Imperial College London
Department of Computing

# Study of the Feasibility of a Distributed Storage System based on a Minimum Bandwidth Exact Regenerating Code

*by:*

Eric Arnoult

Submitted in partial fulfilment of the requirements for the MSc Degree in Advanced Computing of Imperial College London

September 2011

## Abstract

This is the final report for my summer project at the Imperial College. We look at the use of erasure correcting codes in distributed storage. Here we more particularly study Minimum Bandwidth Exact Regenerating codes that allow to minimize the bandwidth required to repair the system following the crash of a server (also called node failure).

This document is intended to put on paper the different things that have been done during this project including the study of the Minimum Bandwidth Exact Regenerating codes for Distributed Storages from [20] and the piece of software used to test the performance of the system but also how the use of the twin codes from [19] can be used in combination with it to improve the system.

**Acknowledgements**

First and foremost, I would like to thank my project supervisor, Dr. Naranker Dulay, for his advice and his constant encouragement. I am deeply indebted to him for his invaluable guidance throughout the project.

I thank my second marker, Dr. Emil Lupu, for taking the time of reading this report and for his constructive question during the presentation.

I would like to thank Thom Haddow for being ready to help me and discuss ideas regarding the use of PlanetLab. At this effect, I would also like to thank PlanetLab Mailing List.

I thank Duncan White, David McBride and all the members from the Computing Support Group for their help in the realisation of my experiments and for the time they accorded me during this project.

Finally, I give my special thanks to Dr. Matthieu Finiasz. Its wide knowledge in the domain of Erasure Codes helped my understanding of complex ideas such as Finite Field or Information Coding theory that would have been much more difficult to apprehend without him.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With the emergence of more and more web applications such as social networks or even the widespread use of e-mail, photos, or videos, comes the need for more and more space. Some systems and companies do now have to store and use a huge amount of data such as media files. Imagine that common phone can now store several gigabytes of information. A single computer can have terabytes of media files so let's imagine the gigantic amount of data that have to deal with companies such as Google, Amazon or Ebay. They can't store all their data on only one disk so they have to distribute everything on several disks and servers. When a common user can have to manage several devices, these important companies surely have thousands and thousands of them.

As we can't store everything on one large disk any more, we need to distribute the storage on different medias. But the problem is that in modern data centres the storage nodes are individually unreliable and that is why we add redundancy to improve reliability against node failures. The problem is that all hardware components have a limited lifetime and that for example a hard drive can live from one year to 20 years. Even if their lifetime was infinite there is always the problem of having an accident. Imagine that if a company like Facebook loses only one disk or node (without any protection against node failure), then several user profiles will be definitely lost like the mails were lost from gmail in 2011 as we can see in the article [5]. This kind of problem justifies the need for redundancy since if part of a document exists at several places, if one place fails, we can still reconstruct the document thanks to the other.

Now let's imagine a more particular but still very used utilisation: an email server. In this particular case, there are a lot of little files so we do not care too much if we have to add a lot of redundancy (since the redundant file will be quite small as well) but we do care about is the fact that if a node fails we can regenerate him very quickly because the users still want to access their emails. That is exactly the goal of this project: provide to the clients several nodes where they can redundantly store their files and where the nodes that failed are very quickly repaired. This algorithm is part of the erasure coding theory on which we will give an introduction in chapter 1.

We still need to know how we can achieve such a thing. We are going to use an algorithm named Minimum Bandwidth Exact Regenerating code that we can found in [20]. This term means that the code is able to rebuild file thanks to less than the maximum of available nodes, but can also auto repair nodes that fail in the exact state it was before the failing (we will come back on this later in chapter 3). And it can do all this in minimizing the bandwidth which is what we need since we want the regeneration to be very quick.

The problem with this kind of algorithm is that it has stayed a very theoretical area of research and has never been implemented before and so we did not know if it was feasible. The goal of this project is to compare some experiments with the theory that we will explain in chapter 3. The structure and the architecture of the piece of software used to realise these experiments will be explained in chapter 4.

Chapter 5 will be the part where we will show the feasibility of such a system through several experiments. These experiments will also be the chance for us to see the impact of some hardware components and some parameters on the system.

We will for example see that the fragmentation has a huge role to play in the performance of the system, but also that the cache that is a whole layer of the implementation detailed in chapter 4 can double the performance. Impact of the bandwidth and of the CPU frequency will also be taken into account.

Finally the last chapter will introduce a very recent paper [19] where the authors described a new system called Twin Codes based on the use of several codes to get better performance. This chapter will also be for us the occasion to propose a new approach for distributed storage systems that will be presented as a draft of the combination of the Twin Codes and the Minimum Bandwidth Exact Regenerating code.

# Chapter 2

# Mathematical Foundations

Before we can talk about the real algorithm we need to introduce some mathematical facts and that is why this chapter is here: giving a better understanding of the underlying mathematical points that are needed for this project. As told in the introduction, the main element on which the whole system is based on is the erasure coding. This is why this chapter will give an overview of the erasure coding theory. To do so, we first need to give an explanation of what are finite fields [10] (that are used in the construction of a code). We will then see how works erasure coding theory and finally have a look at a particular code : Reed-Solomon code [21] which is the one used in our system.

## 2.1 Finite Field

Most of the following comes from [22], where the author gives a very detailed description of the finite field. They are used in a wide range of area such as number theory, Galois theory or cryptography but the one that interests us is coding theory and we will see how in the next section. This section is here to give the most useful properties of this mathematical tool which leads to the explicit construction of a finite field.

Finite fields are fields that contain a finite number of elements. A finite field is completely defined by its number of elements which is always a power of a prime number which means that it exits a finite field for every $q^m$ with $q$ a prime number.

The tool that allows the construction of any finite field is the modular arithmetic for the prime finite fields (the finite fields for which the power is 0) and the polynomial modular arithmetic (which is the euclidean division for the polynomials) for the general case.

In order to understand what happens next, we first need a couple of definitions:

### 2.1.1 Definitions

The number of elements of a field is called the finite field order.

As a finite field is a multiplicative group, every finite field has a primitive element [10]: an element which generates every other element. This particular element will be referred as $\alpha$ in the rest of this document.

Irreducible polynomials may be the most important of the finite field creation process: a polynomial $P$ of degree $m$ is irreducible over $\mathbb{F}_q$ if and only if $P$ divide $X^{q^m} - X$ and $P$ does not divide any of the $X^{q^r} - 1$ with $r = m/d$, where the $d$s are the prime numbers that divide $m$ [22].

A subclass of the irreducible polynomials are the primitive polynomials : they are the polynomials that have for roots every conjugate of the primitive element which means: $\alpha, \alpha^q, \alpha^{p^q}, ..., \alpha^{q^{m-1}}$. Primitive polynomials can be computed thanks to Berlekamp algorithm which can be found in [3] but are more generally found in precomputed tables from the literature.

In order to construct the finite field with $q^m$ elements $\mathbb{F}_{q^m}$, we first need to select a monic irreducible polynomial $f(X)$ of degree $m$ in $\mathbb{F}_q[X]$ (in fact we take the primitive polynomial of any primitive element of the field $\mathbb{F}_{q^m}$). Then, we compute the power of $\alpha$ thanks to this polynomial: $\mathbb{F}_q^m = \mathbb{F}_q[\alpha] modP$(which is also equals to $\alpha, \alpha^q, \alpha^{q^2}, ..., \alpha^{q^{m-1}}$) [22].

Reminding that every element is represented by a polynomial, operations over finite field become very easy: the addition and the multiplication of two elements are the addition and multiplication of two polynomials modulo the primitive polynomial.

### 2.1.2 Toy Example

Let's say we want to construct the finite field of size 16 which is $\mathbb{F}_{2^4}$. We use the polynomial $P = X^4 + X^3 + 1$ which is a primitive polynomial found in the literature, we then use the primitive polynomials to compute every power of $\alpha$.

The computation of the first 3 powers of $\alpha$ are easy since $\alpha^3 = \alpha^3 modP$. Now to compute $\alpha^4$ we use the that $P(\alpha) = 0$ and so $\alpha^4 = \alpha^3 + 1$.

Then,

$$\begin{aligned} \alpha^5 &= \alpha\alpha^4 \\ &= \alpha^4 + \alpha \\ &= 1 + \alpha + \alpha^3 \end{aligned}$$

And we use this method until we have 16 elements. At the end, we find the table 2.1:

| i | polynomial |
|---|---|
| -1 | $0$ |
| 0 | $1$ |
| 1 | $\alpha$ |
| 2 | $\alpha^2$ |
| 3 | $\alpha^3$ |
| 4 | $1+\alpha^3$ |
| 5 | $1+\alpha+\alpha^3$ |
| 6 | $1+\alpha+\alpha^2+\alpha^3$ |
| 7 | $1+\alpha+\alpha^2$ |
| 8 | $\alpha+\alpha^2+\alpha^3$ |
| 9 | $1+\alpha^2$ |
| 10 | $\alpha+\alpha^3$ |
| 11 | $1+\alpha^2+\alpha^3$ |
| 12 | $1+\alpha$ |
| 13 | $\alpha+\alpha^2$ |
| 14 | $\alpha^2+\alpha^3$ |
| 15 | $1$ |

Table 2.1: Finite Fied $\mathbb{F}_{2^4}$

### 2.1.3 In real life

A more common way to represent this polynomial elements is the binary representation: then $\alpha^4 = 1+\alpha^3$ becomes $1001 = 9$ (in decimal base) and $\alpha^5 = 1+\alpha+\alpha^3$ becomes $1011 = 11$ (in decimal base). This representation gives us the table 2.2:

| i | polynomial | binary | decimal |
|---|---|---|---|
| -1 | $0$ | 0000 | 0 |
| 0 | $1$ | 0001 | 1 |
| 1 | $\alpha$ | 0010 | 2 |
| 2 | $\alpha^2$ | 0100 | 4 |
| 3 | $\alpha^3$ | 1000 | 8 |
| 4 | $1+\alpha^3$ | 1001 | 9 |
| 5 | $1+\alpha+\alpha^3$ | 1011 | 11 |
| 6 | $1+\alpha+\alpha^2+\alpha^3$ | 1111 | 15 |
| 7 | $1+\alpha+\alpha^2$ | 0111 | 7 |
| 8 | $\alpha+\alpha^2+\alpha^3$ | 1110 | 14 |
| 9 | $1+\alpha^2$ | 0101 | 5 |
| 10 | $\alpha+\alpha^3$ | 1010 | 10 |
| 11 | $1+\alpha^2+\alpha^3$ | 1101 | 13 |
| 12 | $1+\alpha$ | 0011 | 3 |
| 13 | $\alpha+\alpha^2$ | 0110 | 6 |
| 14 | $\alpha^2+\alpha^3$ | 1100 | 12 |
| 15 | $1$ | 0001 | 1 |

Table 2.2: Finite Fied $\mathbb{F}_{2^4}$ with decimal representation

Note that in the following, we have for example $10 + 2 = 3$ and not $10 + 2 = 12$.

