Hadoop, and Its Mechanisms for Reliable Storage

Ingvild Hovdelien Midthaug

Master of Science in Communication Technology

Submission date:	June 2018
Supervisor:	Danilo Gligoroski, IIK
Co-supervisor:	Katina Kralevska, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Hadoop, and Its Mechanisms for Reliable Storage

Ingvild Hovdelien Midthaug

Submission date: June 2018
Responsible professor: Danilo Gligoroski, IIK, NTNU
Supervisor: Katina Kralevska, IIK, NTNU

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Hadoop, and Its Mechanisms for Reliable Storage
Student:	Ingvild Hovdelien Midthaug

Problem description:

Hadoop is the most popular open-source software framework used for distributed storage and processing of big data sets [Whi12]. It is a collection of many related sub-projects. These projects are hosted by the Apache Software Foundation, which provides support for a community of open source software projects. One of the most important sub-projects in Hadoop is the Hadoop Distributed File System (HDFS). It relies on the concepts of splits and blocks. A split is a logical representation of the data while a block describes the physical alignment of data. Splits and blocks in Hadoop are user-defined: a logical split can be composed of multiple blocks and one block can have multiple splits. All these choices determine in a more complex way the access time for input/output (I/O) operations. A typical mechanism for achieving reliability of data in HDFS is the triple-replication of data. However, the overhead cost in triple-replication is 200%. Erasure codes offer similar reliability with much less overhead. From release 3.0.0 Hadoop offers several erasure codes such as (9, 6) and (14, 10) Reed-Solomon (RS) codes.

This master thesis will present principles of Hadoop and how an experimental Hadoop environment is set up. Further, mechanisms for achieving reliable data storage (triple-replication vs. erasure codes) will be tested and compared. The comparison will include measurements on the time of recovery, measurements on the network traffic during recovery and the influence of different block and file sizes on those measurements.

Responsible professor:	Danilo Gligoroski, IIK, NTNU
Supervisor:	Katina Kralevska, IIK, NTNU

Abstract

Nowadays the global amount of digital data increases rapidly. Internet-connected devices generate massive amounts of data through various interactions such as digital communication and file sharing. In a world surrounded by such interactions every day, this results in Big Data sets. The datasets can be analyzed and further be used for various purposes such as personalized marketing and health research. In order to analyze and utilize the data, it has to be transferred and stored reliably. Failures in storage systems happen frequently, so mechanisms for reliable data storage are needed. The Hadoop software provides a distributed file system that achieves reliable data storage through different coding techniques.

This thesis presents different mechanisms for reliable data storage in Hadoop and gives a practical implementation of an experimental Hadoop environment. The mechanisms include erasure coding (Reed-Solomon codes) and triple-replication. Further, the performance of the mechanisms is tested and compared. The performance parameters considered are the time of file recovery and the amount of network traffic during file recovery. Factors affecting the performance, such as file size and block size, are also considered. The test setup includes wired Ethernet connection, a configured multi-node Hadoop cluster, a managed network switch and a network analysis tool.

The obtained results show the impact of different factors on the Hadoop cluster performance during node failure. In general, the results confirm theory. Both the time of recovery and the network traffic during recovery increase with the file size. For erasure coding, the time of recovery increases with the code length, and block size of 128 MB gives the best overall performance. Moreover, optimized erasure coding variants for improving the cluster performance are presented in related work and then suggested as future work.

Sammendrag

I dag øker den globale mengden av digital data fortere enn noen gang. Ulike enheter som er koblet til internett (smarttelefoner, PC-er osv.) genererer enorme mengder data gjennom forskjellige interaksjoner slik som digital kommunikasjon og fildeling. Når dette skjer overalt i verden hver eneste dag resulterer det i Big Data datasett. Disse datasettene kan analyseres og senere bli brukt til ulike formål, eksempelvis personalisert markedsføring og forskning innen helse. For å kunne analysere og nyttiggjøre dataene må de overføres og lagres pålitelig. Det er stort behov for mekanismer som sikrer pålitelig datalagring fordi hyppige maskinvare- eller programvarefeil er vanlig i store lagringssystemer. Hadoop er et rammeverk som tilbyr pålitelig datalagring gjennom ulike kodeteknikker.

Denne masteroppgaven presenterer ulike mekanismer for pålitelig datalagring i Hadoop og gir en praktisk implementasjon av et eksperimentelt Hadoop-oppsett. Reed-Solomon-koder og trippel-replikasjon er de mekanismene det fokuseres på. Videre er mengden nettverkstrafikk og forløpt tid innenfor en filgjenopprettingsprosess testet og sammenlignet for de ulike mekanismene. I tillegg er faktorer som påvirker ytelsen tatt i betraktning, slik som filstørrelse og blokkstørrelse. Testoppsettet krever kablet Ethernet, et konfigurert Hadoop-cluster, en styrt nettverkssvitsj og et verktøy for nettverksanalyse.

Resultatene fra testene viser hvordan ulike faktorer påvirker cluster-ytelsen i Hadoop når en node feiler. Generelt så bekrefter resultatene teorien som er presentert. Både filgjenopprettingstid og nettverkstrafikk øker i takt med filstørrelsen. For Reed-Solomon-koder så øker filgjenopprettingstiden i takt med kodelengden og blokker av størrelse 128 MB gir totalt sett best ytelse. I tillegg er optimaliserte kodevarianter som forbedrer cluster-ytelsen presentert og foreslått som videre arbeid.

Acknowledgment

The work exposed in this master thesis would not have been possible without the help and support from my supervisor and professor at NTNU. I wish to express my sincere gratitude to supervisor Katina Kralevska for her continuous assistance, advice and knowledge sharing. I am also grateful to professor Danilo Gligoroski for sharing his expertise and for guidance. Their valuable feedback and supervision greatly increased the quality of the thesis.

I also want to thank Pål Sturla Sæther for great hardware support and for him being always available for me. Finally, I would like to thank my parents and friends for giving me encouragement and motivation throughout the work.

Contents

List of Figures

List of Tables

List of Acronyms

3-rep	triple-replication.
FEC	forward error correction.
GF	Galois field.
HDFS	Hadoop Distributed File System.
HTECs	HashTag Erasure Codes.
I/O	input/output.
IDC	International Data Corporation.
IoT	Internet of Things.
LRCs	Locally Repairable Codes.
MDS	Maximum Distance Separable.
MSR	Minimum Storage Regenerating.
OPS	Optical Packet Switching.
RGCs	Regenerating Codes.
RS	Reed-Solomon.
SRCs	self-repairing codes.
SSH	Secure Shell.
WAS	Windows Azure Storage.
ZB	zettabytes.

Chapter 1

Introduction

1.1 Motivation

Apache Hadoop is one of the most popular frameworks for big data analysis. It is open-source and used for distributed storage and processing of big data sets [Whi12]. Big data is a term for extremely large datasets that traditional data processing tools are inadequate to deal with them [IBM]. Smartphones, desktop computers, laptops, Internet of Things (IoT) devices etc. generate tremendous amounts of data. People go through their days communicating digitally, surfing the internet, shopping online, taking photos, sharing files, and by doing that they leave behind huge trails of data. Imagine billions of people doing this every day. The grand total is massive so finding ways to utilize this information is crucial, otherwise it is worthless. Based on reports from the International Data Corporation (IDC) the amount of digital data is predicted to reach 180 zettabytes (ZB) by 2025 [Cor]. This massive growth of digital data over just a short period of time is illustrated in Figure 1.1. A portion of this data has to be transferred and stored reliably.

Software and/or hardware failures in distributed storage systems can happen. For instance, more than 50 machine failure events were registered each day during a six-month experiment of monitoring Facebook's data-warehouse cluster that consists of thousands of nodes [RSG$^+$15]. 98.08% of the failures were classified as single failures. Additionally, failures in Hadoop are not considered exceptions, they are rather treated as the norm because clusters can consist of thousands of nodes where all have a probability of failure [Foub]. That is why there is a need for mechanisms for reliable storage of big data. Hadoop is a framework which introduces such mechanisms.

Reliable storage of data is very important since users want to have seamless access to data even when failures in the system occur. That is challenging especially when we talk about big amounts of data. As the amount of digital data rapidly increases there is also a question about storage cost. *How can corporations store data reliably and* *decrease the storage costs of data replication?* Hadoop introduces methods through different coding techniques for achieving this.

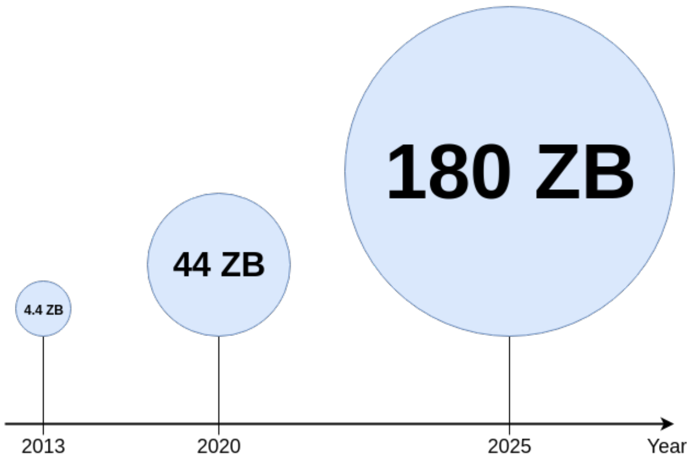


Figure 1.1: Growth of digital data.

1.2 Objective and Methodology

The overall objective of this master thesis work is to set up and configure an experimental Hadoop environment, conduct experiments for examining different techniques for data reliability in that environment and analyze the obtained results. The experiments will be a study of file recovery time and network traffic during file recovery for different erasure codes and triple-replication with varying block and file sizes. Additionally, understanding the principles of Hadoop for reliable data storage is an important objective. In particular, the primary focus is erasure coding in HDFS.

To achieve these objectives, a methodology based on a literature study and a comparative study is used. The literature study applies mainly to the presented background material. Research papers, selected chapters of books, articles and various web pages are selected and read thoroughly to acquire the relevant knowledge. The comparative study is conducted in order to evaluate the results of the experiments and to make an analysis. In addition, a practical methodology is used for the Hadoop implementation and to run experiments. In order to support the accomplishment of the experiments, various tools and methods are required. Hadoop 3.0.0, Ubuntu 16.04 LTS operating system, Wireshark network analysis tool and a port mirroring method are used to carry out the experiments.

1.3 Thesis Structure

Chapter 2 presents the relevant theoretical background of the thesis. Concepts of Hadoop, HDFS, replication and erasure coding are explained, along with a brief theoretical introduction to erasure coding. Related work is also presented in this chapter.

Chapter 3 gives a detailed guide on how the experimental Hadoop environment is set up. It is formed as a step-by-step installation guide including exact terminal commands and configuration parameters in order to set up a multi-node Hadoop cluster. Commands for utilizing HDFS and erasure coding in practice are also given.

Chapter 4 includes a test plan and details about how the experiments are conducted and what methods that are used. Then the results of the experiments are presented. Further, a comparison of measurements against different erasure codes and triple-replication is given. Finally, the test results are analyzed and discussed.

Chapter 5 concludes the thesis and proposes further work that can be done in continuation of this work or further work that explores the limitations set by the scope of this thesis.

Chapter 2 Theoretical Background

This chapter presents the theoretical background of the thesis. It explains the principles of Hadoop Distributed File System (HDFS), replication and erasure coding. The HDFS architecture is presented, as well as examples of the mathematics used to support erasure coding. Finally, related work is presented.

2.1 Hadoop and HDFS

Hadoop consists of a storage part (HDFS) and a processing part known as MapReduce [Whi12]. The focus in this thesis is the HDFS part. The main idea of HDFS is that it splits files into blocks which are then distributed across different nodes in a cluster. That is called distributed storage. All data blocks of a file have the same size except the last block which can be smaller. The number of blocks of a file is the exact multiple of the configured block size, plus an additional final block with the number of bytes that is left. HDFS is used to store big data sets and the default block size is 128 MB. If the block size is small like for instance in the Linux file system (4 KB), HDFS will generate huge amounts of overhead and traffic in order to manage the large number of blocks and metadata (data about the data) files [Fla]. That is not beneficial, so the block size is therefore set to 128 MB. Figure 2.1 illustrates how a 650 MB file is split into 5 data blocks of size 128 MB ($128 * 5 = 640$), plus an additional data block of size 10 MB.

2.1.1 HDFS Architecture

HDFS runs a master-slave architecture. The master's job is to manage the file system namespace and to control clients' access to files (read/write access). It also decides the mapping of split files to different blocks and handles all the metadata. These jobs are done by the NameNode, which is a piece of software running on the master node. No user data is transferred through the NameNode [Foub]. Figure 2.2 illustrates how the HDFS software is connected to the master node and the slave nodes, respectively.

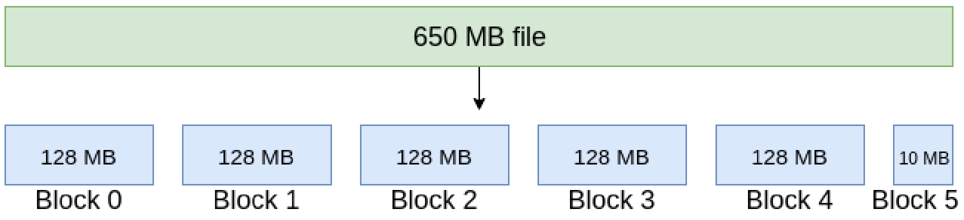


Figure 2.1: Data blocks in HDFS.

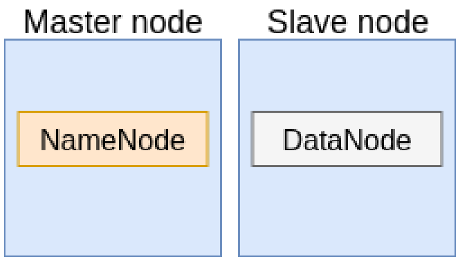


Figure 2.2: Master node and slave node in HDFS.

The slaves in HDFS have DataNodes installed, and they are responsible for the actual storage of data on the particular nodes. Usually, there is one DataNode per slave node in the cluster (DataNodes can contain multiple data blocks), but this is a matter of an architectural design choice. System engineers are free to take own design choices regarding the node cluster setup. DataNodes serve the clients' read/write requests and get instructions from the NameNode to perform replication, deletion and creation of data blocks [Foub]. Figure 2.3 gives an overview of the HDFS architecture.

Hadoop provides a possibility to place DataNodes on different *racks*. A rack is a collection of servers (with DataNodes installed) connected to the same network switch. Having several racks improves the performance of the cluster, but the default in Hadoop is to place all nodes on the same rack if not configured otherwise. This thesis operates with a one-rack cluster, which means that if the network goes down all the servers on the rack will be out of service.

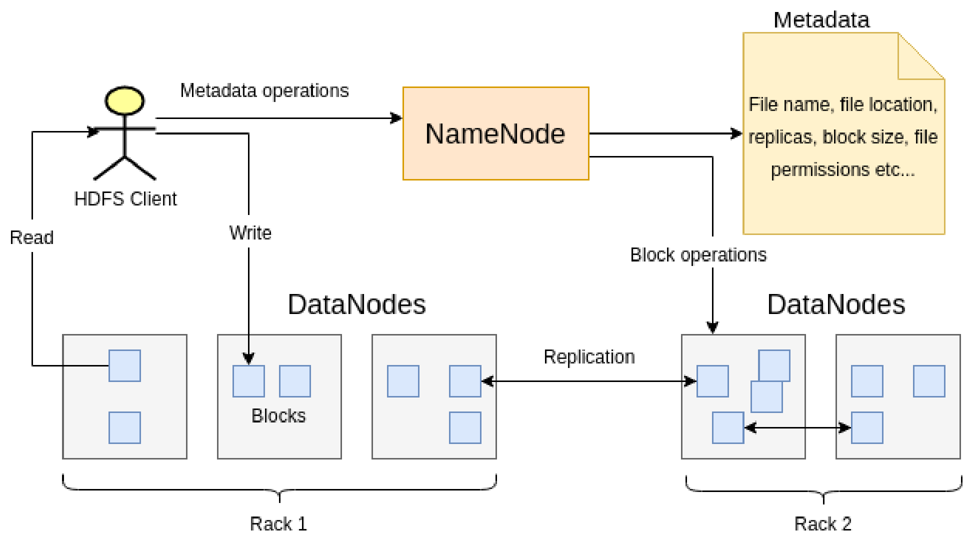


Figure 2.3: HDFS Architecture (adapted from [Foub]).

