citation&cd=0.

[3] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 1013, 2010. URL: http://portal.acm.org/citation.cfm?doid=1807167.1807278, doi:10.1145/1807167.1807278.

[4] Doug Laney. META Delta. *Application Delivery Strategies*, 949(February 2001):4, 2001.

[5] Sas.com. What is big data?, 2014. [Accessed: Dec. 16, 2014]. URL: http://www.sas.com/en_us/insights/big-data/what-is-big-data.html.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review*, 37(5):29, December 2003. URL: http://portal.acm.org/citation.cfm?doid=1165389.945450, doi:10.1145/1165389.945450.

[7] Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, and Jorge-arnulfo Quian. Efficient Big Data Processing in Hadoop MapReduce. *Proc. VLDB Endow.*, 5(12):2014–2015, August 2014. URL: http://dx.doi.org/10.14778/2367502.2367562, doi:10.14778/2367502.2367562.

[8] Jiong Xie. *Improving Performance of Hadoop Clusters*. PhD thesis, Auburn University, 2011.

[9] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, and Others. Apache hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[10] Dhruba Borthakur, Samuel Rash, Rodrigo Schmidt, Amitanand Aiyer, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik

Ranganathan, Dmytro Molkov, and Aravind Menon. Apache hadoop goes realtime at Facebook. *Proceedings of the 2011 international conference on Management of data - SIGMOD '11*, page 1071, 2011. URL: http://portal.acm.org/citation.cfm?doid=1989323.1989438, doi:10.1145/1989323.1989438.

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5496972, doi:10.1109/MSST.2010.5496972.

[12] Dirk deRoos. Input splits in hadoop's mapreduce - for dummies, 2015. [Accessed: Apr. 30, 2015]. URL: http://www.dummies.com/how-to/content/input-splits-in-hadoops-mapreduce.html.

[13] B Y Jeffrey Dean, Sanjay Ghemawat, Jeffrey Dean, and Sanjay Ghemawat. MapReduce: a flexible data processing tool. *Communications of the ACM*, 53:72–77, 2010. URL: http://dl.acm.org/citation.cfm?id=1629198.

[14] Cloudera.com. Migrating from mapreduce 1 (mrv1) to mapreduce 2 (mrv2, yarn), 2014. [Accessed: Nov. 10, 2014]. URL: http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/cdh_ig_mapreduce_to_yarn_migrate.html.

[15] Matei Zaharia, D Borthakur, J S Sarma, K Elmeleegy, S Shenker, and I Stoica. Job Scheduling for Multi-User MapReduce Clusters. *EECS Department University of California Berkeley Tech Rep UCBEECS200955 Apr*, (UCB/EECS-2009-55):2009–55, 2009. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.pdf.

[16] Evan Marcus and Hal Stern. *Blueprints for High Availability*, volume 40. March 2001. doi:10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C.

[17] Hadoop.apache.org. Apache hadoop 2.3.0 - hadoop distributed file system-2.3.0 - high availability, 2015. [Accessed: Oct. 6, 2014]. URL: http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/HDFSHighAvailabilityWithNFS.html.

[18] Cloudera.com. Yarn (mrv2) resourcemanager high availability, 2015. [Accessed: Oct. 6, 2014]. URL: http://www.cloudera.com/content/cloudera/en/documentation/core/latest/topics/cdh_hag_rm_ha_config.html.

[19] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. *in Proceedings of the 5th European conference on Computer Systems*, pages 265–278, 2010. URL: http://portal.acm.org/citation.cfm?id=1755913.1755940.

[20] Yuanquan Fan, Weiguo Wu, Depei Qian, Yunlong Xu, and Wei Wei. Load Balancing in Heterogeneous MapReduce Environments. *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1480–1489, November

2013. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6832091, doi:10.1109/HPCC.and.EUC.2013.209.

[21] Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. Cost-Aware Cloud Bursting for Enterprise Applications. *ACM Transactions on Internet Technology*, 13(3):1–24, 2014. URL: http://dl.acm.org/citation.cfm?doid=2630790.2602571.

[22] Srijith K. Nair, Sakshi Porwal, Theo Dimitrakos, Ana Juan Ferrer, Johan Tordsson, Tabassum Sharif, Craig Sheridan, Muttukrishnan Rajarajan, Afnan Ullah Khan, Ieee European, Web Services, Ieee European, Web Services, Permanent City, and Moral Rights. Towards secure cloud bursting, brokerage and aggregation. In *Proceedings - 8th IEEE European Conference on Web Services, ECOWS 2010*, pages 189–196, 2010.

[23] Tekin Bicer, David Chiu, and Gagan Agrawal. A Framework for Data-Intensive Computing with Cloud Bursting. *2011 IEEE International Conference on Cluster Computing*, pages 169–177, September 2011. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6061052, doi:10.1109/CLUSTER.2011.21.

[24] Sriram Kailasam, Prateek Dhawalia, S. J. Balaji, Geeta Iyer, and Janakiram Dharanipragada. Extending MapReduce across Clouds with BStream. *IEEE Transactions on Cloud Computing*, 2(3):362–376, July 2014. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6786985, doi:10.1109/TCC.2014.2316810.

[25] B. Heintz, a. Chandra, and J. Weissman. Cross-Phase Optimization in MapReduce. *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 338–347, March 2013. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6529301, doi:10.1109/IC2E.2013.26.

[26] Tcpdump.org. Manpage of pcap, 2015. [Accessed: Jan. 8, 2015]. URL: http://www.tcpdump.org/manpages/pcap.3pcap.html.

[27] Lm Garcia. Programming with Libpcap - Sniffing the Network From Our Own Application. *Hakin9-Computer Security Magazine*, 2008. URL: http://www.packet-craft.net/PCAP/SrcCode/aldabaknocking.com/libpcapHakin9LuisMartinGarcia.pdf.

[28] Asrodia Pallavi and Patel Hemlata. Network Traffic Analysis Using Packet Sniffer. *International Journal of Engineering Research and Applications*, 2(3):854–856, 2012.

[29] Yeonhee Lee and Youngseok Lee. Toward scalable internet traffic measurement and analysis with Hadoop. *ACM SIGCOMM Computer Communication Review*, 43(1):5–13, 2012. URL: http://dl.acm.org/citation.cfm?doid=2427036.2427038, doi:10.1145/2427036.2427038.

[30] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. *Proceedings of the 10th ACM conference on Computer and communication security - CCS '03*, page 262, 2003. URL: http://portal.acm.org/citation.cfm?doid=948109.948145, doi:10.1145/948143.948145.

[31] Tcpdump.org. Tcpdump/libpcap public repository, 2015. [Accessed: May. 10, 2015]. URL: http://www.tcpdump.org.