## 2.2 Erasure Coding theory

Now that we have made clear how finite field works we can now talk about erasure coding theory before seeing a code example.

### 2.2.1 Problematic

Error correcting code is a technic based on redundancy. Codes are used in a very concrete context: data transmission. In most of the cases, transmission is done via a channel and this channel is very often noisy: data can be altered when they are in it. For example, during the radio communication, interferences can disrupt the voice. The solution consists in adding redundancy which means repeating the same information in a way or another in order to still understand the information even if noise exists in the channel. It exists a lot of different codes using a wide range of techniques but the most known probably are BCH codes (from which Reed Solomon codes are a subclass) used in CDs and DVDs or Hamming codes used in the Minitel.

Correcting codes all have the same characteristic: if during the transmission the message is altered then a supplementary information is necessary in order to detect or correct the error. A transmitted message (called a code) is the original message put in a bigger space: as we will explain later the purpose of this manipulation is having messages that have not been disrupted by the channel as different as possible in order to not mixed them up: we can also say that we want to separate two different messages as far as possible.

### 2.2.2 Way of Functioning

Now that we have seen an intuitive presentation of the error correcting theory, we can take a more theoretical approach [33]. Here as in most of the cases, we assume that the channel is discrete, which means that the information we want to transmit is a succession of symbols taken into a finite set. Most of the time, we are talking of bits so symbols will be a succession of 0 and 1.

In coding theory, we work with an alphabet which is the finite set that we were talking about before and we call every element that belongs to it a symbol (or a letter). We then call a word a succession of symbols that belong to an alphabet.

Usually, the messages we want to send have not a fixed size: this is for example the case for phone communication. But the problem is that it is really hard to build a code who can take as an input a variable length message. So the solution is to build a code that can take fixed size as an input and segment the original message. The segments of the original message are called information blocks. We now need the following definitions before giving an example.

- The length of a message is the number of symbols it contains.
- A code by block is an error correcting code which can code an input message with a fixed length $k$ and return an encoded message of fixed size $n$.

### 2.2.3 Example

One of the most basic code is the repetition code. The scheme is very since it consists in repeating several time each bits. For example,

$$\begin{aligned} 1 &\mapsto 111 \\ 0 &\mapsto 000 \end{aligned}$$

So the message 010010 becomes 000111000000111000.

To decode a message, we only need to make vote every bit in a group of 3. If there is a majority of 1 then the original message was 1 and if there is a majority of 0 the original message was 0. For example,

$$\begin{aligned} 101 &\mapsto 1 \\ 011 &\mapsto 1 \\ 100 &\mapsto 0 \\ 001 &\mapsto 1 \end{aligned}$$

So the received message 000111000000111000 maps to 010010 but let's imagine that there is an error in it and that we received for example 100111000000111000, the result will still be 010010 which is the goal of this algorithm.

## 2.3 Reed-Solomon code

Reed-Solomon codes are codes by block: they take a fixed size data block as an input and returns another fixed size block. They use finite field with $2^m$ elements and thanks to redundancy they can correct up to 2 different kinds of error:

- errors with a data modification where some bits are transformed from 0 to 1 or 1 to 0,
- errors with a data loss that are called erasures rather than errors.

In practice, the main difference between these two kinds of error is the fact that in the first case we don't know how many errors happened during the transmission neither their positions or their values when in the second case, we know both the number and the position of the erasures (but obviously not their values).

In the following, we will define $RS(n, k, t)$ a Reed-Solomon code that can take as an input a block of size $k$ symbols, returns a block of size $n$ symbols and correct up to $t/2$ errors of the first type (as we will see later, it is equivalent to $t$ errors of the second type). Note that so far, the only restriction on $k$ and $n$ is $k < n < 2^m$ where $2^m$ is the size of the finite field.

This section will now take place in 3 parts: how we encode a block, how we decode it and at last an example that is here to make the algorithms easier to understand.

### 2.3.1 Coding

In this subsection, the goal is to have a look at the theoretical encoder of the Reed Solomon algorithm [2]. The encoder is the part that add redundancy to a message in order to still understand it when errors or erasures have been added.

The symbols of the encoded message are the coefficients of an output polynomial $s(x)$ constructed by multiplying the input polynomial $p(x)$ of degree $k-1$ by a generator polynomial $g(x)$ of degree $t = n-k-1$ defined by having $\alpha, \alpha^2, \alpha^3, ..., \alpha^t$ as its roots with $\alpha$ the primitive element of $\mathbb{F}_{2^m}$. We then have:

$$s(x) = p(x)g(x)$$

with:

$$\begin{aligned} g(x) &= (X-\alpha)(X-\alpha^2)...(X-\alpha^t) \\ &= g_0 + g_1 X + ... + g_{t-1}X^t + X^t \end{aligned}$$

The transmitter sends the $n-1$ coefficients of $s(x) = p(x)g(x)$. The receiver can then divide this polynomial by $g(x)$ to know if it is a message or an error. If the remainder is the zero polynomial then there is no error in the message, otherwise there is at least one error. If $r(x)$ is this polynomial, then the receiver can evaluate $r(x)$ for every power of $\alpha$, he can then eliminate $s(x)$ and identify which coefficients of $r(x)$ are an error. If the system of equations can be solved, then the receiver knows how to modify his $r(x)$ to get the most likely $s(x)$.

Even if we don't need this right now, it is important for our project (and more particularly for the software architecture chapter) to define what is the generator matrix and the systematic form of the generator matrix [16].

The generator matrix is the matrix corresponding to the linear application that take as input the message to transmit and returns the encoded message. Let's be $G$ this matrix, $m$ the vectorial representation of the message, i.e. the $k$ symbols belonging to the finite field with $2^m$ elements represented as a vector (so the vector $m$ will have a size $k$), and $c$ the vectorial representation of the coded word (of size $n$ by definition) then we have $c = Gm$. In the case of a Reed Solomon code, the generator matrix is the following:

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ \alpha^{1^1} & \alpha^{2^1} & \alpha^{3^1} & \dots & \alpha^{k^1} \\ \alpha^{1^2} & \alpha^{2^2} & \alpha^{3^2} & \dots & \alpha^{k^2} \\ \vdots & \vdots & \vdots & & \vdots \\ \alpha^{1^{n-1}} & \alpha^{2^{n-1}} & \alpha^{3^{n-1}} & \dots & \alpha^{k^{n-1}} \end{pmatrix}$$

We then have a matrix of size $k$ rows and $n$ columns and the systematic form of the generator matrix is the same matrix after doing a Gaussian Elimination in order to have the $k$ first rows (that will be the same in the original and encoded message) of this matrix as an identity matrix. We will denote $G$ this matrix in the rest of this project.

In real case (such as our project) we only use the generator matrix under its systematic form because we only need to copy the $k$ first symbols and then make the computation for the remaining encoded code.

### 2.3.2 Decoding

In this section, we will only talk about the error detection and correction. As for the coding part, we will give a more practical algorithm in the chapter dealing with the software architecture.

The first thing to know when talking about Reed Solomon decoding is that a polynomial $g(x)$ of degree $t$ as explained in the subsection on the coding can correct up to $t/2$ errors. Let's know talk about the decoding algorithm itself [17]. The received message is here seen as the coefficients of a polynomial $r(x)$ which is the sum of the encoded message $s(x) = \sum_{i=0}^{n-1} c_i x^i$and the error $e(x) = \sum_{i=0}^{n-1} e_i x^i$ ($e_i$ will be zero if there is no error for the $i$-th power of $x$ and non zero otherwise):

$$r(x) = s(x) + e(x)$$

Let's also recall the generator polynomial $g(x) = \prod_{j=1}^{n-k}(x - \alpha^j)$ where $\alpha$ is a primitive root.

The goal of the decoder is now to find the nonzero position of the error which means the nonzero $e_i$. In order to simplify the notation, we will use the following: we assume that there is $\nu < t/2$ errors and they occur on $i_k$-th power of $x$ then:

$$e(x) = \sum_{i=0}^{\nu} e_{i_k} x^{i_k}$$

So we want to find, $\nu$, $i_k$ and $e_{i_k}$. For the need of the algorithm, we also need to construct the syndromes $S_j = r(\alpha^j)$:

$$\begin{aligned} S_j &= r(\alpha^j) \\ &= s(\alpha^j) + e(\alpha^j) \\ &= 0 + e(\alpha^j) \\ &= \sum_{i=0}^{\nu} e_{i_k} \alpha^{j^{i_k}} \end{aligned}$$

We also need to define $X_k = \alpha^{i_k}$ and the error values $Y_k = e_{i_k}$ thanks to which we can write:

$$S_j = \sum_{i=0}^{\nu} Y_k X_k^i$$

Thanks to these syndromes, we can compute an important polynomial that is called error locator polynomial $\Lambda(x)$ that we can use to find the error locations and values.