2.2 Data Storage with Replication

A conventional way to store data in distributed storage systems is replication. The default replication factor in Hadoop is 3. This means that the same data is stored across 3 different nodes. The availability of the data is therefore increased. The system can tolerate up to two failures, and still the data is available. However, this mechanism is not efficient for big data because the storage overhead is 200%. The storage overhead is defined as the ratio of the added redundancy and original data. Figure 2.4 illustrates providing data availability which is triple-replication over three nodes.

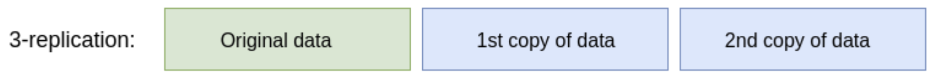


Figure 2.4: Triple-replication gives 200% storage overhead.

2.3 Data Storage with Erasure Coding

New techniques for reliable storage, such as erasure codes, are becoming more applicable than replication because they provide the same level of fault tolerance (number of tolerated node failures) for less storage overhead [Fouc]. In erasure coding, a file is split into k blocks and encoded with an *(n, k)* erasure code into n blocks where n-k=r are redundant blocks. The redundant data is a linear combination of the original data and it is added to the original data, i.e. systematic coding. Conventional Maximum Distance Separable (MDS) erasure codes guarantee that a file can be obtained by accessing any k out of the n nodes. This also guarantees that a block can be recovered by accessing and transferring any k out of the n blocks of the file. Thus, the system can tolerate up to r failures. Figure 2.5 illustrates the differences in storage overhead for triple-replication, i.e. a (3, 1) code, vs. a typical erasure coding that is used in Windows Azure [HSX$^+$12], i.e. an (9, 6) code. The storage overhead is calculated by o=r/k.

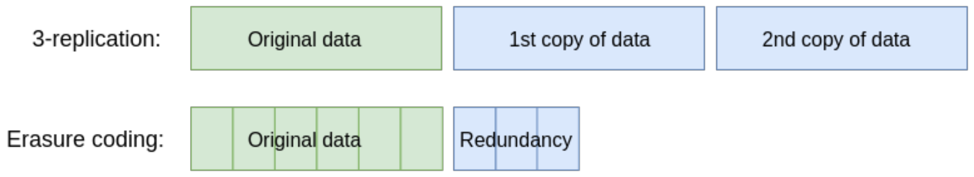


Figure 2.5: Data replication (3, 1) vs. erasure coding (9, 6). The storage overhead of triple-replication is 200%, while the storage overhead of erasure coding is 50%. Both offer a similar level of availability.

Even though erasure codes give less overhead, they require more CPU power, more network bandwidth and more time to recreate lost data compared to triple-replication.

2.3.1 Erasure Coding Example

The Hadoop framework takes care of calculations of erasure codes and other functionality in the background, so in practice knowing its internal workings is not needed. Despite that, it is important to understand what is going on "behind the scenes" in order to fully understand Hadoop and to use it correctly. The following presents a toy example just to illustrate the principle of erasure coding and to emphasize its possible advantages and disadvantages over replication. Further, in Section 2.3.2, a more detailed example with actual calculations is presented.

Figure 2.6 shows a data object which is split into two data blocks and stored in two different ways, both utilizing 4 servers. The first storage method is double-replication and the second storage method is a (4, 2) code where servers 3 and 4 store linear combinations of the original data. Note that the linear functions have to be calculated over a *finite field*. Read more about this in Section 2.3.2.

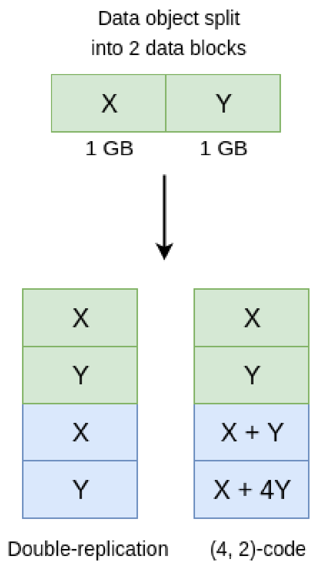


Figure 2.6: Two distributed storage methods of a data object which is split into two data blocks where original data is presented in green and redundancy data in blue.

Figure 2.7 illustrates what happens in both scenarios when two servers fail, respectively servers 1 and 3. In the first scenario data "X" is lost. In the second scenario the data held by servers 2 and 4 can be used to recalculate the temporarily lost data held by servers 1 and 3. This example shows that coding is a better choice in terms of availability of the data, even when the storage overhead is the same. Also for the same amount of added redundancy, erasure coding offers greater reliability.

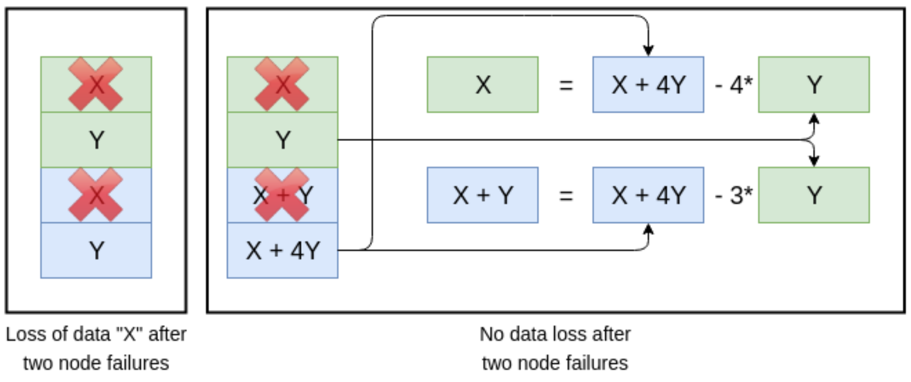


Figure 2.7: Data loss with replication and no data loss with erasure coding after two node failures.

Another thing to keep in mind is the repair cost when server(s) fail. Repair cost refers to the time and resources spent while regenerating lost data when a system node fails. Now, imagine that server 1 fails in both scenarios. In the first scenario, only server 3 has to be accessed in order to copy its contents back to server 1. In the second scenario, the data from two servers have to be accessed and decoding computations have to be performed in order to bring back the contents of server 1. Servers 2 and 4 can be used for the regeneration (see Figure 2.7), or if desired, servers 2 and 3. In the latter, the regeneration looks like this: $X = (X + Y) - Y$. This example shows a possible disadvantage of erasure coding compared to replication in terms of repair cost.

2.3.2 Encoding and Decoding of RS(4, 2)

Reed-Solomon (RS) codes are well known and widely used erasure codes. A simple example of RS(4, 2) encoding and decoding of an 8-bit file is presented below. The system has two nodes which hold the original data (d_1 and d_2) and two nodes which hold the parity data (r_1 and r_2). The parities (r1 and r2) can be used to recalculate the original data if it is lost due to hardware or software failures. Figure 2.8 shows the system's starting point.

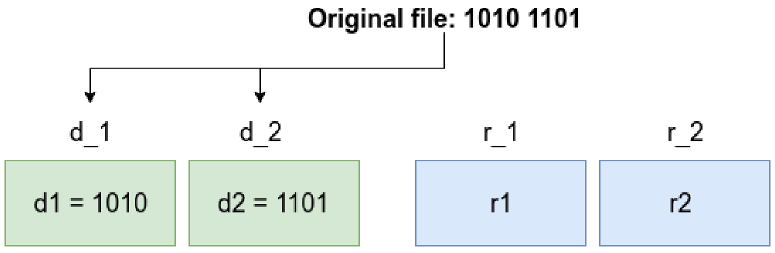
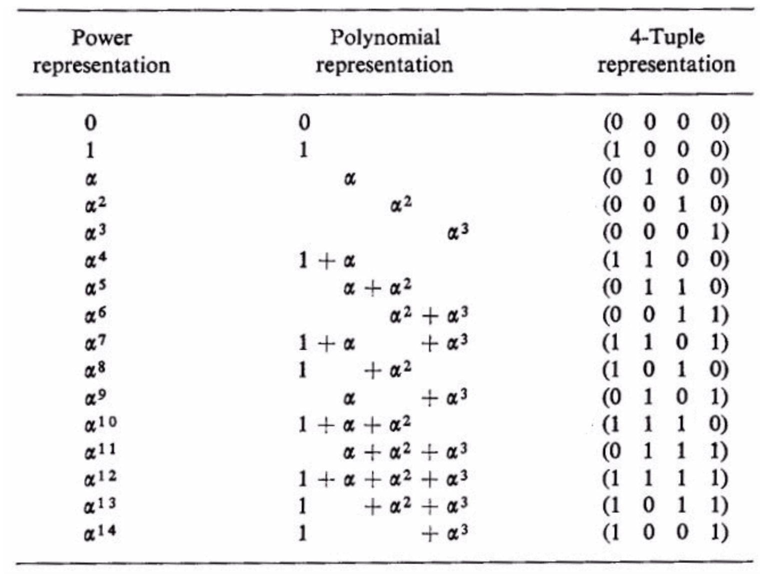


Figure 2.8: A file split into two data blocks (d1 and d2) and stored in d_1 and d_2. r_1 and r_2 will hold the parity blocks (r1 and r2) which will be calculated during the RS encoding.

Knowledge about Galois fields is essential in order to understand erasure coding. A Galois field (GF) is a finite field which contains a finite number of elements. Addition, subtraction, multiplication and division are possible operations within the field [Mat]. In our case, we have GF(2^4), a field with 16 elements. In Hadoop and other commercial implementations of erasure codes, the calculations are normally performed in GF(2^8) or in higher fields. The *order* of our field is $2^4 = 16$ since we operate with 4-bit elements in this example. The GF's irreducible polynomial is $p(X) = X^4 + X + 1$ [LC82]. An irreducible polynomial is a polynomial which is only divisible by itself and 1. In Figure 2.9 all elements of GF(2^4) are given. We use the rule in Equation 2.1 to derive the binary multiplication and the rule in Equation 2.2 to derive the binary division.



Power representation	Polynomial representation	4-Tuple representation
0	0	(0 0 0 0)
1	1	(1 0 0 0)
α	α	(0 1 0 0)
α^2	α^2	(0 0 1 0)
α^3	α^3	(0 0 0 1)
α^4	$1 + \alpha$	(1 1 0 0)
α^5	$\alpha + \alpha^2$	(0 1 1 0)
α^6	$\alpha^2 + \alpha^3$	(0 0 1 1)
α^7	$1 + \alpha + \alpha^3$	(1 1 0 1)
α^8	$1 + \alpha^2$	(1 0 1 0)
α^9	$\alpha + \alpha^3$	(0 1 0 1)
α^{10}	$1 + \alpha + \alpha^2$	(1 1 1 0)
α^{11}	$\alpha + \alpha^2 + \alpha^3$	(0 1 1 1)
α^{12}	$1 + \alpha + \alpha^2 + \alpha^3$	(1 1 1 1)
α^{13}	$1 + \alpha^2 + \alpha^3$	(1 0 1 1)
α^{14}	$1 + \alpha^3$	(1 0 0 1)

Figure 2.9: Three representations for the elements of GF(2^4) generated by $p(X) = X^4 + X + 1$ (taken from [LC82]).

$$\alpha^i \times \alpha^j = \alpha^{(i+j) \bmod 15} \tag{2.1}$$

$$\alpha^i / \alpha^j = \alpha^i \times \alpha^{15-j} = \alpha^{i+(15-j)} \tag{2.2}$$

Now the calculation of the RS encoding is ready to begin. In order to encode the file, the parity data (r1 and r2) are calculated, see Equations 2.3 and 2.4. C1 and C2 in Equation 2.4 are two constants which have to fulfill the requirements of Equation 2.5.

$$r1 = d1 \oplus d2 \tag{2.3}$$

$$r2 = (C1 \times d1) \oplus (C2 \times d2) \tag{2.4}$$

$$M = \begin{bmatrix} 1 & 1 \\ C1 & C2 \end{bmatrix}, \; where \; det(M) \neq 0 \tag{2.5}$$

C1 and C2 are set to 2 and 3 respectively. In binary numbers they are C1 = 0010, and C2 = 0011. Equations 2.6 and 2.7 show the encoding in binary numbers.

$$r1 = 1010 \oplus 1101 = 0111 \tag{2.6}$$

$$r2 = (0010 \times 1010) \oplus (0011 \times 1101) = 0101 \tag{2.7}$$

The original file should now be reliably stored and available. The system in Figure 2.8 can tolerate up to two failures. Now the scenario has changed as seen in Figure 2.10. Two nodes are down, so how can the RS code be used to recover the lost part of the original data (d2)? Equation 2.8 shows the recovery of d2.

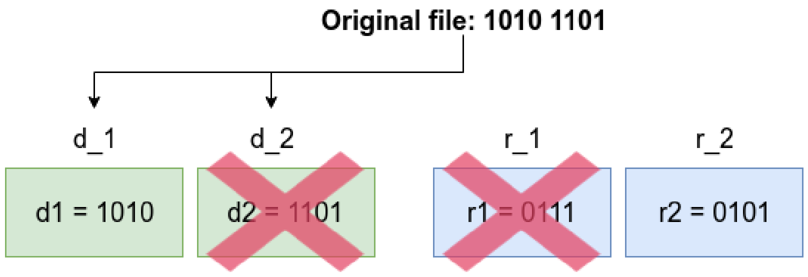


Figure 2.10: Two nodes are down in the system.

$$d2 = \frac{r2 \oplus (C1 \times d1)}{C2} \tag{2.8}$$

The data from d_1 and r_2 have to be accessed and transferred in order to recover d2, and then r1 can be recalculated. Equations 2.9 and 2.10 show the recovery in binary numbers. Finally, the file should be just as reliably stored and available as it was after the initial encoding.

$$d2 = \frac{0101 \oplus (0010 \times 1010)}{0011} = 1101 \quad Success! \tag{2.9}$$

$$r1 = d1 \oplus d2 = 1010 \oplus 1101 = 0111 \quad Success! \tag{2.10}$$

2.4 Related Work

There is a lot of theoretical work that has been done in the area of reliability in distributed storage. In particular, the area of constructing erasure codes for distributed storage systems has been well studied. Erasure coding theory was first introduced in the 1960s [PH13] and since then there has been a significant amount of research work concerning erasure codes and distributed storage. This section presents briefly some relevant work that has been done in the area.

Big data is closely connected to both distributed storage and erasure codes because it is necessary to store big data reliably. In recent years big data has become a complete subject which includes various frameworks, tools and techniques [ZE$^+$11]. In particular, the area of big data analysis has become exceedingly popular. For instance, [PBN12], [NBP$^+$13] and [Shi12] talk about the advantages of big data processing and analysis in Hadoop by utilizing the MapReduce programming model. MapReduce consists of a mapper and a reducer. Simply put, the mapper maps input data into a set of key/value pairs. The reducer reduces this set into a smaller set of values which is the useful data output.

Many cloud systems utilize Reed-Solomon (RS) codes because they are able to tolerate a large number of failures and provide high generality. [KBP$^+$12] defines a new class of RS codes that reduce the repair traffic and perform disk reads more effectively compared to standardized RS codes. A new erasure-coded storage system called "Hitchhiker" is introduced in [RSG$^+$15]. It is implemented in HDFS and runs "on top of" RS codes. Compared to RS-based systems they reduce both the disk input/output (I/O) and network traffic during reconstruction on node failure. Individual component failures in storage systems are the most studied type of failure. In contrast, [FLP$^+$10] provides a study of the overall availability behavior in large cloud-based storage systems. The impact of design choices, such as strategies for replication and data placement, are presented through statistical models.

[Pap14] states that erasure codes in large-scale data storage are associated with high repair cost. It displays the construction of new erasure coding variants to achieve the best possible reliability under different repair cost metrics. For instance some variants of MDS codes are presented to perform optimal repair of the data and/or parity nodes. [VRP$^+$18] introduces Clay codes, which are simplified constructions of Minimum Storage Regenerating (MSR) codes. MSR codes are designed to meet the current practical needs of data centers, and some of them have been practically implemented. For instance, they show reduced repair network traffic and repair time in comparison with RS codes.