Let's define $\Lambda(x)$ such as:

$$\begin{aligned} \Lambda(x) &= \prod_{i=0}^{\nu}(1 - xX_k) \\ &= 1 + \Lambda_1 x + \Lambda_2 x^2 + .. + \Lambda_\nu X^\nu \end{aligned}$$

By doing it this way, we have for every $k$:

$$\begin{aligned} \Lambda(X_k^{-1}) &= 0 \\ 1 + \Lambda_1 X_k^{-1} + \Lambda_2 X_k^{-2} + .. + \Lambda_\nu X_k^{-\nu} &= 0 \end{aligned}$$

So by multiplying by $Y_k X_k^{j+\nu}$, we get:

$$Y_k X_k^{j+\nu} + Y_k X_k^{j+\nu}\Lambda_1 X_k^{-1} + Y_k X_k^{j+\nu}\Lambda_2 X_k^{-2} + .. + Y_k X_k^{j+\nu}\Lambda_\nu X_k^{-\nu} = 0$$

and by simplifying:

$$Y_k X_k^{j+\nu} + Y_k X_k^{j+\nu-1}\Lambda_1 + Y_k X_k^{j+\nu-2}\Lambda_2 + .. + Y_k X_k^{j}\Lambda_\nu = 0$$

We can then sum from 1 to $\nu$:

$$\sum_{k=1}^{\nu}(Y_k X_k^{j+\nu} + Y_k X_k^{j+\nu-1}\Lambda_1 + Y_k X_k^{j+\nu-2}\Lambda_2 + ... + Y_k X_k^{j}\Lambda_\nu) = 0$$
$$\sum_{k=1}^{\nu}(Y_k X_k^{j+\nu}) + \Lambda_1\sum_{k=1}^{\nu}(Y_k X_k^{j+\nu-1}) + \Lambda_2\sum_{k=1}^{\nu}(Y_k X_k^{j+\nu-2}) + ... + \Lambda_\nu\sum_{k=1}^{\nu}(Y_k X_k^{j}) = 0$$

Which allows us to simplify in:

$$\begin{aligned} S_{j+\nu} + \Lambda_1 S_{j+\nu-1} + ... + \Lambda_{\nu-1}S_{j+1} + \Lambda_\nu S_j &= 0 \\ \Lambda_1 S_{j+\nu-1} + ... + \Lambda_{\nu-1}S_{j+1} + \Lambda_\nu S_j &= -S_{j+\nu} \end{aligned}$$

That is how we get the following linear equations:

$$\begin{pmatrix} S_1 & S_2 & \dots & S_\nu \\ S_2 & S_3 & \dots & S_{\nu+1} \\ \vdots & \vdots & & \vdots \\ S_\nu & S_{\nu+1} & \dots & S_{2\nu-1} \end{pmatrix} \begin{pmatrix} \Lambda_\nu \\ \Lambda_{\nu-1} \\ \vdots \\ \Lambda_1 \end{pmatrix} = \begin{pmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{\nu+\nu} \end{pmatrix}$$

We can now use these equations to find the $\Lambda_i$ to build the error locator polynomial from which the roots (found by an exhaustive search) are $\alpha^{-i_k}$. We can then deduce the $i_k$ from these values. At last, error values $e_{i_k}$ are easy to compute since we know that:

$$S_j = \sum_{i=0}^{\nu} Y_k X_k^i$$

we then have a linear equations system which is:

$$\begin{pmatrix} X_1^1 & X_2^1 & \dots & X_\nu^1 \\ X_1^2 & X_2^2 & \dots & X_\nu^2 \\ \vdots & \vdots & & \vdots \\ X_1^{n-k} & X_2^{n-k} & \dots & X_\nu^{n-k} \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_\nu \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_{n-k} \end{pmatrix}$$

Thanks to this system we can then find the $Y_k$ which gives us the $e_{i_k}$

### 2.3.3 Toy example

#### 2.3.3.1 Code Generation

Let's take the case where $m = 4$: then we are working on the finite field with 16 elements. The maximum number of symbols for the output is then $n = 2^m = 16$ (note that we could have choose any number between 0 and 16) each of size 4 bits. We then need a primitive polynomial of degree 4 for this field which can be (according to the literature or the Berlekamp algorithm) $m(x) = x^4 + x^3 + 1$. Let $\alpha = 0010 = 2$ be the primitive element of this field then $\alpha$ verify $\alpha^4 = \alpha^3 + 1$ (we can then build the field as seen in the section dealing with finite field).

Let's say that we want our Reed-Solomon code to correct up to 3 errors so the generator polynomial of the code must be of degree $t = 6$ over the finite field that contains 16 elements (note that by fixing $n$ and $t$, we have $k = n - t - 1 = 9$). We then have this generator polynomial $g(x)$ that verifies:

$$\begin{aligned}
g(x) =&(x-\alpha)(x-\alpha^2)(x-\alpha^3)(x-\alpha^4)(x-\alpha^5)(x-\alpha^6)\\
=&(x-\alpha)(x-\alpha^2)(x-\alpha^3)(x-\alpha-\alpha^3)(x-1-\alpha-\alpha^3)(x-1-\alpha-\alpha^2-\alpha^3)\\
=&x^6+(1+\alpha)x^5+x^4+\alpha^2x^3+(1+\alpha+\alpha^2)x^2+(1+\alpha^2+\alpha^3)x+1+\alpha+\alpha^2+\alpha^3
\end{aligned}$$

#### 2.3.3.2 Coding

Let's say our message is (9,8,7,6,5,4,3,2,1), which gives the polynomial $p(x)$ :

$$\begin{aligned}
p(x) =&9x^8+8x^7+7x^6+6x^5+5x^4+4x^3+3x^2+2x+1\\
=&(1+\alpha^3)x^8+\alpha^3x^7+(1+\alpha+\alpha^2)x^6+(\alpha+\alpha^2)x^5+(1+\alpha^2)x^4+\alpha^2x^3+(1+\alpha)^2+\alpha x+1
\end{aligned}$$

Recall that the encoded message is the multiplication of this polynomial with the generator polynomial $g(x)$:

$$\begin{aligned}
s(x) =&p(x)*g(x)\\
=&(x^6+(1+\alpha)x^5+x^4+\alpha^2x^3+(1+\alpha+\alpha^2)x^2+(1+\alpha^2+\alpha^3)x+1+\alpha+\alpha^2+\alpha^3)\\
&*((1+\alpha^3)x^8+\alpha^3x^7+(1+\alpha+\alpha^2)x^6+(\alpha+\alpha^2)x^5+(1+\alpha^2)x^4+\alpha^2x^3+(1+\alpha)^2+\alpha x+1)\\
=&(1+\alpha^3)x^{14}+\alpha^3x^{13}+(1+\alpha+\alpha^2)x^{12}+(\alpha+\alpha^2)x^{11}+(1+\alpha^2)x^{10}+\alpha^2x^9+(1+\alpha)x^8\\
&+\alpha x^7+1x^6+(\alpha+\alpha^2)x^5+(1+\alpha+\alpha^2+\alpha^3)x^4+(1+\alpha+\alpha^2+\alpha^3)x^3\\
&+(1+\alpha+\alpha^2+\alpha^3)x^2+(1+\alpha+\alpha^3)x+\alpha+\alpha^2+\alpha^3
\end{aligned}$$

So the corresponding encoded word is (9,8,7,6,5,4,3,2,1,6,15,15,15,11,14).

#### 2.3.3.3 Decoding

To illustrate the decoding process, we first need to add an error under for example the polynomial representation. Let's for example choose:

$$\begin{aligned}
e(x) =&7x^{11}+10x^2\\
=&(1+\alpha+\alpha^2)x^{11}+(\alpha+\alpha^3)x^2
\end{aligned}$$

The received word is then

$$\begin{aligned} r(x) =& s(x) + e(x) \\ =& (1+\alpha^3)x^{14} + \alpha^3 x^{13} + (1+\alpha+\alpha^2)x^{12} + x^{11} + (1+\alpha^2)x^{10} + \alpha^2 x^9 + (1+\alpha)x^8 + \alpha x^7 + 1x^6 \\ &+ (\alpha+\alpha^2)x^5 + (1+\alpha+\alpha^2+\alpha^3)x^4 + (1+\alpha+\alpha^2+\alpha^3)x^3 + (1+\alpha^2)x^2 \\ &+ (1+\alpha+\alpha^3)x + \alpha + \alpha^2 + \alpha^3 \end{aligned}$$

As explained in the algorithm, we then compute the syndromes $S_i = r(\alpha^i) = r(2^i)$ for $i = 1..6$, which gives us:

$$\begin{aligned} S_1 =& 1+\alpha+\alpha^3 \\ S_2 =& 0 \\ S_3 =& \alpha^3 \\ S_4 =& 1+\alpha+\alpha^2 \\ S_5 =& 1+\alpha+\alpha^2+\alpha^3 \\ S_6 =& 1 \end{aligned}$$

We can now compute the following linear equations:

$$\begin{pmatrix} 1+\alpha+\alpha^3 & 0 & \alpha^3 \\ 0 & \alpha^3 & 1+\alpha+\alpha^2 \\ \alpha^3 & 1+\alpha+\alpha^2 & 1+\alpha+\alpha^2+\alpha^3 \end{pmatrix} \begin{pmatrix} \Lambda_3 \\ \Lambda_2 \\ \Lambda_1 \end{pmatrix} = \begin{pmatrix} 1+\alpha+\alpha^2 \\ 1+\alpha+\alpha^2+\alpha^3 \\ 1 \end{pmatrix}$$

Thanks to which we find: $\Lambda_1 = 1+\alpha^3$, $\Lambda_2 = \alpha+\alpha^2$ and $\Lambda_3 = 0$ and the locator errors polynomial is:

$$\Lambda(x) = (\alpha+\alpha^2)x^2 + (1+\alpha^3)x + 1$$

An exhaustive search of the roots of this polynomial gives $\alpha^4$ and $\alpha^{13}$ so there are only 2 errors and their positions are: $15-4=11$ and $15-13=2$.

Now that we have the positions, we still need the values, which is easy since we only have to resolve the following system:

$$\begin{pmatrix} \alpha^2 & \alpha^{11} \\ \alpha^4 & \alpha^{22} \\ \alpha^6 & \alpha^{33} \\ \alpha^8 & \alpha^{44} \\ \alpha^{10} & \alpha^{55} \\ \alpha^{12} & \alpha^{66} \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} 1+\alpha+\alpha^3 \\ 0 \\ \alpha^3 \\ 1+\alpha+\alpha^2 \\ 1+\alpha+\alpha^2+\alpha^3 \\ 1 \end{pmatrix}$$

Which is equivalent to:

$$\begin{pmatrix} \alpha^2 & 1+\alpha^2+\alpha^3 \\ 1+\alpha^3 & 1+\alpha+\alpha^2 \\ 1+\alpha+\alpha^2\alpha^3 & \alpha^3 \\ \alpha+\alpha^2\alpha3 & \alpha^2+\alpha^3 \\ \alpha+\alpha^3 & \alpha+\alpha^3 \\ 1+\alpha & 1+\alpha+\alpha^2+\alpha^3 \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \end{pmatrix} = \begin{pmatrix} 1+\alpha+\alpha^3 \\ 0 \\ \alpha^3 \\ 1+\alpha+\alpha^2 \\ 1+\alpha+\alpha^2+\alpha^3 \\ 1 \end{pmatrix}$$

The resolution of this system gives:

$$\begin{aligned} Y_1 =& \alpha + \alpha^3 \\ Y_2 =& 1 + \alpha + \alpha^2 \end{aligned}$$

This gives us the polynomial $e(x)$ and by using it we can find the original encoded message which is the original goal of this algorithm.