Locally Repairable Codes (LRCs) address two important metrics in distributed storage systems, the number of contacted nodes during repair (locality) and the update complexity. For instance, the locality and the update complexity are balanced (equal) for all nodes in a system with Balanced LRCs [PJHO13, KGØ16a]. A variant of LRCs is Pyramid codes, an erasure coding policy which utilizes a pyramid-based scheme. [Vel18] compares the performance of Pyramid codes vs. RS codes for both encoding and reconstruction of files. Compared to RS codes, it shows that Pyramid codes reduce the network traffic during reconstruction. Windows Azure Storage (WAS) is a cloud storage system that uses erasure coding to keep the storage costs down [HSX+12]. WAS utilizes LRCs and the paper describes that these codes can reduce I/Os and bandwidth for repair reads. LRCs are also presented in [SAP+13]. They introduce a new novel erasure coding family that incorporates LRCs. The new codes are implemented in Facebook and provide higher reliability compared to RS codes because of faster repair of failures. [GHSY12] presents a thorough study of relations between different parameters which are required in data storage applications. The parameters include low locality for parity and data coordinates, small redundancy and large distance.

[DGW+10] discusses ways to transfer as little data as possible when nodes fail. They point out that common erasure-coded systems reconstruct all the encoded data to regenerate just one encoded data block. Regenerating Codes (RGCs) are introduced as an alternative to reduce the repair bandwidth. Functions of the stored data from the non-failed nodes will now be communicated through a new node. [OD11] state that RGCs, which utilize (n, k) erasure codes, need to contact at least k nodes to recreate a lost fragment. Further, data needs to be downloaded from n-1 nodes to fulfill the recreation. Moreover, they introduce self-repairing codes (SRCs), a new type of erasure codes which improves the maintenance process in storage systems.

New efficient erasure coding constructions, HashTag Erasure Codes (HTECs) and Balanced LRCs, are provided in [Kra16]. It investigates how these coding constructions are applied to different applications, emphasizing distributed storage systems, network coding and Optical Packet Switching (OPS) networks. HTECs reduce the repair bandwidth for single failures [KGJØ17]. In particular, compared to RS codes the repair bandwidth savings can go up to 70%. Erasure codes in OPS networks are also discussed in [BCKØ16] and [KOG15]. In particular, [BCKØ16] presents a case study of utilizing forward error correction (FEC) for OPS. They show that it can reduce packet loss and decoding errors when used correctly. [KOG15] presents a transport mechanism scheme for OPS networks. It exploits erasure coding benefits for efficient packet loss recovery.

In this thesis, the focus is not on LRCs or RGCs. Even though there are many practical and theoretical works related to erasure coding in distributed storage, still some aspects of RS codes implemented in HDFS have not been investigated. More precisely, how various block and file sizes affect the performance of the Hadoop cluster to support reliable data storage, which is what this thesis is motivated by.

Chapter 3

Experimental Hadoop Environment

One of the main objectives for the thesis is to set up an experimental Hadoop environment. This chapter is therefore formed as an installation guide on how to successfully set up a working multi-node Hadoop cluster. It starts with a single-node Hadoop setup, and expands to a multi-node setup. For simplicity, the guide covers a 5-node cluster setup (consisting of five physical Ubuntu machines), but one is free to add as many nodes as preferred following the same procedure. Note that some experiments conducted later in the thesis rely on more than five nodes. A total of 15 nodes are set up during this master thesis work. This chapter also presents some important commands and configurations in order to enable erasure coding and to correctly store files within HDFS. Everything presented in this chapter is a prerequisite to being able to run experiments presented in Chapter 4. The content of this chapter is inspired by the Hadoop documentation on The Apache Software Foundation's official website [Foua], but it is customized specifically to fit the thesis' objective and to fit architectural design choices.

3.1 Hadoop Cluster Setup

If you follow the whole upcoming guide, Figure 3.1 shows the architecture of the final Hadoop cluster setup. The Hadoop environment distinguishes between the HDFS layer (distributed storage layer) and the MapReduce layer (data processing layer). HDFS can operate without MapReduce, but not the other way around because the files which are processed and analyzed by the MapReduce programming model are stored in HDFS. The NameNode, SecondaryNameNode and DataNodes are in the HDFS layer and the ResourceManager and NodeManagers are in the MapReduce layer. Following this guide also gives you the opportunity to utilize MapReduce, but as mentioned before, only the HDFS layer components are part of the experiments later.

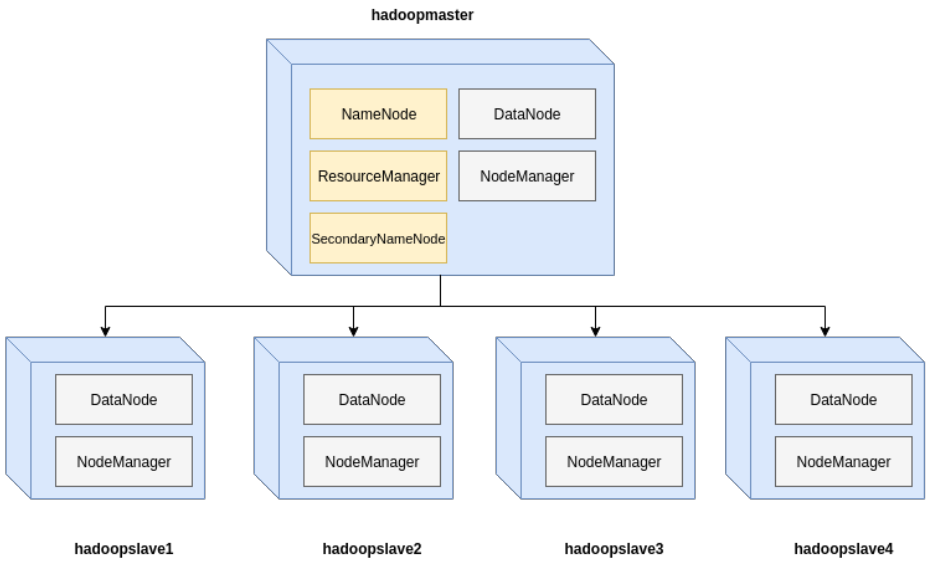


Figure 3.1: The final Hadoop architectural outcome of this guide.

3.1.1 Single-node Hadoop Cluster Setup

The following instructions cover a single-node Hadoop cluster setup for one node (server1). The exact same instructions are followed for the remaining four nodes.

1. Start by running an update and upgrade in the Ubuntu terminal. It updates the list of available packages and their versions and then installs newer versions of the packages you already have. See Figure 3.2.

```
ingvild@server1:~$ sudo apt-get update
ingvild@server1:~$ sudo apt-get upgrade
```

Figure 3.2: Ubuntu terminal commands: Update and upgrade.

2. Next, Oracle Java 8 has to be installed, see Figure 3.3. It is a required software when using Hadoop.

```
ingvild@server1:~$ sudo add-apt-repository ppa:webupd8team/java
ingvild@server1:~$ sudo apt-get update
ingvild@server1:~$ sudo apt-get install oracle-java8-installer

# Check the java version
ingvild@server1:~$ java -version
java version "1.8.0_161"
```

Figure 3.3: Ubuntu terminal commands: Install Oracle Java 8.

3. A Hadoop user (hduser) should be created for accessing HDFS and MapReduce, see Figure 3.4. It is recommended to set up a new user group (hadoop) in order to avoid security issues. Also, give correct and secure root privileges to hduser by adding the content in Figure 3.5 to the `/etc/sudoers` file.

```
# Add a new user group named Hadoop
ingvild@server1:~$ sudo addgroup hadoop
# Add a new user and set password and other optional info when prompted
ingvild@server1:~$ sudo adduser --ingroup hadoop hduser
# Log in to root
ingvild@server1:~$ sudo su root
# Open the /etc/sudoers file
root@server1:~$ sudo gedit /etc/sudoers
```

Figure 3.4: Ubuntu terminal commands: Create Hadoop user.

/etc/sudoers

```
# User privilege specification
hduser ALL=(ALL:ALL) ALL
```

Figure 3.5: Configurations in /etc/sudoers.

4. Secure Shell (SSH) needs to be installed and configured. Hadoop uses SSH to access its nodes. For our single-node Hadoop setup, we therefore need to configure SSH access to the node's `localhost`, see Figure 3.6.

```
# Install an OpenSSH server
ingvild@server1:~$ sudo apt-get install openssh-server
# Log in to hduser
ingvild@server1:~$ sudo su hduser
# Generate SSH key for the hduser account
hduser@server1:~$ ssh-keygen -t rsa -P ""
# Add the key to the list of authorized keys
hduser@server1:~$ cat ~/.ssh/id_rsa.pub » ~/.ssh/authorized_keys
```

Figure 3.6: Ubuntu terminal commands: SSH configuration.

5. Hadoop does not work over IPv6, it only works over IPv4. Therefore, IPv6 needs to be disabled. Update the `/etc/sysctl.conf` file with the content in Figure 3.8 and run a command to reboot the machine (see Figure 3.7).

```
# Open the /etc/sysctl.conf file
hduser@server1:~$ sudo gedit /etc/sysctl.conf
# Reboot machine
hduser@server1:~$ sudo reboot
```

Figure 3.7: Ubuntu terminal commands: Open file for disabling IPv6 and reboot command.

```
/etc/sysctl.conf

# disable ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Figure 3.8: Configurations in /etc/sysctl.conf.

6. You have to decide on which Hadoop version you want to use. The thesis utilizes Hadoop version 3.0.0 because it supports erasure coding and it is now considered Generally Available (GA). It means that it is a more stable and safe version to use than the previous -alpha and -beta versions. Its release date was 13 December 2017 and it supports six built-in storage policies. Replication, RS(5, 3), RS(9, 6) and RS(14, 10) will be the primary focus, and they will be tested through experiments. If you have to update an already installed Hadoop cluster

to a more recently developed version, you can follow a simple guide attached in Appendix A. Now, go to the Apache Hadoop mirror site (http://apache.uib.no/hadoop/common/) and download the preferable `hadoop-X.Y.Z-src.tar.gz` file. Then follow the commands shown in Figure 3.9.

```
# Move the downloaded folder to the Hadoop directory
hduser@server1:~$ sudo mv '/home/ingvild/Downloads/hadoop-3.0.0' /us-
r/local/hadoop
# Assign ownership of the Hadoop directory to hduser
hduser@server1:~$ sudo chown hduser:hadoop -R /usr/local/hadoop
# Make directory for NameNode
hduser@server1:~$ sudo mkdir -p /usr/local/hadoop/hadoopData/hdfs/na-
menode
# Make directory for DataNode
hduser@server1:~$ sudo mkdir -p /usr/local/hadoop/hadoopData/hdfs/-
datanode
```

Figure 3.9: Ubuntu terminal commands: Hadoop components setup.

7. Next you have to add the contents of Figure 3.10 at the end of the `.bashrc` file. Open it with the command: `sudo gedit .bashrc`.

.bashrc

```
# -- HADOOP VARIABLES -- #
export JAVA_HOME=/usr/lib/jvm/java-8-oracle
export HADOOP_HOME=/usr/local/hadoop
export PATH=$PATH:$HADOOP_HOME/bin
export PATH=$PATH:$HADOOP_HOME/sbin
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib"
export PATH=$PATH:/usr/local/hadoop/bin/
# -- HADOOP VARIABLES -- #
```

Figure 3.10: Configurations in .bashrc.

8. The next step is to configure some Hadoop configuration files. Add the contents in Figures 3.11-3.15 to the respective files. Logged in as hduser, you open the files by using this command in the Ubuntu terminal: `sudo gedit /usr/local/hadoop/etc/hadoop/*filename*`.

   a) `hadoop-env.sh`: This file contains environmental variable settings in Hadoop, such as where log files are stored. See configurations in Figure 3.11.

hadoop-env.sh

```
#The Java implementation to use
export JAVA_HOME='/usr/lib/jvm/java-8-oracle'
```

Figure 3.11: Configurations in hadoop-env.sh.

   b) `core-site.xml`: This file contains configurations that override the default values for core properties in Hadoop. See configurations in Figure 3.12. NameNode runs on the `localhost` on port 9000.

core-site.xml

```
<property>
    <name>fs.default.name</name>
    <value>hdfs://localhost:9000</value>
</property>
```

Figure 3.12: Configurations in core-site.xml.

   c) `hdfs-site.xml`: This file enables you to override several default values that control HDFS. See configurations in Figure 3.13. The `dfs.replication` property sets the default replication factor that is used for each data block of a file. `dfs.namenode.name.dir` sets the path to the directory where NameNode stores the metadata of the file system. `dfs.datanode.data.dir` sets the path to the directory where DataNode stores the actual blocks of file data.

hdfs-site.xml

```
<property>
    <name>dfs.replication</name>
    <value>1</value>
</property>

<property>
    <name>dfs.namenode.name.dir</name>
    <value>file:/usr/local/hadoop/hadoopData/hdfs/namenode</value>
</property>

<property>
    <name>dfs.datanode.data.dir</name>
    <value>file:/usr/local/hadoop/hadoopData/hdfs/datanode</value>
</property>
```

Figure 3.13: Configurations in hdfs-site.xml.

d) `yarn-site.xml`: This file enables you to override several default values that control the components used by YARN. YARN is one of the components in MapReduce 2.0. See configurations in Figure 3.14.

yarn-site.xml

```
<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>

<property>
<name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
<value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

Figure 3.14: Configurations in yarn-site.xml.

e) `mapred-site.xml`: This file enables you to override several default values that control the job execution components in MapReduce. See configurations in Figure 3.15.

```
mapred-site.xml

<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

Figure 3.15: Configurations in mapred-site.xml.

9. Now we are ready to format (done only once) and start Hadoop (see Figure 3.16). To see whether Hadoop has started correctly the `jps` command should output six services that are important to successfully run Hadoop as a single-node cluster.

```
# Formatting NameNode
hduser@server1:~$ hdfs namenode -format

# Start HDFS daemons/services
hduser@server1:~$ start-dfs.sh

# Start MapReduce daemons/services
hduser@server1:~$ start-yarn.sh

# See if Hadoop started correctly
hduser@server1:~$ jps
30148 DataNode
30023 NameNode
31191 Jps
30792 NodeManager
30651 ResourceManager
30348 SecondaryNameNode
```

Figure 3.16: Ubuntu terminal commands: Hadoop startup.

10. In the same way as the starting commands you are able to stop the cluster by using the commands shown in Figure 3.17. You can also monitor the NameNode and the ResourceManager through a web interface in your web browser. Go to http://localhost:8088 for monitoring the ResourceManager and go to http://localhost:9870 for monitoring the NameNode.