Note that in this example as in the whole chapter, we only deal with classic errors (what we named before errors of the first type), we will deal with erasures (errors of the second type) in the software architecture part.

Also, this chapter was the theoritical explanation of coding theory and Reed-Solomon code, we will see a more usable way to encode and decode a message in the Software Architecture part.

# Chapter 3

# Minimum Bandwidth Exact Repairing Algorithm

This chapter is here to explain the algorithm on which is based the software: Minimum Bandwidth Exact Repairing code [20]. To do so, we will first see a quick overview of what is Minimum Bandwidth Exact Repairing code, then see how the algorithm works in more details and finally we will see a quick example to have a better understanding of the problem.

## 3.1 Overview

This section is here to put into context the algorithm. We will first see what it is used for and then we will try to classify compared with other kinds of algorithms.

### 3.1.1 Problematic

Let's directly take an example to illustrate what we are dealing with. Imagine that you have a huge file and 4 different servers (called nodes) at your disposal. How are you going to distribute the message on all of these nodes ? The easiest answer will be to cut the file in 4 and put each part on one of the nodes like on the figure 3.1 ($a$, $b$, $c$ and $d$ represents the parts of the file):

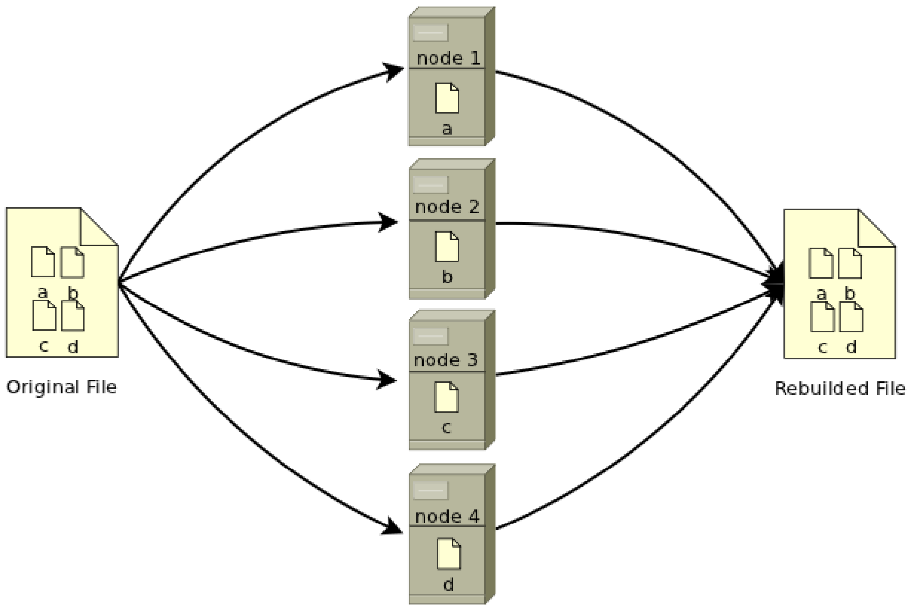


Figure 3.1: Distributed Storage System without redundancy

The problem is now: what happens if as explained in the introduction, one of the node failed ? Can we still reconstruct the original file from the 3 remaining nodes ? As we can see on the figure 3.2 the answer is obviously no.

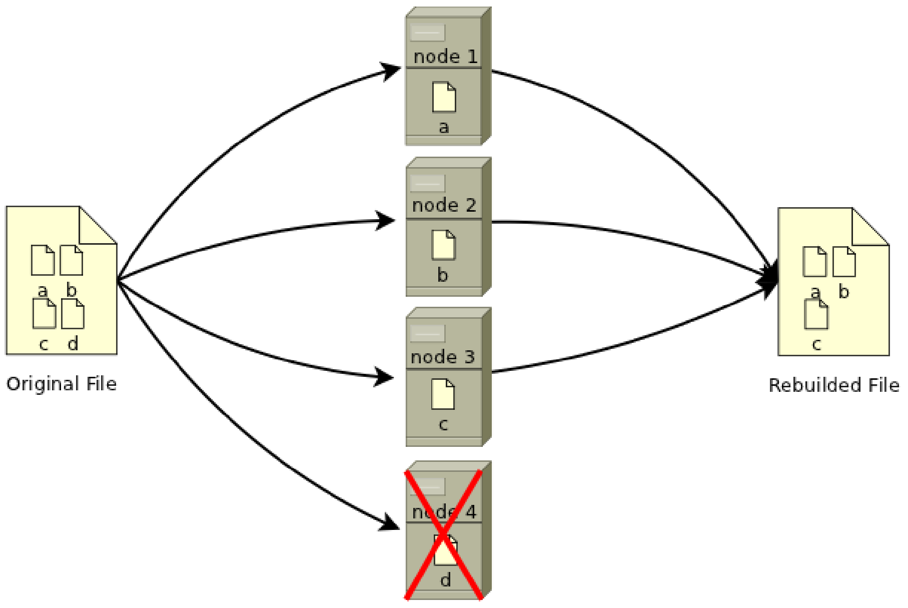


Figure 3.2: Node Failure in a Distributed Storage System without redundancy

The way we store our data on several is called distributed storage and here is a more formal definition of what is really distributed storage: ”distributed system is a collection of independent computers that appear to the users of the system as a single computer” [29].

This definition highlights a very important problem that we have to deal with in distributed storage: everything has to be transparent for the user which leads to the reliability problem. How can we achieve maximum reliability ? The obvious answer is as easy as it is impossible in practice: we store the entire message on each available device. That would be great if we had no bandwidth or storage constraint because we could successfully rebuild the original message as long as we gain access to any one node.

The whole problem is that we have these constraints to take into account and this why the next section is useful.

## 3.1.2 Classification

As we have said before, distributed storage refers to the way we distribute the files on different devices. The fact is that there is a lot of ways to do it and it most depends on the features that we want to give our system. That is what this subsection is for: give an explanation for some well-known systems that are related to our project.

### 3.1.2.1 Erasure code

Like the figures in the problematic were showing us, the problem we have to take into account is node failure. The solution found to this problem is redundancy. Let's take an easy example: as before we have a big file but this time we have 5 nodes. We still cut our file into 4 parts but we use the last node to add redundancy in the following form: $a + b + c + d$ which gives us the figure 3.3 (here we can rebuild the original thanks to any 4 out of the 5 nodes).

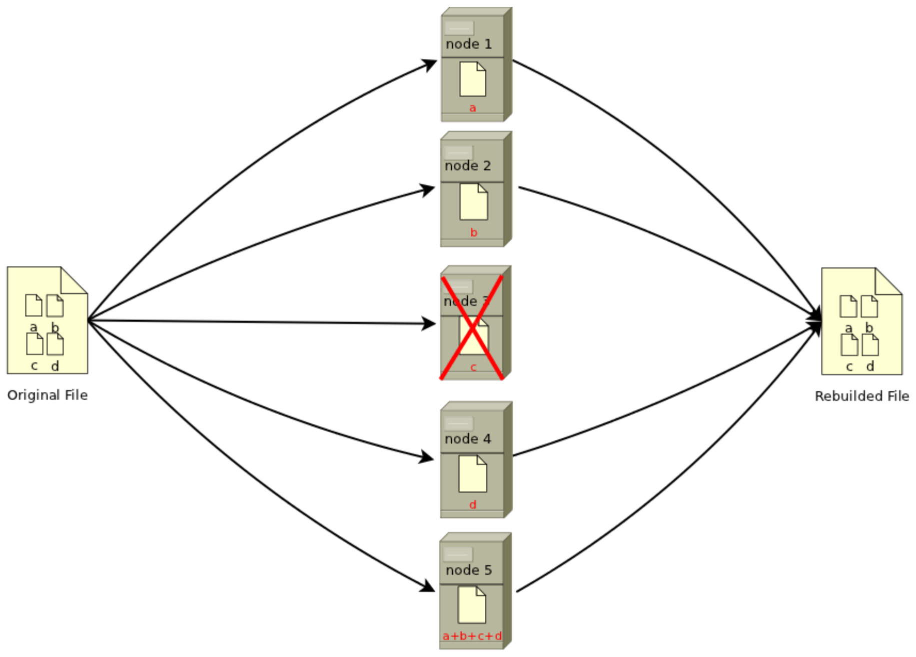


Figure 3.3: Node Failure in a Distributed Storage System Based on Erasure Code

As shown on the figure, we can rebuild the original file from any 4 out of 5 nodes. For example, if the first node fails we can still rebuild the original file from the 4 remaining nodes since we have $b$, $c$ and $d$ from node 2, 3 and 4 but also $a$ thanks from node 5: $a = (a + b + c + d) - b - c - d$.

This example is really simple but still usable for real systems using erasure codes as we have seen in the previous chapter [6]. Indeed, let's take for example a Reed Solomon code RS(16,10,6): it means that we can correct up to 9 erasures so if we take a word (or a file) of length 10, we can put it on 16 nodes. Let's consider that 6 nodes fail then every missing symbol is considered as an erasure. And as our code can correct up to 6 erasures we can still rebuild the whole original file.

This kind of system takes place in any environment where we can not be sure of the reliability of every node which means pretty in every case since the life time of a storage device is not infinite.

#### 3.1.2.2 Repairing code

Again, the whole problem is that node can fail. So what do we do when it happens ? The easiest answer would consist in rebuilding the whole file on the client computer and then resending it to the node that has failed in order to repair it. As you can imagine, this is not the best solution since it is both very time consuming and very resources consuming. So what can we do in this case ?