```
hduser@server1:~$ stop-dfs.sh
hduser@server1:~$ stop-yarn.sh

hduser@server1:~$ jps
31729 Jps
```

Figure 3.17: Ubuntu terminal commands: Hadoop shutdown.

3.1.2 Multi-node Hadoop Cluster Setup

Now there are five individual single-node Hadoop clusters where all serve as both a master node and a slave node. Next, they are tangled together to become one multi-node Hadoop cluster with only one master node, see Figure 3.18. The master node serves as both a master and a slave (not required), so there are five slave nodes in the cluster in total. Instructions for setting up the multi-node cluster are presented below.

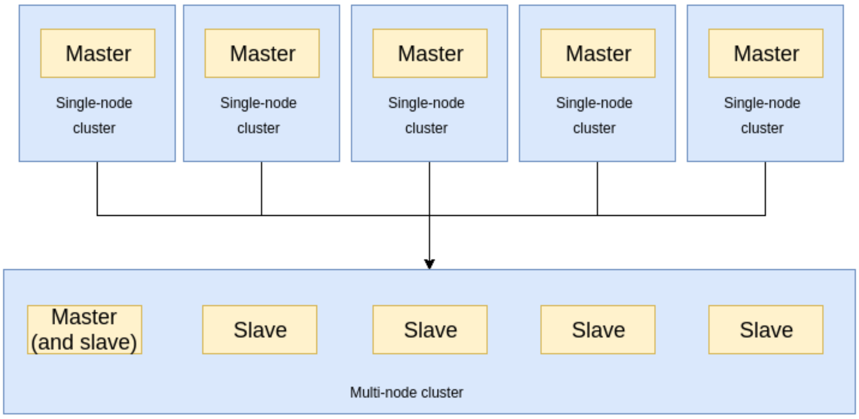


Figure 3.18: Going from single-node Hadoop clusters to a multi-node Hadoop cluster.

1. The following should be done on **all 5 nodes**:

   a) Update the `/etc/hosts` file with the nodes' hostnames and corresponding inet addresses, see Figure 3.19. You find the addresses by typing `ifconfig` in the terminals and you can freely choose the hostnames. You also have to choose which server that should be the Hadoop master node.

/etc/hosts

```
129.241.208.152 hadoopmaster
129.241.209.211 hadoopslave1
129.241.209.200 hadoopslave2
129.241.209.178 hadoopslave3
129.241.208.247 hadoopslave4
```

Figure 3.19: Configurations in /etc/hosts.

b) Update the `/etc/hostname` file with the node's corresponding hostname. For instance, just put `hadoopslave1` at the beginning of the file.

c) `core-site.xml`: Replace `localhost` with `hadoopmaster`.

d) `hdfs-site.xml`: Replace the value `1` with `3`. It sets the default replication factor.

e) `mapred-site.xml`: Replace the name with `mapred.job.tracker` and replace the value with `hadoopmaster:54311`.

f) `yarn-site.xml`: Add the contents of Figure 3.20 to the file.

yarn-site.xml

```
<property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>hadoopmaster:8025</value>
</property>

<property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>hadoopmaster:8030</value>
</property>

<property>
    <name>yarn.resourcemanager.address</name>
    <value>hadoopmaster:8050</value>
</property>
```

Figure 3.20: Configurations in yarn-site.xml for multi-node setup.

g) Reboot all nodes: `sudo reboot`.

2. The commands in Figure 3.21 should be done on **all hadoopslave nodes**. `hadoopslave2` is used as an example. In `hdfs-site.xml`, remove the property `dfs.namenode.name.dir` because now the master node is the only node that should have a running NameNode.

```
hduser@hadoopslave2:~$ sudo rm -rf /usr/local/hadoop/hadoopData
hduser@hadoopslave2:~$ sudo mkdir -p /usr/local/hadoop/hadoopData/hdf-
s/datanode
hduser@hadoopslave2:~$ sudo chown -R hduser:hadoop /usr/local/hadoop
hduser@hadoopslave2:~$ sudo gedit /usr/local/hadoop/etc/hadoop/hdfs-
site.xml
```

Figure 3.21: Ubuntu terminal commands: Hadoopslave specific commands.

3. The following should be done on the **hadoopmaster node only**:

   a) Go to `/usr/local/hadoop/etc/hadoop/masters` and paste "`hadoopmaster`" at the beginning of the file.

   b) In the `/usr/local/hadoop/etc/hadoop/slaves` file, remove `localhost` and add the contents of Figure 3.22. As mentioned, `hadoopmaster` is also a slave (recall Figure 3.1 and Figure 3.18).

/usr/local/hadoop/etc/hadoop/slaves

```
129.241.208.152 hadoopmaster
129.241.209.211 hadoopslave1
129.241.209.200 hadoopslave2
129.241.209.178 hadoopslave3
129.241.208.247 hadoopslave4
```

Figure 3.22: Configurations in /slaves.

   c) Next you can configure passwordless SSH to control all the slaves from the master node, see Figure 3.23. This will make it easy to navigate between the slave nodes without typing the password every time. After the configuration, use the command `ssh *hostname*` to get access to the respective slave's command line.

```
hduser@hadoopmaster:~$ sudo ssh-copy-id -i ~/.ssh/id_rsa.pub hduser@hadoopmaster
hduser@hadoopmaster:~$ sudo ssh-copy-id -i ~/.ssh/id_rsa.pub hduser@hadoopslave1
hduser@hadoopmaster:~$ sudo ssh-copy-id -i ~/.ssh/id_rsa.pub hduser@hadoopslave2
hduser@hadoopmaster:~$ sudo ssh-copy-id -i ~/.ssh/id_rsa.pub hduser@hadoopslave3
hduser@hadoopmaster:~$ sudo ssh-copy-id -i ~/.ssh/id_rsa.pub hduser@hadoopslave4
```

Figure 3.23: Ubuntu terminal commands: Hadoopmaster specific commands.

d) Finally, you have to format the NameNode (done only once) and then you can start the cluster, see Figure 3.24. Similar to the starting command, you can stop the whole cluster with `stop-all.sh` from the `hadoopmaster`, but you can also stop (and start) all the slaves individually. The services that should now run on the nodes are shown in Figure 3.25 and you can see that they correspond to the architecture presented initially in Figure 3.1.

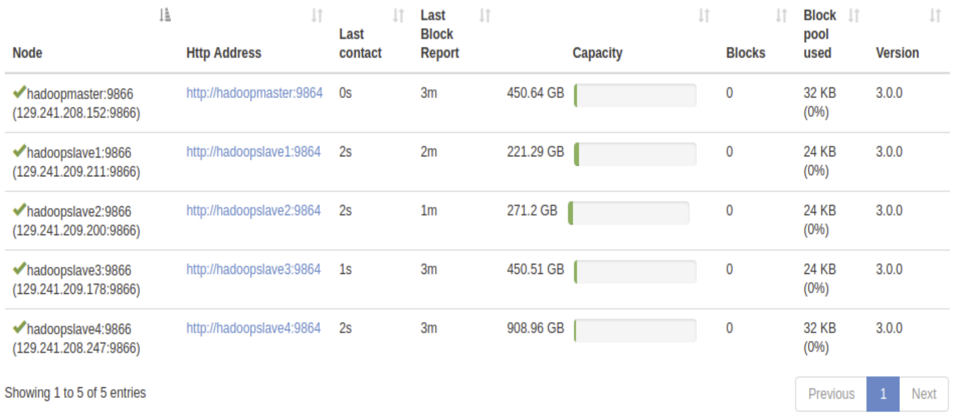
```
hduser@hadoopmaster:~$ hadoop namenode -format
hduser@hadoopmaster:~$ start-all.sh
```

Figure 3.24: Ubuntu terminal commands: Starting cluster.

You should now have a fully working 5-node Hadoop cluster. Figure 3.26 shows 5 live nodes in the web interface of the NameNode (http://localhost:9870) with nothing stored within HDFS yet. Next, see Section 3.2 for HDFS and erasure coding specific commands to start using the Hadoop installation in practice.

```
hduser@hadoopmaster:~$ jps
32083 DataNode
32740 NodeManager
31956 NameNode
13189 Jps
32599 ResourceManager
32280 SecondaryNameNode
hduser@hadoopslave1:~$ jps
7858 NodeManager
29973 DataNode
15289 Jps
hduser@hadoopslave2:~$ jps
9825 NodeManager
26173 DataNode
9069 Jps
hduser@hadoopslave3:~$ jps
10209 NodeManager
19111 Jps
31656 DataNode
hduser@hadoopslave4:~$ jps
9475 Jps
21844 DataNode
4485 NodeManager
```

Figure 3.25: Ubuntu terminal commands: Running 5-node cluster.



Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓hadoopmaster:9866 (129.241.208.152:9866)	http://hadoopmaster:9864	0s	3m	450.64 GB	0	32 KB (0%)	3.0.0
✓hadoopslave1:9866 (129.241.209.211:9866)	http://hadoopslave1:9864	2s	2m	221.29 GB	0	24 KB (0%)	3.0.0
✓hadoopslave2:9866 (129.241.209.200:9866)	http://hadoopslave2:9864	2s	1m	271.2 GB	0	24 KB (0%)	3.0.0
✓hadoopslave3:9866 (129.241.209.178:9866)	http://hadoopslave3:9864	1s	3m	450.51 GB	0	24 KB (0%)	3.0.0
✓hadoopslave4:9866 (129.241.208.247:9866)	http://hadoopslave4:9864	2s	3m	908.96 GB	0	32 KB (0%)	3.0.0

Showing 1 to 5 of 5 entries

Figure 3.26: Five live nodes after successful multi-node Hadoop setup.

3.2 HDFS Storage and Erasure Coding Configuration

HDFS does not directly map to the Unix file system because it is a logical file system [Spa]. For instance, you can not change directory (`cd`), you must specify the whole path. Another example is that it is not possible to display the contents of a file in a text editor (e.g. `sudo gedit`). Figure 3.27 presents some HDFS specific commands in order to store files within HDFS, enable erasure coding and to set an erasure coding policy on a specific HDFS directory. Read the comments carefully. Note that if you do not set a specific storage policy on a directory, Hadoop will use triple-replication which is the default storage policy.

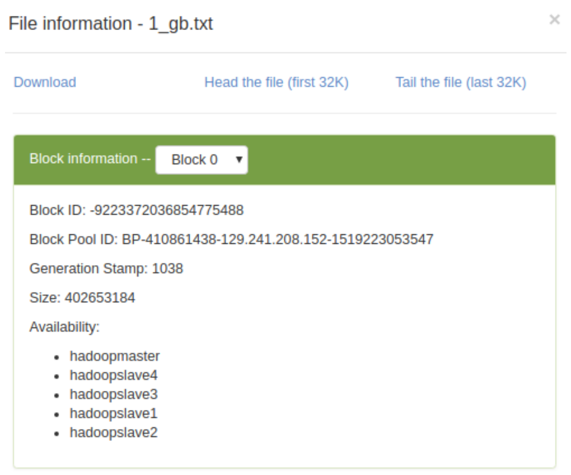
Figures 3.28, 3.29 and 3.30 demonstrate that Hadoop automatically preserves file data when the system discovers node failure(s) and when the number of failures does not exceed the fault tolerance of the storage policy. A file (`1_gb.txt`) is encoded with RS(5, 3) and the data is available at all the nodes in the cluster with three data nodes (original data) and two parity nodes (redundant data), see Figure 3.28. Then, `hadoopslave3` and `hadoopslave4` are manually shut down and marked as "down" in the web interface (see Figure 3.29). NameNode discovers this and is now able to coordinate the regeneration of the temporarily lost data that resided on `hadoopslave3` and `hadoopslave4`. Finally, all the file data is available at `hadoopslave1`, `hadoopslave2` and `hadoopmaster` (see Figure 3.30). No file data was lost during the demonstration, but the system cannot tolerate any additional node failures without losing some parts of the original data. Note that the RS(5, 3) storage policy can tolerate up to two node failures as it is explained in Section 2.3.

By default, Hadoop uses 10.5 minutes to mark a node as "down" when it is shut down or when it fails due to hardware or software failures. When testing node failures, it is more sufficient to have a shorter "time-to-down". You can change this by editing the value of the `dfs.namenode.heartbeat.recheck-interval` property in the `hdfs-site.xml` file on the `hadoopmaster` node. Note that the unit of the value is in milliseconds.

If you want to create a random `.txt` file of a particular size for testing purposes you can run a simple command in the terminal (`base64 /dev/urandom | head -c *number of bits* > *filename*.txt`) and then put it in HDFS. Also, if you want to set a particular block size on a file in HDFS (it is 128 MB by default) you can add `hadoop fs -D dfs.block.size=*number of bytes*` at the beginning of the `-put` command when you put files in HDFS. You can also override the default block size value in the `hdfs-site.xml` configuration file.

```
# Enable RS(5, 3). Note that Hadoop uses another notation for the RS codes
hduser@hadoopmaster:~$ hdfs ec -enablePolicy -policy RS-3-2-1024k

# Make HDFS directory "testDir"
hduser@hadoopmaster:~$ hadoop fs -mkdir /user/hduser/testDir

# Set erasure coding policy on testDir
hduser@hadoopmaster:~$ hdfs ec -setPolicy -path /user/hduser/testDir
-policy RS-3-2-1024k

# Put a locally stored 1 GB file (1_gb.txt) in testDir
hduser@hadoopmaster:~$ hadoop fs -put /home/ingvild/Docu-
ments/1_gb.txt /user/hduser/testDir

# Check the contents of testDir
hduser@hadoopmaster:~$ hadoop fs -ls /user/hduser/testDir

# See the contents of 1_gb.txt in the terminal
hduser@hadoopmaster:~$ hadoop fs -cat /user/hduser/testDir/1_gb.txt

# Check the policy set on a file (or possibly a directory)
hduser@hadoopmaster:~$ hdfs ec -getPolicy -path /user/hduser/test-
Dir/1_gb.txt
```

Figure 3.27: Ubuntu terminal commands: HDFS storage and erasure coding setup.



Block ID: -9223372036854775488

Block Pool ID: BP-410861438-129.241.208.152-1519223053547

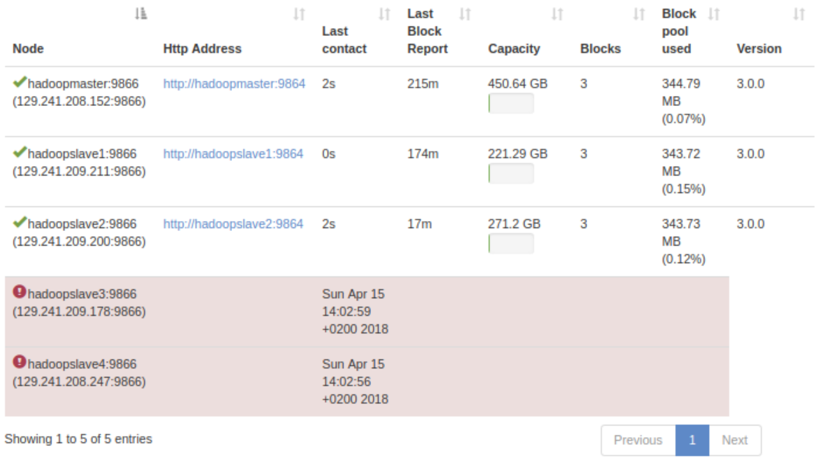
Generation Stamp: 1038

Size: 402653184

Availability:

- hadoopmaster
- hadoopslave4
- hadoopslave3
- hadoopslave1
- hadoopslave2

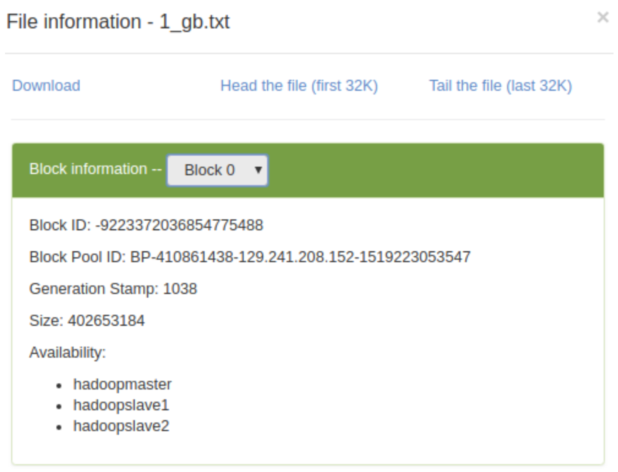
Figure 3.28: Node availability for 1_gb.txt before node failures.



Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓hadoopmaster:9866 (129.241.208.152:9866)	http://hadoopmaster:9864	2s	215m	450.64 GB	3	344.79 MB (0.07%)	3.0.0
✓hadoopslave1:9866 (129.241.209.211:9866)	http://hadoopslave1:9864	0s	174m	221.29 GB	3	343.72 MB (0.15%)	3.0.0
✓hadoopslave2:9866 (129.241.209.200:9866)	http://hadoopslave2:9864	2s	17m	271.2 GB	3	343.73 MB (0.12%)	3.0.0
❗hadoopslave3:9866 (129.241.209.178:9866)		Sun Apr 15 14:02:59 +0200 2018					
❗hadoopslave4:9866 (129.241.208.247:9866)		Sun Apr 15 14:02:56 +0200 2018					

Showing 1 to 5 of 5 entries

Figure 3.29: Two node failures in the 5-node Hadoop cluster.



File information - 1_gb.txt

Block ID: -9223372036854775488

Block Pool ID: BP-410861438-129.241.208.152-1519223053547

Generation Stamp: 1038

Size: 402653184

Availability:

- hadoopmaster
- hadoopslave1
- hadoopslave2

Figure 3.30: Node availability for 1_gb.txt after two node failures.

Chapter 4

Experiment and Analysis of Erasure Coding and Replication

This chapter presents how experiments are conducted and which tools and methods are used to obtain the results. It gives a test plan and corresponding results. Finally, the results are analyzed and discussed.

4.1 Experiment

The storage policies that are tested and compared are triple-replication (3-rep), RS(5, 3), RS(9, 6) and RS(14, 10). They are tested for three different file sizes (2.5 GB, 5 GB and 10 GB) and for three different block sizes (64 MB, 128 MB and 256 MB). Measurements include time of recovery of files (in seconds) and network traffic during recovery of files (in GB). More details about performing the measurements are given in Section 4.1.1. The measurements are managed and collected by an additional node which is not part of the Hadoop cluster. Table 4.1 compares the storage policies in terms of fault tolerance and storage overhead. Table 4.2 shows the HDFS nodes' specifications and the network speed of the Hadoop rack. The specifications are the same for all nodes in the cluster which consists of 15 physical nodes.

Table 4.1: Storage policies comparison.

Storage Policy	**Fault Tolerance**	**Storage Overhead**
3-rep	2	200%
RS(5, 3)	2	$\approx$ 67%
RS(9, 6)	3	50%
RS(14, 10)	4	40%

Table 4.2: Node specifications and network speed of the Hadoop rack.

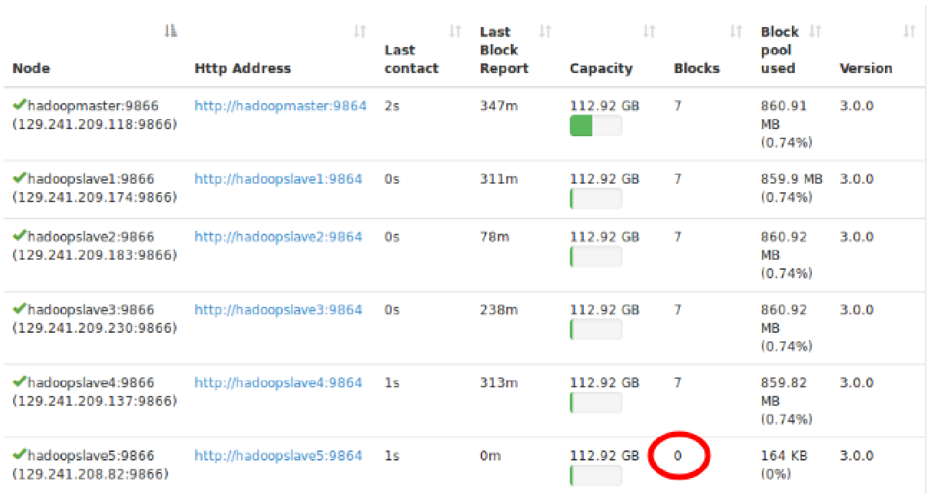
Disk Type	SSD
CPU Cores	2
Cache	3 MB SmartCache
Disk Size	128 GB
Memory Size	4 GB
Max Memory Bandwidth	34.1 GB/s
Operating System	Ubuntu 16.04 LTS (64-bit)
Network Speed	1000 Mb/s

Table 4.3 lists the experiments which are conducted in the thesis, in total 36 individual experiments. They are tested with node failures where the nodes are manually shut down. In all experiments, exactly one node is shut down because single failures are the most common type of failures in distributed storage as it was pointed out in Section 1.1 [RSG+15]. The node that is shut down is randomly chosen because in any case the system has to access and transfer any k out of the n blocks of the file due to the MDS property of the used erasure codes. The exception is the `hadoopmaster` node. It is never shut down because NameNode resides on that node. In every experiment, only one file is stored in HDFS to avoid interference of the testing results by a simultaneous recovery of other files. Also, for each storage policy, one additional node will be participating in the experiments. This enables correct comparison of the recovery time between 3-rep and RS codes. The number of participating nodes for 3-rep is therefore 4 nodes. Likewise, for RS(5, 3), RS(9, 6) and RS(14, 10) the number of participating nodes is 6, 10 and 15 nodes, respectively.

Next, a walk-through of the execution of experiment number 11 is presented. First, 5 nodes are turned on and marked as "in service" in HDFS. Then, a text file of size 2.5 GB filled with random text is stored in HDFS with RS(5, 3) storage policy and with 128 MB block size. The file is encoded and stored in the 5 nodes. Then one additional node is turned on and is a part of the experiment's participating nodes. As seen in Figure 4.1, it has no data stored and is "free" to be allocated to the file's available nodes when one of the other fails. Further, one of the available nodes is shut down and after a few seconds marked as "down". From this point, the time of recovery and network traffic are measured and noted. Then the file is deleted from HDFS and a next experiment is performed.

Table 4.3: Test plan for 36 experiments including different storage policies, file sizes and block sizes.

Experiment #	Policy	File Size (GB)	Block Size (MB)
1-3	3-rep	2.5	64/128/256
4-6	3-rep	5	64/128/256
7-9	3-rep	10	64/128/256
10-12	RS(5, 3)	2.5	64/128/256
13-15	RS(5, 3)	5	64/128/256
16-18	RS(5, 3)	10	64/128/256
19-21	RS(9, 6)	2.5	64/128/256
22-24	RS(9, 6)	5	64/128/256
25-27	RS(9, 6)	10	64/128/256
28-30	RS(14, 10)	2.5	64/128/256
31-33	RS(14, 10)	5	64/128/256
34-36	RS(14, 10)	10	64/128/256



Node	Http Address	Last contact	Last Block Report	Capacity	Blocks	Block pool used	Version
✓hadoopmaster:9866 (129.241.209.118:9866)	http://hadoopmaster:9864	2s	347m	112.92 GB	7	860.91 MB (0.74%)	3.0.0
✓hadoopslave1:9866 (129.241.209.174:9866)	http://hadoopslave1:9864	0s	311m	112.92 GB	7	859.9 MB (0.74%)	3.0.0
✓hadoopslave2:9866 (129.241.209.183:9866)	http://hadoopslave2:9864	0s	78m	112.92 GB	7	860.92 MB (0.74%)	3.0.0
✓hadoopslave3:9866 (129.241.209.230:9866)	http://hadoopslave3:9864	0s	238m	112.92 GB	7	860.92 MB (0.74%)	3.0.0
✓hadoopslave4:9866 (129.241.209.137:9866)	http://hadoopslave4:9864	1s	313m	112.92 GB	7	859.82 MB (0.74%)	3.0.0
✓hadoopslave5:9866 (129.241.208.82:9866)	http://hadoopslave5:9864	1s	0m	112.92 GB	0	164 KB (0%)	3.0.0

Figure 4.1: Experiment number 11 with RS(5, 3) has 5 available nodes and one additional free node (hadoopslave5).

4.1.1 Obtaining Time of Recovery and Network Traffic During Recovery

Time of recovery (recovery duration) refers to the time spent from the moment when a node is marked as "down" and until the file data is recovered and placed in a new node. The reallocation of a new node (when another fails) makes the storage of a particular file as reliable as it can be with the implemented storage policy, it optimizes the storage policy. In case of 3-rep, time of recovery means the time from the moment when one node (out of three) is shut down and until the file data is copied to a new node. In case of RS codes, the time of recovery means the time from the moment when one node is shut down and until the file data is recovered and a new node is allocated so that there are always enough available nodes to support the RS codes. *Network traffic during recovery* refers to the amount of network traffic transferred from the available nodes to the newly added node where the lost data is recovered during the time interval measured as the time of recovery.

Wireshark is used to measure the time of recovery and the network traffic during recovery. It is a network analysis tool which displays network traffic in real-time with detailed information about what is happening in the network at a low level [Wir]. Port mirroring is used to capture all the network traffic between the participating nodes in the recovery process. It is a method used on a managed network switch to copy all incoming and outgoing traffic on specified switch ports to a selected port where a monitoring device is attached. On that device a network analysis tool is installed, in this case Wireshark, which displays the copied traffic flowing through the switch. Figure 4.2 illustrates the test environment set up for the experiments and how the network switch, the HDFS nodes, Wireshark and the monitoring device are connected.

A managed switch (in this case HP ProCurve 2824) is configured to support port mirroring. Figure 4.3 shows a screenshot of the menu interface of the switch which is accessed through the switch console. It is set to monitor port 1-5 and port 15 through port 17 (monitoring port). That specific setup is used to monitor the traffic during the RS(5, 3) experiments. The allocation of ports to HDFS nodes can be freely chosen, but always be aware of which nodes that are connected to which ports. In particular, port 1-15 are set to be monitored when testing RS(14, 10).

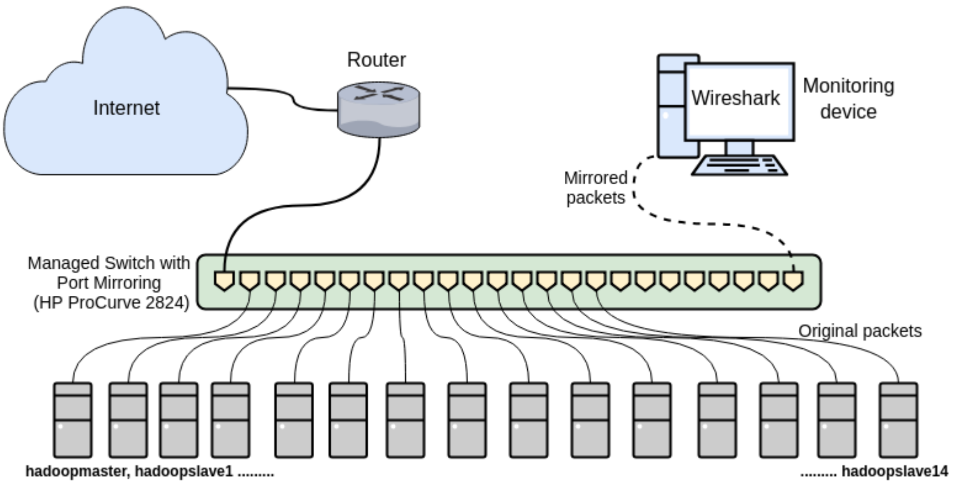
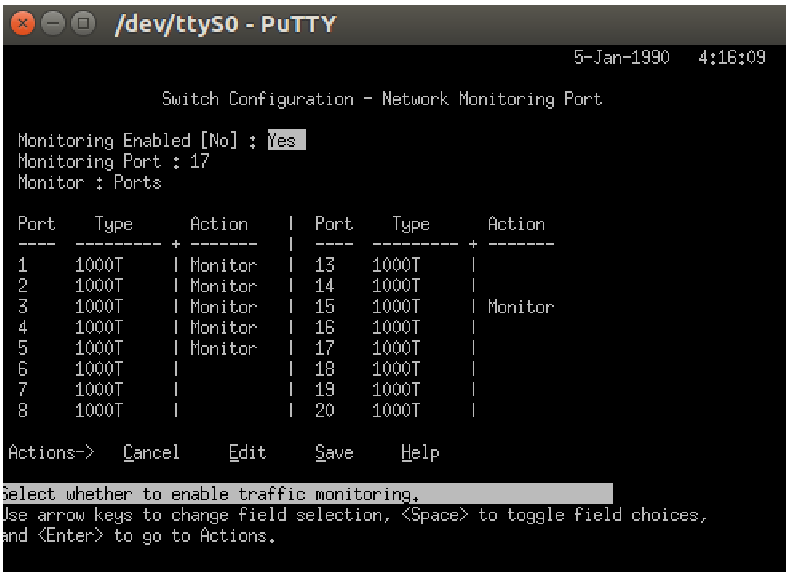


Figure 4.2: Test setup including port mirroring and Wireshark.



```
                                                  5-Jan-1990   4:16:09

                 Switch Configuration - Network Monitoring Port

 Monitoring Enabled [No] : Yes
 Monitoring Port : 17
 Monitor : Ports

 Port     Type        Action    |  Port    Type       Action
 ----   ---------- + --------   |  -----  ---------- + --------
 1      1000T      | Monitor    |  13     1000T      |
 2      1000T      | Monitor    |  14     1000T      |
 3      1000T      | Monitor    |  15     1000T      | Monitor
 4      1000T      | Monitor    |  16     1000T      |
 5      1000T      | Monitor    |  17     1000T      |
 6      1000T      |            |  18     1000T      |
 7      1000T      |            |  19     1000T      |
 8      1000T      |            |  20     1000T      |

Actions->   Cancel     Edit     Save     Help

Select whether to enable traffic monitoring.
Use arrow keys to change field selection, <Space> to toggle field choices,
and <Enter> to go to Actions.
```

Figure 4.3: Menu interface of HP ProCurve 2824 Switch for configuring port mirroring. This example configuration enables port mirroring for RS(5, 3).

Figures 4.4-4.6 illustrate how the time of recovery and the network traffic during recovery are obtained in Wireshark. They show the execution of experiment number 29 (see Table 4.3). The same procedure is followed for all experiments. Figure 4.4 displays packets/sec during the experiment. It is easy to see when the recovery process in HDFS started and ended (when the traffic is much higher than the "normal" traffic in a stable system). The network traffic of interest is captured during the time interval between t0 and t1. The timestamps of the first and last packets of particular sizes which stands out from the "normal" traffic are used to filter out the relevant traffic.

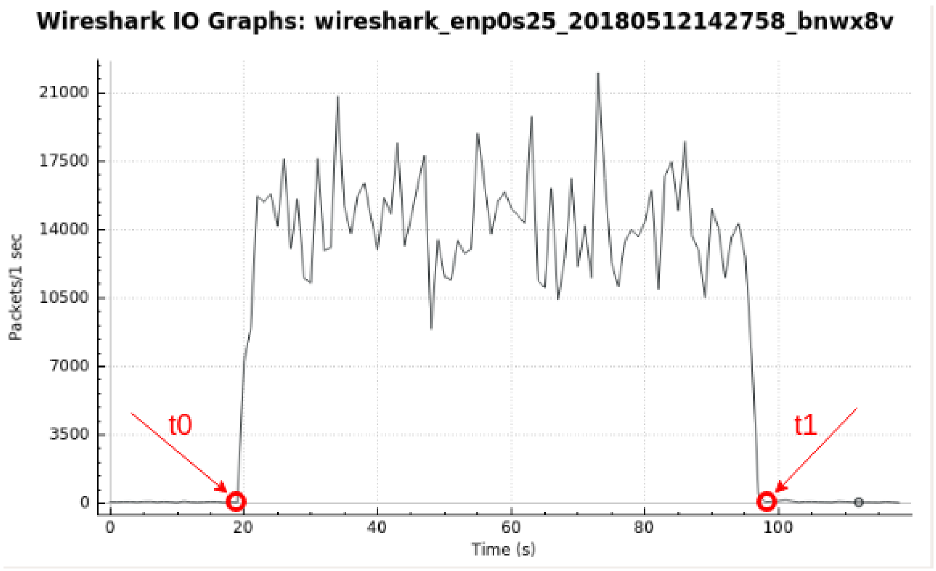
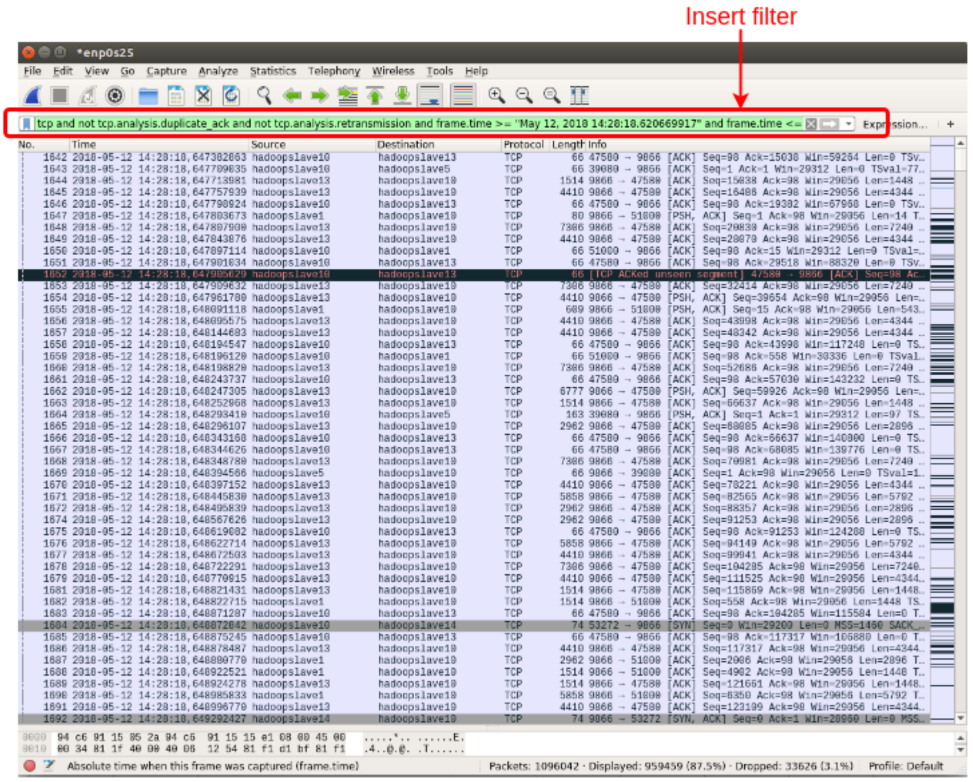


Figure 4.4: Packet capture start time (t0) and end time (t1) for experiment number 29.

Figure 4.5 shows the packet capture display in Wireshark. A display filter (regular expression) is set to only filter out HDFS traffic (network traffic between participating nodes). TCP packets within the time interval are filtered out, and duplicates are excluded since a packet transfer for instance from hadoopslave7 to hadoopslave8 appears twice (as incoming and outgoing packets respectfully). Therefore, one of them has to be canceled out.

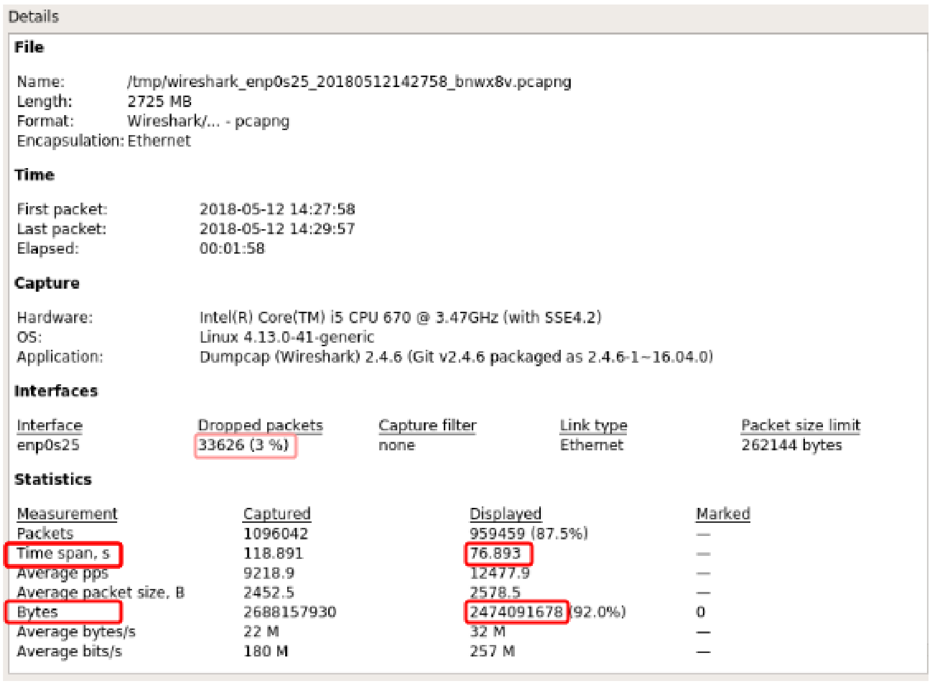

Insert filter
*enp0s25
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Expression...
+