The solution depends on what property we want to give to our system. Let's say here that the only thing we want is repairing the failed so we don't care what is really in the node: we just want it to allow the rebuilding of the original file if another node was to failed. Let's take the example base on figure 3.4:

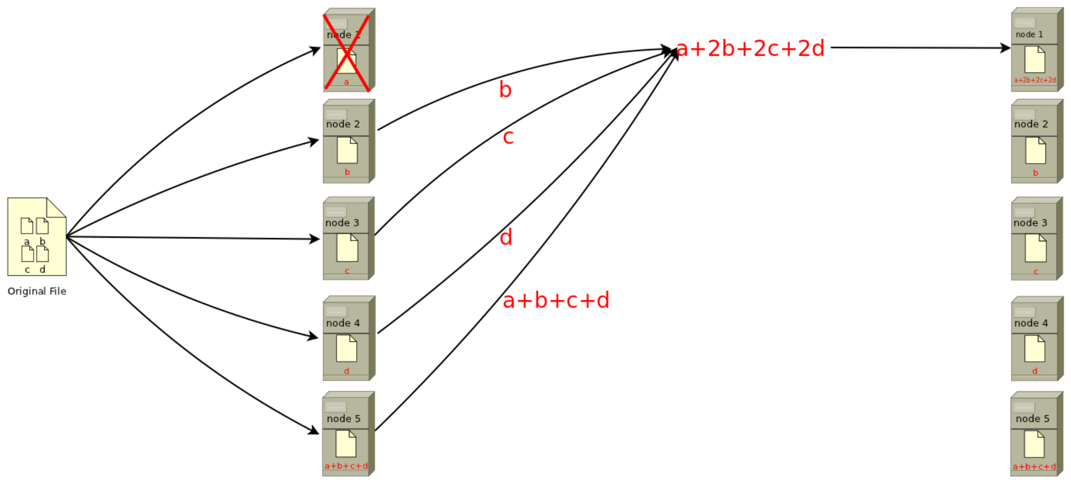


Figure 3.4: Node Failure in a Distributed Storage System based on Repairing code

Here the first node (the one that contains $a$) has failed and in order to maintain the system we need to repair it (we can't let every node fails one by one until there is no one left). The problem is that we don't have $a$ anywhere, we only have $a+b+c+d$ in node 5. In the example, we chose to repair the node with $a+2b+2c+2d$, you can notice that by doing it this way, we can still rebuild the original file from any 4 out of 5 nodes. Indeed, if we for example take the 4 first nodes, we have $b$, $c$, and $d$ from node 1, 2 and 3 and node 1 allows to build $a = a+2b+2c+2d-2b-2c-2d$. In fact, we keep the functional aspect of the system [7]. It is opposed by definition to the exact repairing that we will see in the next subsection.

This functional repairing is often preferred because it is easier (less computation). But the problem is that every node has to be aware of the state of the other nodes (which is much more complicated in practice) that is why we justify the need for an Exact Repairing code.

#### 3.1.2.3 Exact Repairing code

So why are the exact repairing codes useful ? There are at least two reasons for that. The first one is that as we said we don't need any computation to be done on the nodes. As you have seen in the previous example, we for example need to subtract the content of the 3 last nodes from the last node. Repeated a huge number of time, this kind of operation can be very CPU consuming (adding the fact that in real cases the operations are not that simple).

The second reason is that every node has to be aware of the state of the other nodes (in order to know which exact operations have to be done). It is first very difficult to maintain since the states of the nodes change at every step. Furthermore, it is very bandwidth consuming since we have to transmit the states to the other nodes.

The following figure shows how we can construct an easy example of an exact repairing system: note that when in the previous section, the system was not able to reconstruct the exact node, it is now possible. Also in order to make it easier to understand, we now store 2 values at each node. For example, node 1 stores $a$ and $b$, node 2 stores $c$ and $d$, node 3 stores $a + c$ and $b + d$ and node 4 stores $b + c$ and $a + b + d$. For every file $abcd$, each node stores 2 parts of the file.

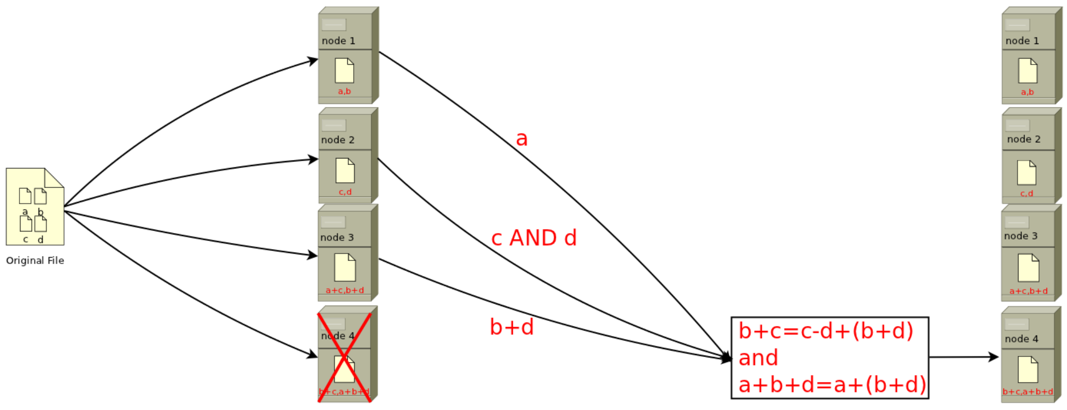


Figure 3.5: Node Failure in a Distributed Storage System based on Exact Repairing code

In figure 3.5, the node which fails is the node 5 and it contains both the values $b + c$ and $a + b + d$ and the challenge is to reconstruct it. To do so, we use the value $a$ from node 1, value $b + d$ from node 3 and both values ($c$ and $d$) from node 2 as you can see on the figure. Note that to reconstruct the values of node 4 we need 2 values from node 2 (and not only 1) which means that the bandwidth used in the system is not optimised.

#### 3.1.2.4 Minimum Bandwidth Exact Repairing code

As we will see in the next section, to repair a node we always have to find a trade off between the necessary bandwidth and the storage [35]. As we have seen in the previous example, when a node that stores several values (as it is the case in most of the real system) fails, it becomes difficult to repair it by transferring the minimum amount of data from the other nodes (which means only one value out of all of those that are present on the node). As you can see on the figure 3.6, here we use only one of the 2 values that are available.

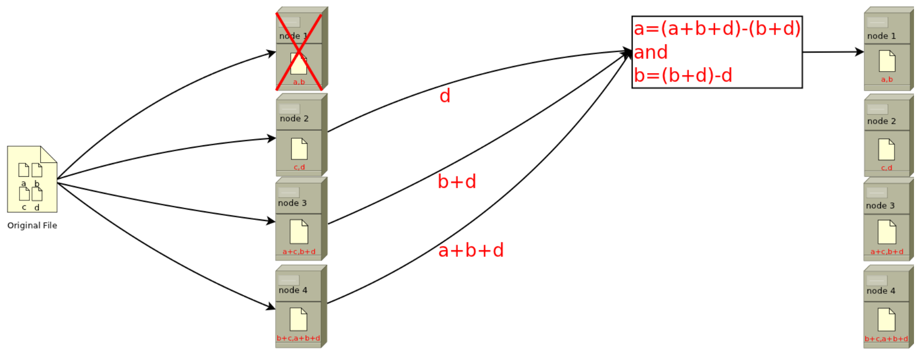


Figure 3.6: Node Failure in a Distributed Storage System based on Minimum Bandwidth Exact Repairing code

As you can see on this example, node 1 has failed and we want to rebuild exactly $a$ and $b$ using only 1 value from any of the other nodes we then use $d$ from node 2 and $b+d$ from node 3 which gives $b = (b+d) - d$ and $a+b+d$ from node 4 and $b+d$ from node 3 which gives $a = (a+b+d) - (b+d)$.

This kind of system is very useful when we want to easily maintain (refers to the exact repairing code) an maintain very quickly (refers to the fact that we minimise the bandwidth). It is for example used for web mail server.

As for each of the different systems we have illustrated in this section, there are a lot of possible algorithms that we can apply. Here we will describe, one of this algorithm for the Minimum Bandwidth Exact Repairing code [20].

## 3.2 Functioning

Now that we have classified our algorithm compared with the other ones and what are its advantages, we can begin to really explain how it works first by describing the algorithm and then showing an example.

### 3.2.1 Context

The basic feature of a distributed storage system is that it can retrieve all the information thanks to only $k$ out of the $n$ initial nodes. The first feature we want to add to the original system is that we want the system to be able to repair itself if one node fails (note that we can still recover all the information thanks to $k$ nodes). The kind of codes used for this property are called regenerating codes because we can repair nodes thanks to other ones. Obviously, the amount of data that we need to download from the existing nodes are an important parameter that we need to control during the process of recovery : it will be preferable to download as few data as possible in order to improve the speed of the recovery.

But what is gained somewhere has to be lost elsewhere and to minimize this bandwidth, we need to store more information than necessary on every node. We must then find a trade-off between the amount of data that we need to store on every node and the minimum bandwidth used to repair a node. In [35] the authors establish a trade-off between the amount of storage required at each node and the repair bandwidth. We can isolate the two extremes points on this storage-repair bandwidth trade-off curve that are the minimum bandwidth regeneration (MBR) point where we need the minimum bandwidth, and the minimum storage regeneration (MSR) point where we need the minimum storage to repair a node. You can see on the figure 3.7 that the optimal trade-off curve between storage $\alpha$ and required bandwidth $\gamma$ for $k = 5$ and $n = 10$ (this curve is extracted from [7].

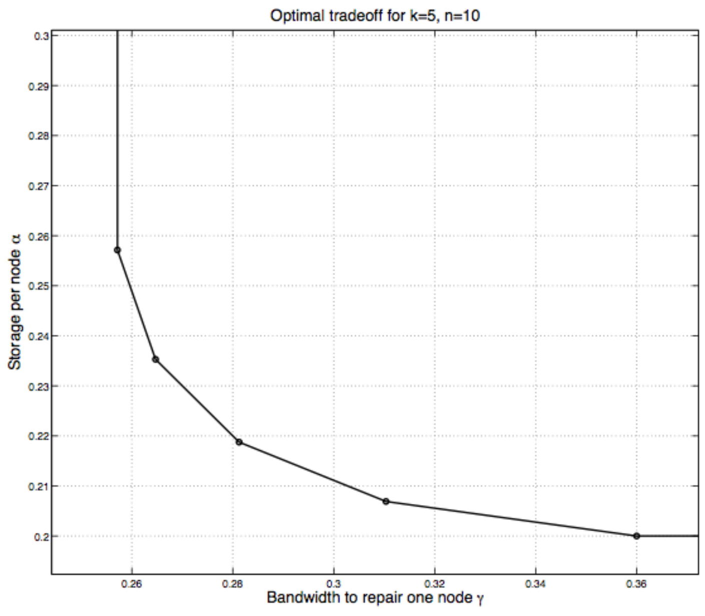


Figure 3.7: Trade-off curve between storage $\alpha$ and required bandwidth $\gamma$ for $k = 5$ and $n = 10$

As you can see on this figure, the MSR point is at the right-end of this curve and the MBR point is at the left end point so both of them cannot be minimize at the same time.

On the figure 3.8 you can see an illustration of what we mean by minimising the bandwidth but not the storage needed. Let's assume that we send $p$ units of data to each server (in this example $p = 2$ since you can see that there is 2 values on each node). Let's now imagine that node 4 fails, we then want to reconstruct it using the minimum amount of data ($q = 1$) in order to have the minimum possible amount of data in the network.

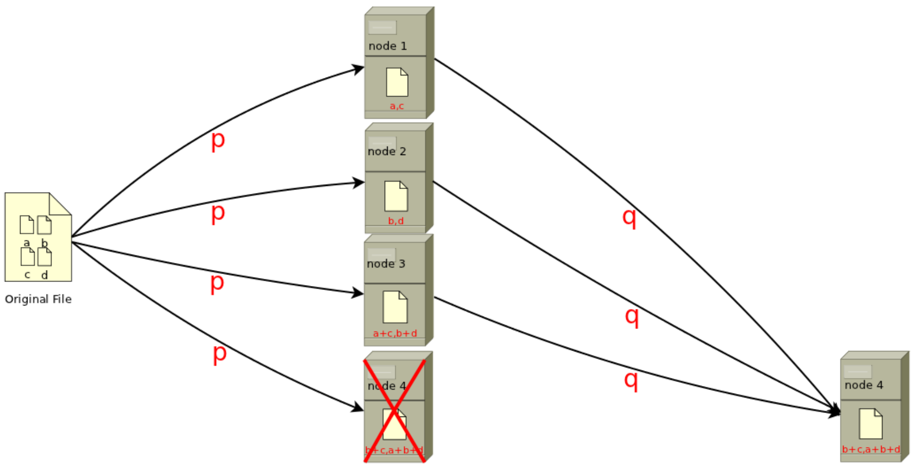


Figure 3.8: Bandwidth Minimization in a Distributed Storage system

### 3.2.2 System Construction

Now that we have made clear that we want to try to minimize the bandwidth using Minimum Bandwidth Exact Regenerating code, we need to introduce the notion of Exact regeneration. Contrary to functional repair, where when only one node fails we want to repair it in such a way that we can still retrieve all the information from $k$ out of the $n$ nodes (i.e. the information contained in the node at the end of the repairing process does not have to be the same that what it was before the failure), exact regeneration will restore the node in the exact state where it was before the failure. This property is useful because we don't need to do more computation on the client (the process who wants to retrieve the whole information) and the other servers don't have to be aware of the state of the servers (nodes).

Here the codes we want to study are Minimum Bandwidth Exact Repairing codes and more particularly the one described in [20]. As we chose that we want to repair only one node at a time we then have $d = n - 1$ remaining nodes. According to [3], we then get the maximum file size that can be stored: $B = \frac{kd-k(k-1)}{2}$. For any larger file size, the source file is split into chunks of size B, each of which can be separately solved. When then use the following steps to create our system: Denote the source symbols of the file by $\underline{f} = (f_0, f_1, f_2, ..., f_{B-1})^t$. Let $d = n - 1$ and $\theta = \frac{d(d+1)}{2}$ and let $V$ be a $n * \theta$ matrix with the following properties:

1. Each element is either 0 or 1.
2. Each row has exactly $d$ 1s.
3. Each column has exactly two 1s.
4. Any two rows have exactly one intersection of 1s.

It is easy to see that $V$ is the incidence matrix of a fully connected undirected graph with n vertices (it will be clearer in the example). Our construction of exact regenerating codes for the MBR point uses the above described matrix $V$. Consider a set of $\theta$ vectors $\{\underline{v}_1, \underline{v}_2, ..., \underline{v}_\theta\}$ which forms a $B$-dimensional code (this is for example the generator matrix of a Reed-Solomon code). The vectors $\underline{v}_i (i = 1, ..., \theta)$ are of length $B$ with the constituent elements taken from the field $\mathbb{F}_q$ such as $\underline{f}^t \underline{v}_1$, $\underline{f}^t \underline{v}_2$, ..., $\underline{f}^t \underline{v}_\theta$ represents the encoded message (the message after coding with a Reed-Solomon code). Node $j$ stores the symbol $\underline{f}^t\underline{v}_i$ if and only if $V(j,i) = 1$. Thus in the graph corresponding to $V$, vertices represent the nodes, and edges represent the vectors corresponding to the symbols stored. Thus, by the properties of the matrix $V$, we get n nodes each storing $d$ symbols.

### 3.2.3 Data Reconstruction

Let's assume that the client can have access to any $k$ out of the $n$ storage nodes and retrieves all the symbols stored on it ($k*p$ since there are $p$ units of data on each node). As any two rows of the matrix $V$ intersect in only one column and any row intersects all other rows in distinct columns, if we take in consideration the $k*p$ downloaded symbols, there are exactly $\binom{k}{2}$ symbols that are repetitions and do not add any value. Hence the client has $k*p-\binom{k}{2} = B$ distinct symbols of a B-dimensional erasure code. By using these values he can easily obtain $(f_0, f_1, f_2, ..., f_{B-1})$.

### 3.2.4 Exact Regeneration

The matrix V provides a special structure to the code which helps in exact regeneration. As each row of the matrix has exactly two 1's and that any two rows have exactly one intersection of 1s, it means that every $n-1$remaining nodes contains exactly one distinct symbol of the failed node. So exact regeneration of the failed node is possible by downloading only one symbol from the remaining $n-1$ nodes.

## 3.3 Toy Example

What has been said in the previous section is a bit theoretical and that why the following example could give some help to the reader.

### 3.3.1 System Construction

Let's assume that we have 5 nodes at our disposal and that we want to give enough redundancy to reconstruct a file thanks to only 3 of them which means $n = 5$ and $k = 3$. Then $d = n-1 = 4$, $\theta = 10$ and $B = 9$. According to the algorithm described in the previous section the nodes contain what is described in table 3.1:

| | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ | $v_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $n_1$ | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $n_2$ | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| $n_3$ | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| $n_4$ | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| $n_5$ | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |

Table 3.1: Nodes Contents

In this table, be careful that when we note $v_1$ we mean $f^t v_1$ which means the multiplication of the original message vector $f^t$ by the vector code $v_1$. This table is also equivalent to the following graph: the vertices represent the nodes and edges represent vectors corresponding to the common symbol between two nodes.

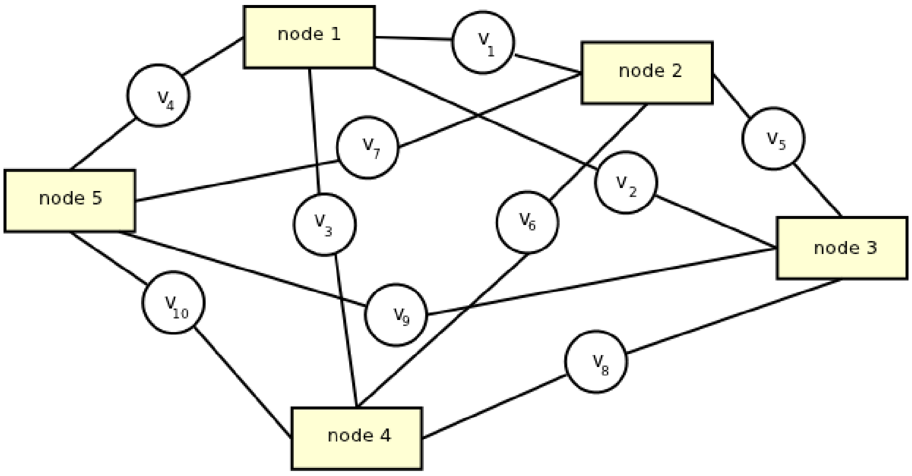


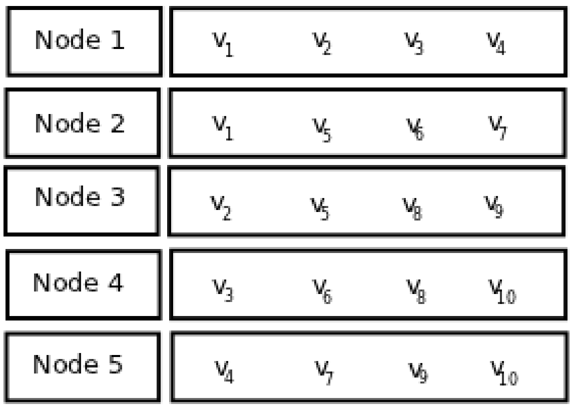
Figure 3.9: Incidence Graph representing the connection between the nodes of the system

The graph on figure 3.9 is very important since it shows that it is fully connected so every node is linked to every other node by a distinct symbol so each of them can be rebuild.

The interpretation of this matrix means that the nodes store the following data:

- Node 1 stores: $\{\underline{f}^t\underline{v}_1, \underline{f}^t\underline{v}_2, \underline{f}^t\underline{v}_3, \underline{f}^t\underline{v}_4\}$.
- Node 2 stores: $\{\underline{f}^t\underline{v}_1, \underline{f}^t\underline{v}_5, \underline{f}^t\underline{v}_6, \underline{f}^t\underline{v}_7\}$.
- Node 3 stores: $\{\underline{f}^t\underline{v}_2, \underline{f}^t\underline{v}_5, \underline{f}^t\underline{v}_8, \underline{f}^t\underline{v}_9\}$.
- Node 4 stores: $\{\underline{f}^t\underline{v}_3, \underline{f}^t\underline{v}_6, \underline{f}^t\underline{v}_8, \underline{f}^t\underline{v}_{10}\}$.
- Node 5 stores: $\{\underline{f}^t\underline{v}_4, \underline{f}^t\underline{v}_7, \underline{f}^t\underline{v}_9, \underline{f}^t\underline{v}_{10}\}$.