```
tcp and not tcp.analysis.duplicate_ack and not tcp.analysis.retransmission and frame.time >= "May 12, 2018 14:28:18.620669917" and frame.time <=
```

```
No.  Time                           Source          Destination     Protocol Length Info
1642 2018-05-12 14:28:18,647382063 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=15038 Win=59264 Len=0 TSv…
1643 2018-05-12 14:28:18,647709835 hadoopslave10   hadoopslave5    TCP      66 39080 → 9866 [ACK] Seq=1 Ack=1 Win=29312 Len=0 TSval=77…
1644 2018-05-12 14:28:18,647713981 hadoopslave13   hadoopslave10   TCP    1514 9866 → 47580 [ACK] Seq=15038 Ack=98 Win=29056 Len=1448 …
1645 2018-05-12 14:28:18,647757939 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=16486 Ack=98 Win=29056 Len=4344 …
1646 2018-05-12 14:28:18,647790924 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=19382 Win=67968 Len=0 TSv…
1647 2018-05-12 14:28:18,647803673 hadoopslave1    hadoopslave10   TCP      80 9866 → 51000 [PSH, ACK] Seq=1 Ack=98 Win=29056 Len=14 T…
1648 2018-05-12 14:28:18,647807900 hadoopslave13   hadoopslave10   TCP    7306 9866 → 47580 [ACK] Seq=20830 Ack=98 Win=29056 Len=7240 …
1649 2018-05-12 14:28:18,647843876 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=28070 Ack=98 Win=29056 Len=4344 …
1650 2018-05-12 14:28:18,647897114 hadoopslave10   hadoopslave1    TCP      66 51000 → 9866 [ACK] Seq=98 Ack=15 Win=29312 Len=0 TSval=…
1651 2018-05-12 14:28:18,647901034 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=29518 Win=88320 Len=0 TSv…
1652 2018-05-12 14:28:18,647905629 hadoopslave10   hadoopslave13   TCP      66 [TCP ACKed unseen segment] 47580 → 9866 [ACK] Seq=98 Ac…
1653 2018-05-12 14:28:18,647909632 hadoopslave13   hadoopslave10   TCP    7306 9866 → 47580 [ACK] Seq=32414 Ack=98 Win=29056 Len=7240 …
1654 2018-05-12 14:28:18,647961780 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [PSH, ACK] Seq=39654 Ack=98 Win=29056 Len=…
1655 2018-05-12 14:28:18,648091118 hadoopslave1    hadoopslave10   TCP     609 9866 → 51000 [PSH, ACK] Seq=15 Ack=98 Win=29056 Len=543…
1656 2018-05-12 14:28:18,648095575 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=43998 Ack=98 Win=29056 Len=4344 …
1657 2018-05-12 14:28:18,648144683 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=48342 Ack=98 Win=29056 Len=4344 …
1658 2018-05-12 14:28:18,648194547 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=43998 Win=117248 Len=0 TS…
1659 2018-05-12 14:28:18,648196120 hadoopslave10   hadoopslave1    TCP      66 51000 → 9866 [ACK] Seq=98 Ack=558 Win=30336 Len=0 TSval…
1660 2018-05-12 14:28:18,648198820 hadoopslave13   hadoopslave10   TCP    7306 9866 → 47580 [ACK] Seq=52686 Ack=98 Win=29056 Len=7240 …
1661 2018-05-12 14:28:18,648243737 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=57030 Win=143232 Len=0 TS…
1662 2018-05-12 14:28:18,648247305 hadoopslave13   hadoopslave10   TCP    6777 9866 → 47580 [PSH, ACK] Seq=59926 Ack=98 Win=29056 Len=…
1663 2018-05-12 14:28:18,648252968 hadoopslave13   hadoopslave10   TCP    1514 9866 → 47580 [ACK] Seq=66637 Ack=98 Win=29056 Len=1448 …
1664 2018-05-12 14:28:18,648293410 hadoopslave10   hadoopslave5    TCP     163 39080 → 9866 [PSH, ACK] Seq=1 Ack=1 Win=29312 Len=97 TS…
1665 2018-05-12 14:28:18,648296107 hadoopslave13   hadoopslave10   TCP    2962 9866 → 47580 [ACK] Seq=68085 Ack=98 Win=29056 Len=2896 …
1666 2018-05-12 14:28:18,648343168 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=66637 Win=140800 Len=0 TS…
1667 2018-05-12 14:28:18,648344626 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=68085 Win=139776 Len=0 TS…
1668 2018-05-12 14:28:18,648348780 hadoopslave13   hadoopslave10   TCP    7306 9866 → 47580 [ACK] Seq=70981 Ack=98 Win=29056 Len=7240 …
1669 2018-05-12 14:28:18,648394566 hadoopslave5    hadoopslave10   TCP      66 9866 → 39080 [ACK] Seq=1 Ack=98 Win=29056 Len=0 TSval=1…
1670 2018-05-12 14:28:18,648397152 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=78221 Ack=98 Win=29056 Len=4344 …
1671 2018-05-12 14:28:18,648445830 hadoopslave13   hadoopslave10   TCP    5858 9866 → 47580 [ACK] Seq=82565 Ack=98 Win=29056 Len=5792 …
1672 2018-05-12 14:28:18,648495839 hadoopslave13   hadoopslave10   TCP    2962 9866 → 47580 [ACK] Seq=88357 Ack=98 Win=29056 Len=2896 …
1674 2018-05-12 14:28:18,648567626 hadoopslave13   hadoopslave10   TCP    2962 9866 → 47580 [ACK] Seq=91253 Ack=98 Win=29056 Len=2896 …
1675 2018-05-12 14:28:18,648619082 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=91253 Win=124288 Len=0 TS…
1676 2018-05-12 14:28:18,648622714 hadoopslave13   hadoopslave10   TCP    5858 9866 → 47580 [ACK] Seq=94149 Ack=98 Win=29056 Len=5792 …
1677 2018-05-12 14:28:18,648672503 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=99941 Ack=98 Win=29056 Len=4344 …
1678 2018-05-12 14:28:18,648722291 hadoopslave13   hadoopslave10   TCP    7306 9866 → 47580 [ACK] Seq=104285 Ack=98 Win=29056 Len=7240…
1679 2018-05-12 14:28:18,648770915 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=111525 Ack=98 Win=29056 Len=4344…
1681 2018-05-12 14:28:18,648821431 hadoopslave13   hadoopslave10   TCP    1514 9866 → 47580 [ACK] Seq=115869 Ack=98 Win=29056 Len=1448…
1682 2018-05-12 14:28:18,648822715 hadoopslave1    hadoopslave10   TCP    1514 9866 → 51000 [ACK] Seq=558 Ack=98 Win=29056 Len=1448 TS…
1683 2018-05-12 14:28:18,648871287 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=104285 Win=115584 Len=0 T…
1684 2018-05-12 14:28:18,648872842 hadoopslave10   hadoopslave14   TCP      74 53272 → 9866 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_…
1685 2018-05-12 14:28:18,648875245 hadoopslave10   hadoopslave13   TCP      66 47580 → 9866 [ACK] Seq=98 Ack=117317 Win=106880 Len=0 T…
1686 2018-05-12 14:28:18,648878487 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=117317 Ack=98 Win=29056 Len=4344…
1687 2018-05-12 14:28:18,648880770 hadoopslave1    hadoopslave10   TCP    2962 9866 → 51000 [ACK] Seq=2006 Ack=98 Win=29056 Len=2896 T…
1688 2018-05-12 14:28:18,648922521 hadoopslave1    hadoopslave10   TCP    1514 9866 → 51000 [ACK] Seq=4902 Ack=98 Win=29056 Len=1448 T…
1689 2018-05-12 14:28:18,648924278 hadoopslave13   hadoopslave10   TCP    1514 9866 → 47580 [ACK] Seq=121661 Ack=98 Win=29056 Len=1448…
1690 2018-05-12 14:28:18,648985833 hadoopslave1    hadoopslave10   TCP    5858 9866 → 51000 [ACK] Seq=6350 Ack=98 Win=29056 Len=5792 T…
1691 2018-05-12 14:28:18,648996770 hadoopslave13   hadoopslave10   TCP    4410 9866 → 47580 [ACK] Seq=123109 Ack=98 Win=29056 Len=4344…
1692 2018-05-12 14:28:18,649292427 hadoopslave14   hadoopslave10   TCP      74 9866 → 53272 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS…

0000  94 c6 91 15 05 2a 94 c6  91 15 15 e1 08 00 45 00   .....*.. ......E.
0010  00 34 81 1f 40 00 40 06  12 54 81 f1 d1 bf 81 f1   .4..@.@. .T......

Absolute time when this frame was captured (frame.time)    Packets: 1096042 · Displayed: 959459 (87.5%) · Dropped: 33626 (3.1%)    Profile: Default
```

Figure 4.5: Snapshot of the packet capture display in Wireshark with a filter for HDFS traffic (experiment number 29).

When the filter is properly set, Wireshark provides a summary of the total capture as it is shown in Figure 4.6. The "Displayed" column refers to the filtered traffic. Along with other measurements, the time span in seconds and number of bytes are displayed, which are the measurements of interest in this thesis. It also shows the percentage of non-captured packets of the total capture. The issue of non-captured packets related to port mirroring is explained more thoroughly in Section 4.2.3.



```
Details
File

Name:          /tmp/wireshark_enp0s25_20180512142758_bnwx8v.pcapng
Length:        2725 MB
Format:        Wireshark/... - pcapng
Encapsulation: Ethernet

Time

First packet:          2018-05-12 14:27:58
Last packet:           2018-05-12 14:29:57
Elapsed:               00:01:58

Capture

Hardware:              Intel(R) Core(TM) i5 CPU 670 @ 3.47GHz (with SSE4.2)
OS:                    Linux 4.13.0-41-generic
Application:           Dumpcap (Wireshark) 2.4.6 (Git v2.4.6 packaged as 2.4.6-1~16.04.0)

Interfaces

Interface              Dropped packets    Capture filter    Link type    Packet size limit
enp0s25                33626 (3 %)        none              Ethernet     262144 bytes