This can more easily be understood on the figure 3.10:



| | | | | |
|---|---|---|---|---|
| Node 1 | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
| Node 2 | $v_1$ | $v_5$ | $v_6$ | $v_7$ |
| Node 3 | $v_2$ | $v_5$ | $v_8$ | $v_9$ |
| Node 4 | $v_3$ | $v_6$ | $v_8$ | $v_{10}$ |
| Node 5 | $v_4$ | $v_7$ | $v_9$ | $v_{10}$ |

Figure 3.10: Representation of the nodes contents

On this example it is very easy to understand the underlying idea of the system. If we look at the first node, the idea is to have other nodes that store each block that already belongs to the first node. Using it we have:

- node 2 stores $\underline{f}^t\underline{v}_1$
- node 3 stores $\underline{f}^t\underline{v}_2$
- node 4 stores $\underline{f}^t\underline{v}_3$
- node 5 stores $\underline{f}^t\underline{v}_4$

In order to ensure repair, it suffices to have only one duplicated block between any two storage nodes. Hence, node 2 can store 3 more blocks: $\underline{f}^t\underline{v}_5$, $\underline{f}^t\underline{v}_6$, $\underline{f}^t\underline{v}_7$ in their remaining spaces. By using the same process:

- node 3 stores $\underline{f}^t\underline{v}_5$
- node 4 stores $\underline{f}^t\underline{v}_6$
- node 5 stores $\underline{f}^t\underline{v}_7$

We keep using this process until all the spaces in all the nodes are occupied as you can see on the figure 3.11.

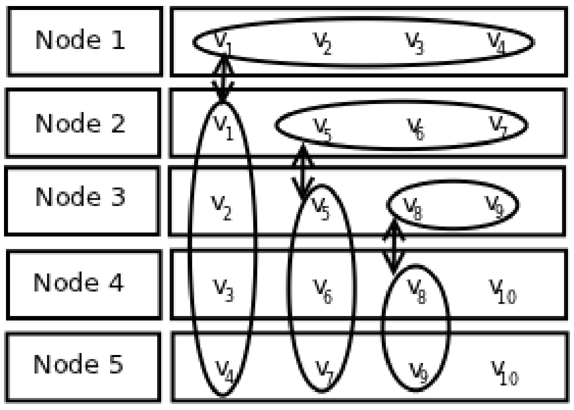


Figure 3.11: Duplication of the Data in the nodes

### 3.3.2 Data Reconstruction

The goal of this subsection is to retrieve the encoded message $\underline{f}^t\underline{v}_1$, $\underline{f}^t\underline{v}_2$, $\underline{f}^t\underline{v}_3$, $\underline{f}^t\underline{v}_4$, $\underline{f}^t\underline{v}_5$, $\underline{f}^t\underline{v}_6$, $\underline{f}^t\underline{v}_7$, $\underline{f}^t\underline{v}_8$, $\underline{f}^t\underline{v}_9$, $\underline{f}^t\underline{v}_{10}$.

Let's suppose we have access to node 1, 2 and 3, which means that we have access to $\underline{f}^t\underline{v}_1$, $\underline{f}^t\underline{v}_2$, $\underline{f}^t\underline{v}_3$, $\underline{f}^t\underline{v}_4$, $\underline{f}^t\underline{v}_5$, $\underline{f}^t\underline{v}_6$, $\underline{f}^t\underline{v}_7$, $\underline{f}^t\underline{v}_8$, $\underline{f}^t\underline{v}_9$. So we still need to find $\underline{f}^t\underline{v}_{10}$.

The main idea here is that the retrieved symbols are seen as an encoded message in which some symbols are missing. The missing symbols are considered as erasures and then our Reed-Solomon code can correct them. We can then retrieve $\underline{f}^t\underline{v}_{10}$ to find the encoded message and finally the original message.

The same holds for the 4 other nodes.

### 3.3.3 Exact Regeneration

Let's now suppose node 2 fails. Then, node 1 provides $\underline{f}^t\underline{v}_1$, node 3 provides $\underline{f}^t\underline{v}_5$, node 4 provides $\underline{f}^t\underline{v}_6$ and node 5 provides $\underline{f}^t\underline{v}_7$. All these four symbols are stored as the new node 3.

Thus the regenerated node 2 stores exactly the same symbols as the failed node as you can see on the figure 3.12:

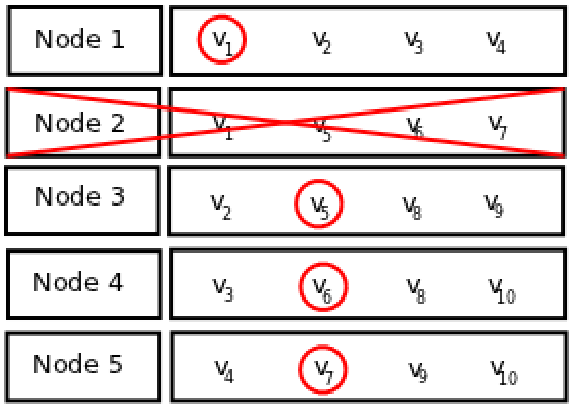


Figure 3.12: Repairing of a node

The same holds for the 4 other nodes.

# Chapter 4

# Software Architecture

This is a very nice approach of the problem but the point is that it has never been really implemented and used under real conditions: that will be the aim of the rest of this project. The problem of this kind of systems is that there is no empirical study about it which makes it difficult for someone to know for example which kind of hardware is necessary and that is why a piece of software was needed.

## 4.1 Overview

In order to test the performance of this algorithm, we need to build an environment that will simulate real conditions as good as possible. This environment is inspired from [11] for the ideas of the different layers but is very different in the way that the authors were interested in RAID systems (and a very different version of Minimum Bandwidth Exact Regenerating code which only supports $k = n - 1$) while our version deals only with Minimum Bandwidth Exact Regenerating codes but with every possible values of $k$ such as it was presented in [20]. It appears that, by the end of this project, the authors of [11] released a new version of their software dealing now with the whole range of $k$.

### 4.1.1 Functionality

This subsection is here to explain what the program can achieve and how to use it. We want it to be able to have the properties of the Minimum Bandwidth Exact Repairing code which are [20]:

- store any data on $n$ nodes by adding redundancy
- being able to reconstruct data from only $k$ nodes out of the $n$ available ones
- repair itself when one node fails

#### 4.1.1.1 Node Creations

The first problem that has to be resolved in this project is maybe the most important one: how to store the data ? The answer was found in the observation of common utilities such VirtualBox, Nero Burning Rom or every other software that needs to use virtual disks: they use IMG file. This is very convenient since it is supported by the GNU project RaWrite [8] and is used to mount virtual environments or for numerical storage.

An IMG file is an archive format used for creating a disk image and by doing so it allows to store data by exact replication on device such as hard drive disks. This exactly suits the requirement of our project and that is why every node will then be stored on IMG file. They can be easily created on linux using for example the following command [23]:

```
dd if=/dev/zero of=file.img bs=1M count=5
```

Here we will create a file named `file.img` of 5 MB.

Of course, our software can not read and write directly in these IMG files. After creating them, we need to mount them on a device. To begin, let's suppose that every file are stored on the same computer (which can not be the case since it would decrease the transfer speed). We then just have to run the following command to mount a file as a device [25]:

```
losetup /dev/loop1 file.img
```

Here the file named `file.img` will be mounted on the device named `/dev/loop1$`.

#### 4.1.1.2 File Managing

Once we have created and mounted the nodes and installed the software (running the *make* command), we only need to put the name of the devices (and not the name of the IMG files) in the settings file along with other information such as the number of nodes and run the following command:

```
./mbr -f <rootdir> <mountdir>
```

where `<mountdir>` is the folder where the user can put its file and `<rootdir>` is the place where the files stored are listed (idea from [11]). Be careful to always put the `-f` since it is the only way to run FUSE in the foreground (we will come back on FUSE later in this document). You can also use `-d` to run the debugging mode of FUSE.

Using another terminal, we can then put files, folders, links in it as we would do for a real folder. Except that it is more like an additional drive. In fact, when using with a Linux distribution like Ubuntu, the folder appears on the desktop as it would with a flash drive.

The utilisation is then completely transparent for the user: he can use the linux `cp` command or the `mv` command as he would with a common folder except that it is stored on several devices. Here again the user does not have to worry on which device he stores the data, the software will choose by itself (as it is required in the definition of a distributed storage given in chapter 3).

You can see an illustration of the system on the following figure 4.1:

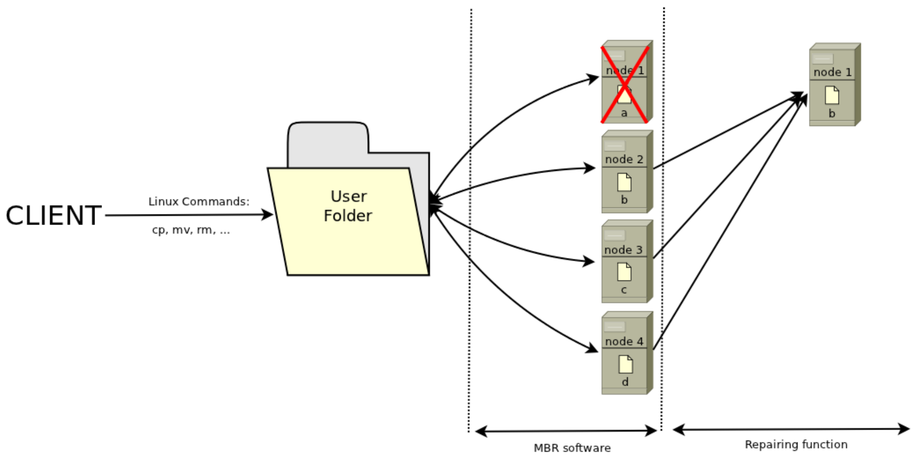


Figure 4.1: User Folder

#### 4.1.1.3 Node Repairing

Let's assume that we want to simulate the failure of a node. Then, once the program is stopped, you just need to change some parameters as we will see in the System Configuration section and run the following command:

```
./repairing\ <device>
```

where the `<device>` is the new device on which will be stored the data that were stored on the device that has failed.