Statistics

Measurement               Captured       Displayed              Marked
Packets                   1096042        959459 (87.5%)         —
Time span, s              118.891        76.893                 —
Average pps               9218.9         12477.9                —
Average packet size, B    2452.5         2578.5                 —
Bytes                     2688157930     2474091678 (92.0%)     0
Average bytes/s           22 M           32 M                   —
Average bits/s            180 M          257 M                  —
```

Figure 4.6: Summary with details from Wireshark capture. It displays various information, the most relevant in this case is the time span and number of bytes which are highlighted in red (experiment number 29).

4.2 Result and Analysis

This section presents the results of the experiments given in Table 4.3. First, the file recovery time for different scenarios is compared and then for the same scenarios, the network traffic during recovery is compared. The results are mainly presented through graphs because they illustrate possible differences well. Finally, an analysis and discussion of the results are presented. If exact measurements are of interest, full results in tabular form are attached in Appendix B.

4.2.1 Comparison of Time of Recovery

Figure 4.7 compares the time of recovery for 2.5 GB files. The block sizes do not significantly affect the time in general, but RS(14, 10) with 256 MB block size stands out. The time is around 1/3 higher than the time for 64 MB and 128 MB block sizes, even when several runs of the experiment is performed. 3-rep had the best performance, but it is relatively close to RS(5, 3). RS(9, 6) had the longest recovery duration. Also, note that 128 MB block size (yellow line) gives a slightly better performance for all storage policies. Figures 4.8 and 4.9 compare the time of recovery for 5 GB and 10 GB files respectively. They show higher recovery duration compared to experiments with 2.5 GB files, but they show almost the same results as the 2.5 GB file experiments in terms of performance from best to worst: 3-rep - RS(5, 3) - RS(14, 10) - RS(9, 6). They also show that 128 MB block size gives a slightly shorter recovery duration.

Figure 4.10 presents the comparison of all the file sizes in one line diagram, where the round bullets represent 2.5 GB files, the square bullets represent 5 GB files and the triangle bullets represent 10 GB files. This diagram is included to show that the time of recovery increases as the file size increases. For each storage policy, the time increases almost linearly with the file size. That makes sense because the file size is always the double (2.5, 5 and 10). For instance, in case of RS(5, 3) the time for 2.5 GB files is around 40 seconds, the time for 5 GB files is around 80 seconds and the time for 10 GB files is around 160 seconds. Likewise, for the other storage policies, this "linear" increase in recovery time is displayed.

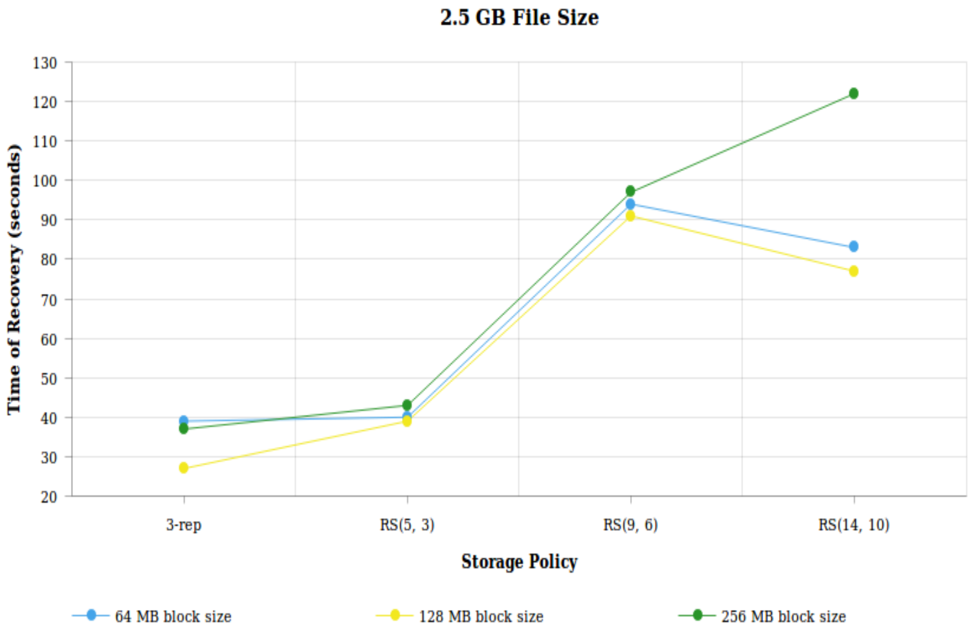


Figure 4.7: Comparing time of recovery for 2.5 GB files for different policies and block sizes.

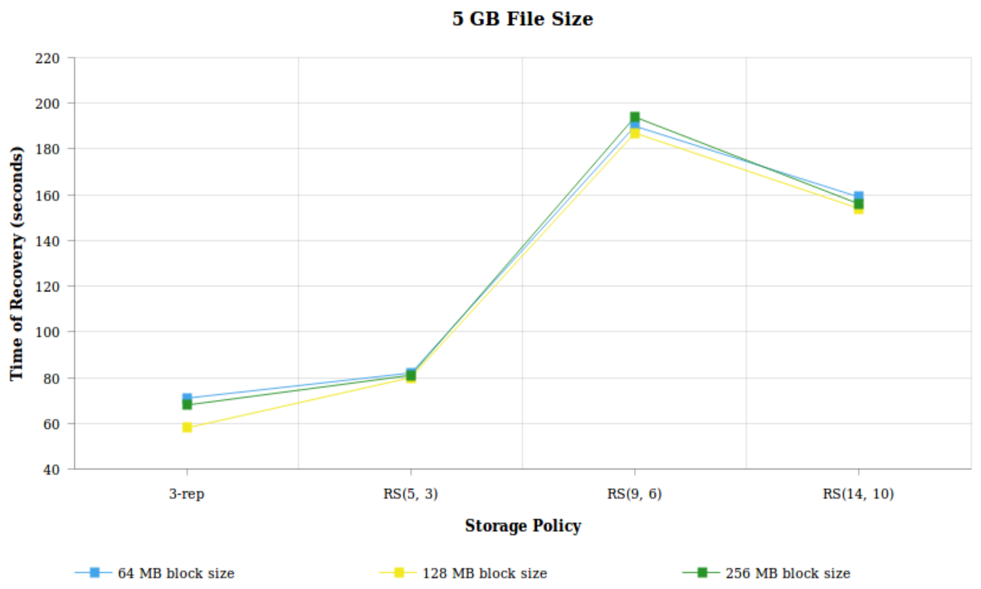


Figure 4.8: Comparing time of recovery for 5 GB files for different policies and block sizes.

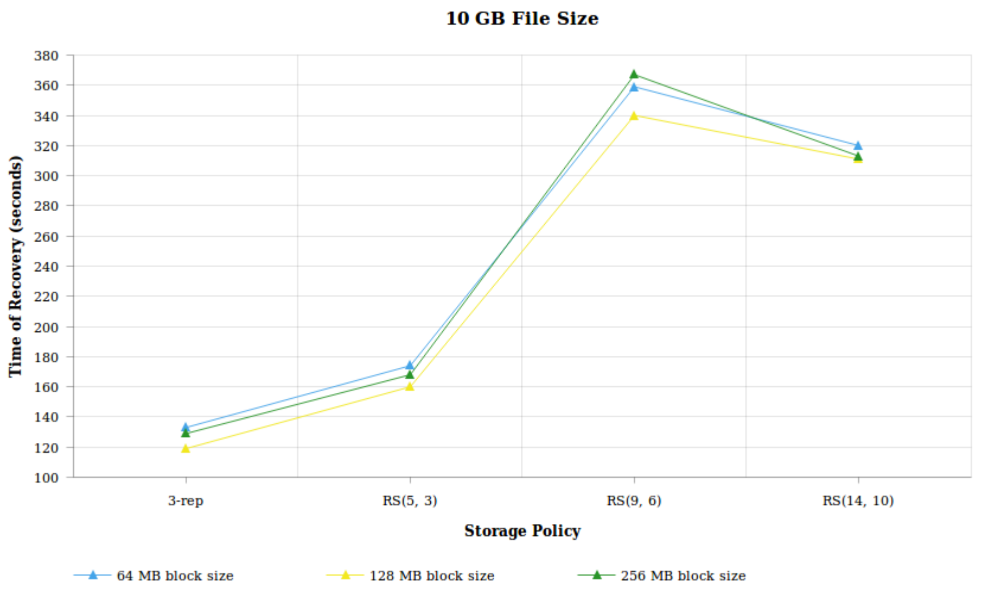


Figure 4.9: Comparing time of recovery for 10 GB files for different policies and block sizes.

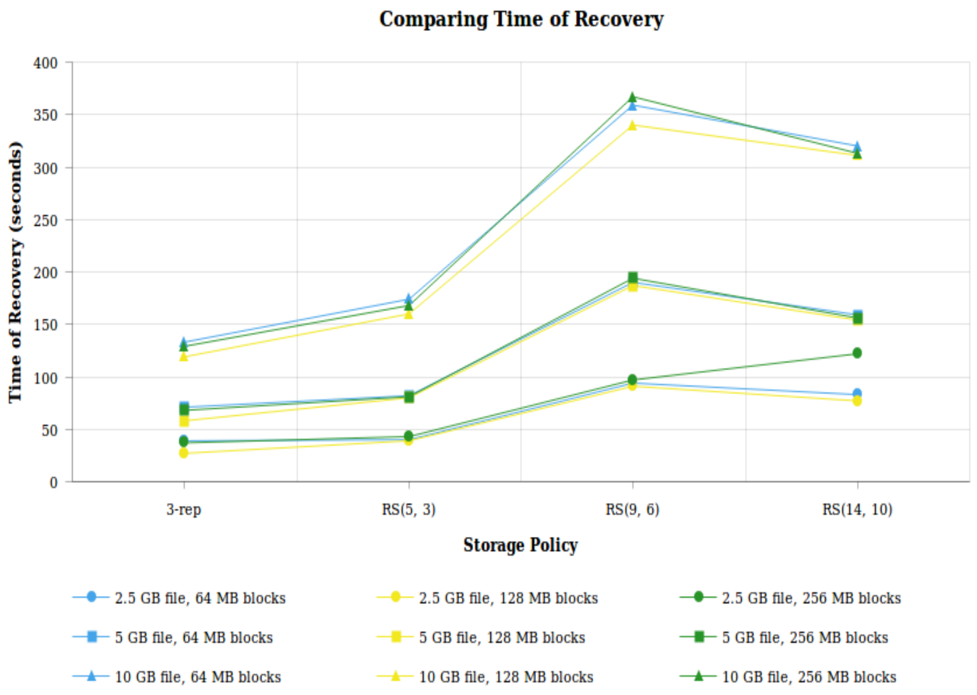


Figure 4.10: Comparison of time of recovery for all file and block sizes.

4.2.2 Comparison of Network Traffic During Recovery

This subsection presents a comparison of the network traffic during recovery for all the experiments (see Figure 4.11). The traffic for different block sizes with the same file size is not displayed separately because the differences are so minimal. Figure 4.11 shows that the traffic in all experiments is almost equivalent to the size of the file that participated in the experiments. The bullets in the diagram represent the same metrics as explained for Figure 4.10. There is almost 2.5 GB network traffic for experiments including 2.5 GB files, and correspondingly almost 5 GB and 10 GB network traffic for experiments with 5 GB and 10 GB files, respectively. Generally, there are not many differences in the traffic for the different storage policies and block sizes. The most important thing to point out is that the network traffic increases with the file size. This applies to all storage policies and for all block sizes. Another thing to mention is that 3-rep and RS(5, 3) show slightly more varying traffic compared to RS(9, 6) and RS(14, 10). That is because more packets are not captured in Wireshark. There are more packets/second in the 3-rep and RS(5, 3) experiments which caused more packets not to be captured. An important thing to notice is that these packets are not dropped, instead they are not captured because the packet capturing method is not always able to capture all packets on the network.

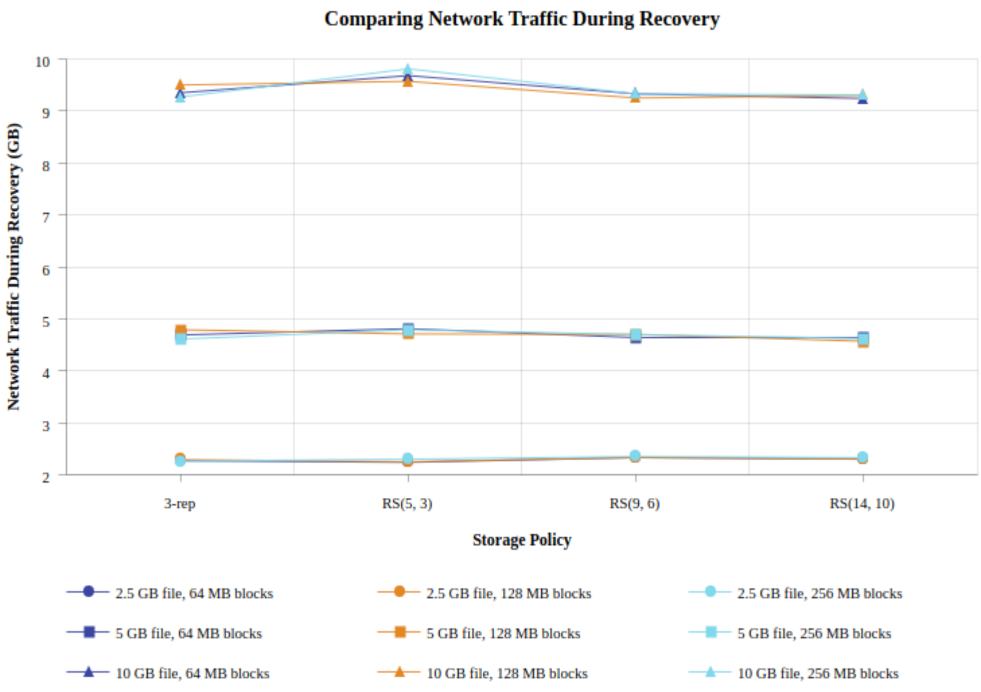


Figure 4.11: Comparison of network traffic during recovery for all file and block sizes.

4.2.3 Discussion

This section provides a more complete discussion of the test results with explanations that are related to the theory presented in Chapter 2. In addition, some limitations of the used method (port mirroring), the software tool (Wireshark) and hardware components are discussed.

Utilizing HDFS storage with 128 MB block size shows slightly better performance in terms of recovery duration compared to 64 MB and 256 MB block sizes. This corresponds to Apache Hadoop's choice of having default block size in HDFS set to 128 MB, which is a good finding. Further, one experiment stood out from the others. A 2.5 GB file stored with RS(14, 10) and 256 MB block size results in a notably longer recovery duration. Since the block size is rather big, there is only one block stored on each of the 14 nodes. A possible reason for the long recovery duration is when such a small file as 2.5 GB (in a big data context) is split into rather big chunks (256 MB), HDFS has trouble keeping the time down when the participating nodes only hold one block each. Despite that, the block sizes show in general minimal impact on the recovery duration in the specific test environment set up for the thesis.

The recovery duration increases with the code length. This is supported by the theory that erasure coding takes more time to recreate lost data compared to replication (see Section 2.3). One of the reasons is because more disk I/O's are required in order to recover a file when one of its storage nodes fails. Namely, data from k nodes is accessed and transferred with an *(n, k)* code compared to replication. Moreover, an interesting finding is that RS(9, 6) shows longer recovery duration than RS(14, 10). This somehow conflicts with the theory presented initially. There can be many reasons related to the concrete setup such as specific network speeds, buffer speeds of the network cards and speeds of the network devices. All these parameters can affect the recovery duration (time of recovery). Engineers are normally required to fine-tune all of these parameters to specifically fit scenarios of interest.

In case of network traffic during recovery, the results completely confirm theory. That is a great match. The traffic is almost equivalent to the file sizes, independent of storage policy and block size. Note that because of non-captured packets the traffic is always a little less than the size of the files. Overall, the experiments had an average of 3.9% non-captured packets. The network traffic results reflect theory because in case of 3-rep the whole file of a particular size is recovered, e.g. 5 GB traffic is transferred to recover files in experiments including 5 GB files. In case of an encoding of a file of size 5 GB with an RS(*n, k*) code, the file is first split into k segments and stored in k nodes where each node holds $5/k$ GB of the file. Also, r (n-k=r) redundancy nodes are added. In order to recover one node ($5/k$ GB data), data from k nodes is accessed and transferred. Therefore, the total amount of repair traffic is k*$(5/k)$ GB = 5 GB. This applies to all RS codes explored in the thesis. Recall the theory presented in Section 2.3.

The chosen method for capturing network traffic is port mirroring. That has some limitations. Not capturing all packets can happen when using this method, especially when the traffic is really high, due to overload on the mirroring wire. Testing also 20 GB files was the original plan, but the method causes too many non-captured packets during the recovery process, so Wireshark does not display precise enough results. Also, robust managed network switches with the possibility to enable port mirroring are quite expensive. Because of these limitations, other methods for capturing network traffic can be more applicable especially when testing bigger files. Methods like machine-in-the-middle, man-in-the-middle (software) or utilizing a network TAP can be explored for this kind of usage. They can all be used in correlation with Wireshark. Also, other network analysis tools like Nload, iftop or iptraf can be used to replace Wireshark. They require all the HDFS nodes to run the tool during recovery and a script to collect all the traffic logs and to filter out the relevant information.

Finally, some comments about the actual storage/encoding of files in HDFS follow. There are not exact measurements collected for this, but after monitoring all 36 experiments individually there are remarkable differences in the time of encoding for the different storage policies. The encoding duration for RS codes is much shorter than the storage process for 3-rep. The RS(14, 10) encoding processes are the quickest. This supports theory because the storage overhead is bigger for 3-rep, there are more actual data to store (200% overhead). As the RS code lengths get bigger, the storage overhead gets smaller (see Table 4.1) and so does the encoding duration.

Chapter 5

Conclusion and Further Work

5.1 Conclusion

The process of setting up a multi-node Hadoop cluster is proven to be convenient through this work. Once a single-node cluster is configured it is rather straightforward to add as many DataNodes as preferred following the same procedure. HDFS requires specific terminal commands in order to manage and interact with the file system. They differ from the Unix file system commands, but they are perceived as user-friendly. Erasure coding is easy to enable and HDFS automatically encodes files correctly and reliably when enough cluster nodes are provided.

We can conclude that both the network traffic and duration of a file recovery process increase as the file size increases. Additionally, based on the results of the experiments conducted in the testing environment set up for the thesis, the following conclusions can be drawn. Triple-replication is the best choice when the objective is getting the shortest possible time of recovery. If a little more time for file recovery is tolerated, then RS(5, 3) is a good choice. Less storage overhead is needed, while still the same level of reliability is offered (triple-replication and RS(5, 3) have both a fault tolerance of 2). RS(14, 10) is probably the overall best storage policy of choice. It has a shorter recovery duration than RS(9, 6) and offers a fault tolerance of 4 node failures. It provides the least storage overhead (40%) and offers a high level of reliability. Also, choosing the default 128 MB block size in HDFS is a clever choice because it has been proven through this work that it always scored the highest on performance in terms of time of recovery.

Conventional MDS erasure codes that are not optimized for recovery, such as Reed-Solomon codes, transfer the same amount of traffic as the file size. This comes from the k out of n property. The results demonstrate that the amount of traffic during recovery is the same even for different RS parameters. The only difference is the amount of data that is recovered. That depends on the code parameters, i.e. k. HDFS is normally used to store big data. Therefore, using pure RS codes in HDFS may not be the optimal choice because it would take a really long time to recreate lost data and require tremendous amounts of network traffic. In other works it is proven that optimized erasure codes can provide better performance in terms of e.g. network traffic and recovery time (see Section 2.4). So, optimized erasure coding variants are probably the most optimal storage policies to select in terms of both performance and reliability.

5.2 Further Work

It would be a natural extension of this work to test other RS codes, such as RS(16, 12), RS(20, 18), RS(20, 16), or even the popular RS(255, 223). That will give a broader view of which codes are most suitable for various file and block sizes. For some of those codes there will be a need for a lot of extra nodes, but all those situations are addressable in practice.

Also, testing much bigger file sizes (such as hundreds of gigabytes or even tens of terabytes) will give more realistic and practical test results because HDFS is normally used to store big data sets in a production environment. Additionally, comparing other measurements such as time of encoding and decoding can be interesting [Vel18]. Another direction is to study the performance when the maximum tolerated number of nodes have failed for various RS codes.

Testing other types of erasure codes in Hadoop is also necessary, such as Regenerating Codes (RGCs) and Locally Repairable Codes (LRCs) [RSG+15, GKJS17, KGØ16a, KGØ16b]. Their designs differ from RS codes, and they have been studied and used in practical systems, but they are not given yet in any official version of Hadoop. RGCs minimize the repair traffic, while LRCs minimize the number of contacted nodes during a node repair.

This work utilizes a one-rack Hadoop cluster. Placing nodes on different racks will support rack awareness which will give the opportunity to test for instance network switch failures, and not only failures on node level [Fouc]. Also, in the present work physical nodes are used to build the Hadoop cluster. Testing and comparing the performance of erasure coding and replication on virtual servers vs. physical servers can be an interesting point of view because virtualization and cloud computing are more and more used.

References

[BCKØ16] Gergely Biczók, Yanling Chen, Katina Kralevska, and Harald Øverby. Combining forward error correction and network coding in bufferless networks: A case study for optical packet switching. In *2016 IEEE 17th International Conference on High Performance Switching and Routing (HPSR)*, pages 61–68, June 2016.

[Cor] International Data Corporation. Amount of data created annually to reach 180 zettabytes in 2025. https://whatsthebigdata.com/2016/03/07/amount-of-data-created-annually-to-reach-180-zettabytes-in-2025/. Accessed: 2018-03-24.

[DGW+10] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE transactions on information theory*, 56(9):4539–4551, 2010.

[Fla] Data Flair. Data block in hdfs. https://data-flair.training/blogs/data-block/. Accessed: 2018-04-20.

[FLP+10] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Osdi*, volume 10, pages 1–7, 2010.

[Foua] The Apache Software Foundation. Apache hadoop 3.0.0. http://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-common/SingleCluster.html. Accessed: 2018-02-15.

[Foub] The Apache Software Foundation. HDFS architecture. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. Accessed: 2018-03-24.

[Fouc] The Apache Software Foundation. HDFS erasure coding. https://hadoop.apache.org/docs/r3.0.0-alpha1/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html. Accessed: 2018-03-24.

[GHSY12] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 58(11):6925–6934, 2012.

[GKJS17] D. Gligoroski, K. Kralevska, R. E. Jensen, and P. Simonsen. Repair duality with locally repairable and locally regenerating codes. In *IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress*, pages 979–984, Nov 2017.

[HSX+12] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure coding in windows azure storage. In *Usenix annual technical conference*, pages 15–26. Boston, MA, 2012.

[IBM] IBM. Big data analytics. https://www.ibm.com/analytics/hadoop/big-data-analytics. Accessed: 2018-03-24.

[KBP+12] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *FAST*, page 20, 2012.

[KGJØ17] Katina Kralevska, Danilo Gligoroski, Rune E Jensen, and Harald Øverby. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data*, pages 1–1, 2017.

[KGØ16a] Katina Kralevska, Danilo Gligoroski, and Harald Øverby. Balanced locally repairable codes. In *2016 9th International Symposium on Turbo Codes and Iterative Information Processing (ISTC)*, pages 280–284, Sept 2016.

[KGØ16b] Katina Kralevska, Danilo Gligoroski, and Harald Øverby. General sub-packetized access-optimal regenerating codes. *IEEE Communications Letters*, 20(7):1281–1284, July 2016.

[KOG15] Katina Kralevska, Harald Overby, and Danilo Gligoroski. Coded packet transport for optical packet/burst switched networks. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2015.

[Kra16] Katina Kralevska. Applied erasure coding in networks and distributed storage. *Ph.D. thesis*, December 2016.

[LC82] Shu Lin and Daniel Costello. *Error Control Coding: Fundamentals and Applications*, pages 29–39. Pearson, 1982.

[Mat] Wolfram MathWorld. Finite field. http://mathworld.wolfram.com/FiniteField.html. Accessed: 2018-04-02.

[NBP+13] Jyoti Nandimath, Ekata Banerjee, Ankur Patil, Pratima Kakade, Saumitra Vaidya, and Divyansh Chaturvedi. Big data analysis using apache hadoop. In *Information Reuse and Integration (IRI), 2013 IEEE 14th International Conference on*, pages 700–703. IEEE, 2013.

[OD11] Frederique Oggier and Anwitaman Datta. Self-repairing homomorphic codes for distributed storage systems. In *INFOCOM, 2011 Proceedings IEEE*, pages 1215–1223. IEEE, 2011.

[Pap14] Dimitrios Papailiopoulos. *Distributed large-scale data storage and processing.* PhD thesis, The University of Texas at Austin, 2014.

[PBN12] Aditya B Patel, Manashvi Birla, and Ushma Nair. Addressing big data problem using hadoop and map reduce. In *Engineering (NUiCONE), 2012 Nirma University International Conference on*, pages 1–5. IEEE, 2012.

[PH13] James S Plank and C Huang. Tutorial: Erasure coding for storage applications, part 2. In *Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies http://web.eecs.utk.edu/ plank/plank/papers/FAST-2013-Tutorial.html*, 2013.

[PJHO13] Lluis Pamies-Juarez, Henk DL Hollmann, and Frédérique Oggier. Locally repairable codes with multiple repair alternatives. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 892–896. IEEE, 2013.

[RSG$^+$15] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM SIGCOMM Computer Communication Review*, 44(4):331–342, 2015.

[SAP$^+$13] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. *Proceedings of the VLDB Endowment*, 6(5):325–336, 2013.

[Shi12] Kyuseok Shim. Mapreduce algorithms for big data analysis. *Proceedings of the VLDB Endowment*, 5(12):2016–2017, 2012.

[Spa] Cyber Security Space. Big data series(post 1): What is big data? what is hadoop? http://www.cybersecurityspace.com/2016/08/big-data-seriespost-1-what-is-big-data.html. Accessed: 2018-05-29.

[Vel18] Maximiliano Matias Vela. A comparative study on distributed storage and erasure coding techniques using apache hadoop over nornet core. Master's thesis, The University of Bergen, 2018.

[VRP$^+$18] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. Clay codes: moulding mds codes to yield an msr code. In *16th USENIX Conference on File and Storage Technologies*, page 139, 2018.

[Whi12] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[Wir] Wireshark. About wireshark. https://www.wireshark.org/. Accessed: 2018-05-14.

[ZE$^+$11] Paul Zikopoulos, Chris Eaton, et al. *Understanding big data: Analytics for enterprise class hadoop and streaming data.* McGraw-Hill Osborne Media, 2011.

Appendix A

Upgrading Hadoop Single-node Cluster

This appendix presents an easy guide on how to upgrade the Hadoop version on a single-node Hadoop cluster.

1. Stop the Hadoop cluster (`stop-all.sh`).
2. Run the commands `sudo apt-get update` and `sudo apt-get upgrade`.
3. Check the java version (`java -version`). Should be Java 8.
4. Download the preferable `hadoop-X.Y.Z-src.tar.gz` file from the Apache Hadoop mirror site (http://apache.uib.no/hadoop/common/) and extract the Hadoop folder in a location of your choice.
5. Navigate to the `/usr/local/hadoop/etc/hadoop` directory and copy five files (listed below) to the recently extracted Hadoop folder. Here you replace the copied files with the existing ones.

   – hadoop-env.sh

   – core-site.xml

   – hdfs-site.xml

   – yarn-site.xml

   – mapred-site.xml

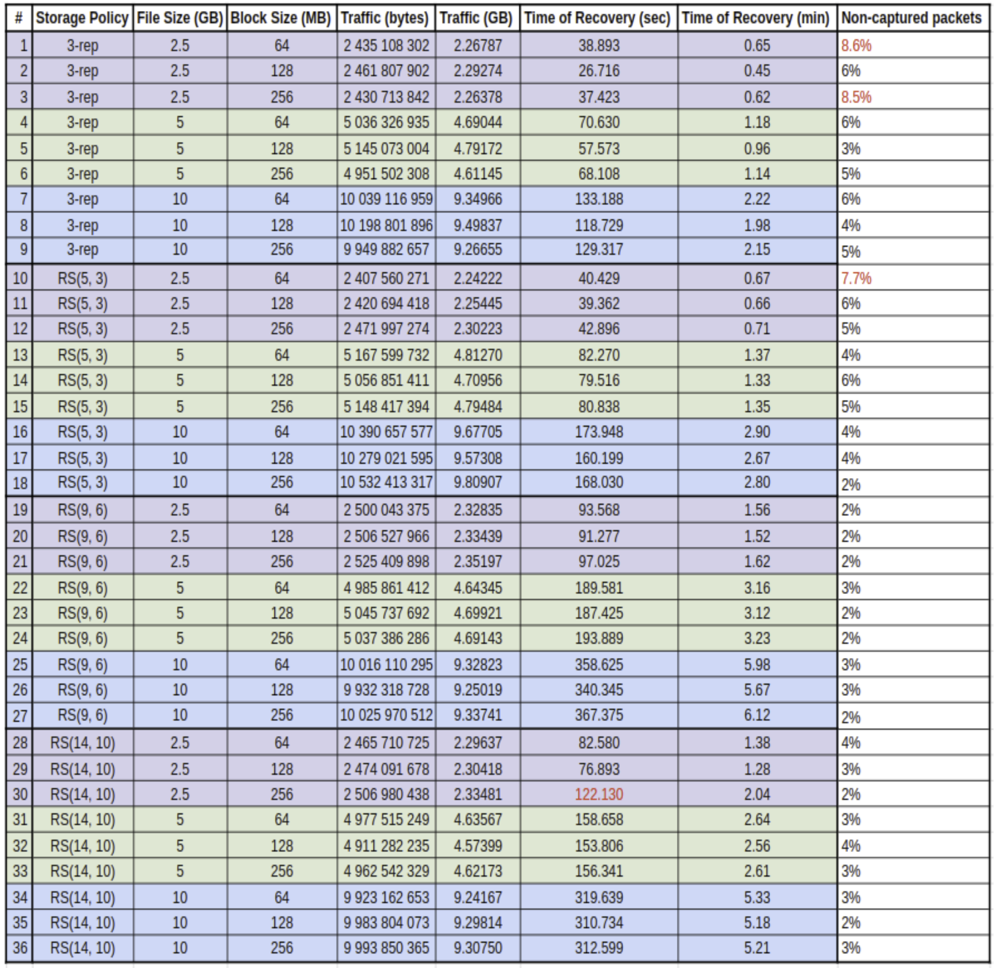
6. Run the commands shown in Figure A.1.

```
# Uninstall and remove old version
hduser@hadoopServer:~$ sudo rm -r -f /usr/local/hadoop

# Move the extracted Hadoop folder to the Hadoop directory
hduser@hadoopServer:~$ sudo mv /*path to the extracted Hadoop folder*
/usr/local/hadoop

# Assign ownership of the Hadoop directory to hduser
hduser@hadoopServer:~$ sudo chown hduser:hadoop -R /usr/local/hadoop

# Upgrade DFS from the Hadoop directory
hduser@hadoopServer: /usr/local/hadoop$ start-dfs.sh -upgrade

# Upgrade YARN
hduser@hadoopServer: /usr/local/hadoop$ start-yarn.sh -upgrade

# If the components run properly, finalize the HDFS upgrade
hduser@hadoopServer:      /usr/local/hadoop/bin$    hdfs    dfsadmin    -
finalizeUpgrade

# Check the new Hadoop version
hduser@hadoopServer:~$ hadoop version
```

Figure A.1: Ubuntu terminal commands: Upgrading single-node Hadoop cluster.

Appendix B

Full Results from Experiments



#	Storage Policy	File Size (GB)	Block Size (MB)	Traffic (bytes)	Traffic (GB)	Time of Recovery (sec)	Time of Recovery (min)	Non-captured packets
1	3-rep	2.5	64	2 435 108 302	2.26787	38.893	0.65	8.6%
2	3-rep	2.5	128	2 461 807 902	2.29274	26.716	0.45	6%
3	3-rep	2.5	256	2 430 713 842	2.26378	37.423	0.62	8.5%
4	3-rep	5	64	5 036 326 935	4.69044	70.630	1.18	6%
5	3-rep	5	128	5 145 073 004	4.79172	57.573	0.96	3%
6	3-rep	5	256	4 951 502 308	4.61145	68.108	1.14	5%
7	3-rep	10	64	10 039 116 959	9.34966	133.188	2.22	6%
8	3-rep	10	128	10 198 801 896	9.49837	118.729	1.98	4%
9	3-rep	10	256	9 949 882 657	9.26655	129.317	2.15	5%
10	RS(5, 3)	2.5	64	2 407 560 271	2.24222	40.429	0.67	7.7%
11	RS(5, 3)	2.5	128	2 420 694 418	2.25445	39.362	0.66	6%
12	RS(5, 3)	2.5	256	2 471 997 274	2.30223	42.896	0.71	5%
13	RS(5, 3)	5	64	5 167 599 732	4.81270	82.270	1.37	4%
14	RS(5, 3)	5	128	5 056 851 411	4.70956	79.516	1.33	6%
15	RS(5, 3)	5	256	5 148 417 394	4.79484	80.838	1.35	5%
16	RS(5, 3)	10	64	10 390 657 577	9.67705	173.948	2.90	4%
17	RS(5, 3)	10	128	10 279 021 595	9.57308	160.199	2.67	4%
18	RS(5, 3)	10	256	10 532 413 317	9.80907	168.030	2.80	2%
19	RS(9, 6)	2.5	64	2 500 043 375	2.32835	93.568	1.56	2%
20	RS(9, 6)	2.5	128	2 506 527 966	2.33439	91.277	1.52	2%
21	RS(9, 6)	2.5	256	2 525 409 898	2.35197	97.025	1.62	2%
22	RS(9, 6)	5	64	4 985 861 412	4.64345	189.581	3.16	3%
23	RS(9, 6)	5	128	5 045 737 692	4.69921	187.425	3.12	2%
24	RS(9, 6)	5	256	5 037 386 286	4.69143	193.889	3.23	2%
25	RS(9, 6)	10	64	10 016 110 295	9.32823	358.625	5.98	3%
26	RS(9, 6)	10	128	9 932 318 728	9.25019	340.345	5.67	3%
27	RS(9, 6)	10	256	10 025 970 512	9.33741	367.375	6.12	2%
28	RS(14, 10)	2.5	64	2 465 710 725	2.29637	82.580	1.38	4%
29	RS(14, 10)	2.5	128	2 474 091 678	2.30418	76.893	1.28	3%
30	RS(14, 10)	2.5	256	2 506 980 438	2.33481	122.130	2.04	2%
31	RS(14, 10)	5	64	4 977 515 249	4.63567	158.658	2.64	3%
32	RS(14, 10)	5	128	4 911 282 235	4.57399	153.806	2.56	4%
33	RS(14, 10)	5	256	4 962 542 329	4.62173	156.341	2.61	3%
34	RS(14, 10)	10	64	9 923 162 653	9.24167	319.639	5.33	3%
35	RS(14, 10)	10	128	9 983 804 073	9.29814	310.734	5.18	2%
36	RS(14, 10)	10	256	9 993 850 365	9.30750	312.599	5.21	3%