As you can see on figure 4.2, the system will take care of reading the data from survival nodes and then write them on the new replacement node.

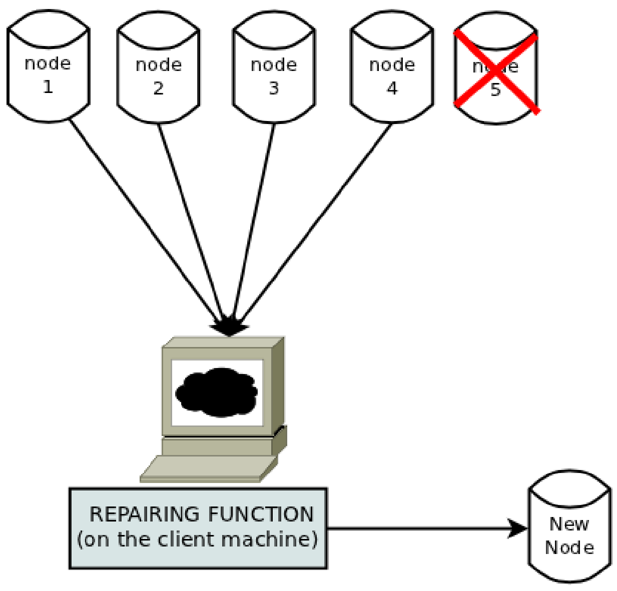


Figure 4.2: Node Repairing

#### 4.1.1.4 System Configuration

If you want to run the software, you need to edit the `settings` and `status` from which an example can be found in appendix $A$ and $B$. Note that in the `settings` example:

- `n_param` represents the number of available nodes $n$,
- `k_param` represents the number of nodes $k$ from which the system has to be able to retrieve a file,
- `block_param` is the size of a block (definition of a block will be given in the architecture of the software),
- `cache_param` allows the user to tell the system if he wants to use the cache or not
- `dev` allows the user to specify the name of each disk
- `free_offset` is the position of the first available byte
- `dev_size` is the size available on the disk

In the `status` example 1 represents a failed disk and 0 a non failed disk.

Once these file are edited, you just need to run the following command:

```
./mbr -f <rootdir> <mountdir>
```

Finally, if one wants to simulate the failure of a node, he can change its status in the `status` file and then run the previous command again to check that his data are still reachable or run the folowing command in order to repair it if there is only one faile node.:

```
./repairing\ <device>
```

More details are provided in appendix $C$. Even if one can not test the software on a real distributed storage system because he has not access to enough different computers, he can still run it on a single computer thanks to this appendix. However, note that the performances in the case of a single one can not be compared to the ones of a real distributed system.

### 4.1.2 Structure

One of the great idea of [11] reused in this project was to use FUSE to create a new file system. It appears that it is the best idea to achieve this kind of feature. When the program is running, it creates a folder in which the user can copy its own files. The program will then take care of coding and sending the files to the right node: it is completely transparent for the user which does not have to care about the coding or to have any knowledge about the state of the servers. FUSE is very convenient for this kind of things since we can manage every operation such as read, write and delete and we just have to take into account the need for coding.

The file system layer then calls for the coding layer that is the implementation of what has been explained in the previous part. The coding layer is responsible for the encoding and the decoding functions of the Minimum Bandwidth Exact Regenerating code algorithm.

Once the block is encoded the system passes it to the input/output layer that caches the accessed blocks in main memory. Then the read and write operations can directly access blocks via memory without accessing the storage nodes. We then just have to send the coded block to the selected node and on the selected location on the node. The media on which the coded blocks are stored depending on the user but to better reflect the reality we use PlanetLab in order to send the data to distant servers according to the incidence matrix $V$ described in the second part.

The figure 4.3 summarizes this structure:

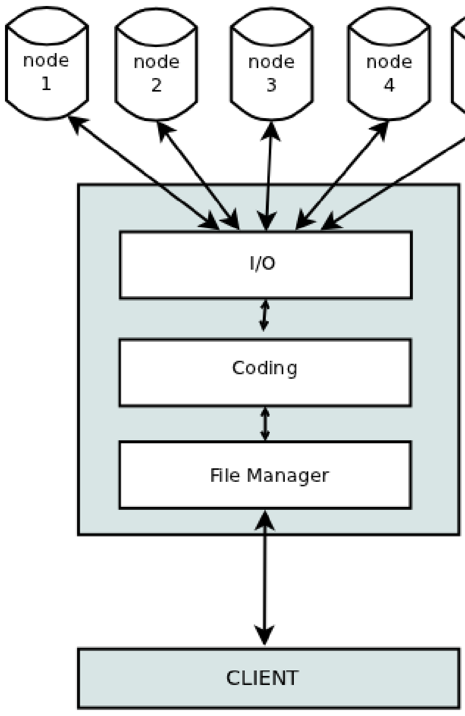


Figure 4.3: Layers

### 4.1.3 Storage

A distributed storage can be set up on different environment as well as it is on different computers (devices). In practice, we can then distribute the servers on a local network (as the one from Imperial College) where every device shares an ethernet connection which is very fast or on a distant network. In the last case, we need to transfer the data over an internet protocol which is much slower than an ethernet network.

In order to take into account these two configurations, we need two different environments: the first one is an ethernet network that in our case will be the local network of Imperial College and in the second one will be the PlanetLab nodes. A description of PlanetLab will be given later but here it will be used as node that can only be accessed via an internet connection.

#### 4.1.3.1 Imperial College Computers

This paragraph is here to explain how we store data over the Imperial College network. As we said before, the data are stored in different IMG files that each represents one node but in order to take into account the speed of the hard drive disk, we need to store these files on different computers (otherwise the files would have been store on only one computer and the transfer speed will be wrong).

The problem with Imperial College labs computers is that everything is stored on servers and we can not control what happens or where the file are stored and we so can not even be sure that everything is not stored on only one server disk.

The solution to this problem was found by talking with the CSG: there is a local disk on every machine of the laboratories. So we can store our files on different machine under the location `/data`. But here again, we have another problem: we must be able to access these files from the client machine (which is also a laboratory computer). To do so, we need to mount the data from each nodes on the client machine using ssh protocol. Let's assume we use the machine named `fusion01` as the client and `fusion02`, `fusion03` and `fusion04` as the nodes, we can then use the following commands on `fusion01` to mount the folders in which the IMG files are stored:

```
/usr/lib/gvfs/gvfs-fuse-daemon /data/ea310/servers
gvfs-mount sftp://fusion02/data/ea310
gvfs-mount sftp://fusion03/data/ea310
gvfs-mount sftp://fusion04/data/ea310
```

In this example, the first command says to the operating system (here we are working on Ubuntu) that every device mounted with the gvfs-mount command (ssh protocol) has to be mounted on the local disk: `/data/ea310/servers`. Otherwise the devices are only mounted in the file manager (named `nautilus` in our case) and are not accessible in command line or with our software. Once these commands are run, the other computers are mounted on the client machine and the IMG files can be accessed as every other file (and mounted as explained before).

What has been written before can be illustrate by the figure 4.4:

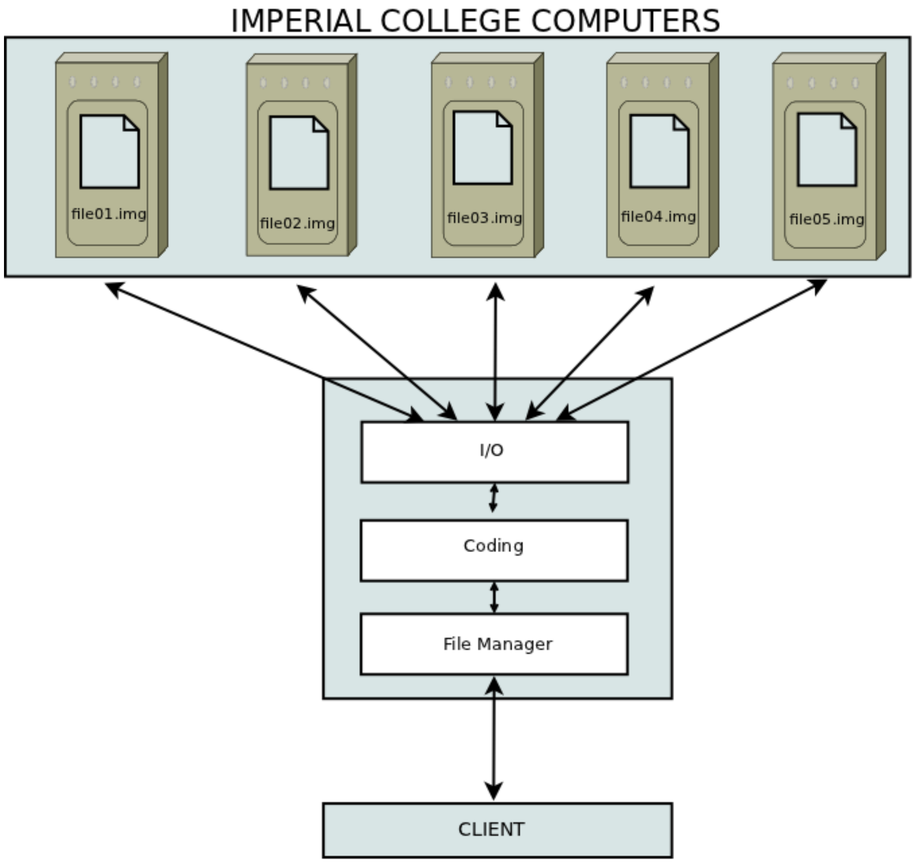


Figure 4.4: Data Distribution on Imperial College machines

#### 4.1.3.2 PlanetLab

PlanetLab is an open platform for developing, deploying and accessing planetary-scale services. It supports the development of new network services [9]. In this Project, PlanetLab was mostly used as a storage solution over the internet.

The way to use it was to install an iSCSI program on both the nodes (PlanetLab servers) and on the client. iSCSI is an Internet Protocol based storage networking standard for linking data storage facilities [15]. By carrying SCSI commands over IP networks, iSCSI is used to facilitate data transfers over intranets and to manage storage over long distances.

Once installed, the PlanetLab nodes are seen as Hard Drives by the client and we just have to use them as we would have done with normal Hard Drives (the IMG files are created on the nodes and are directly seen by the client as a device). The figure 4.5 illustrates this point:

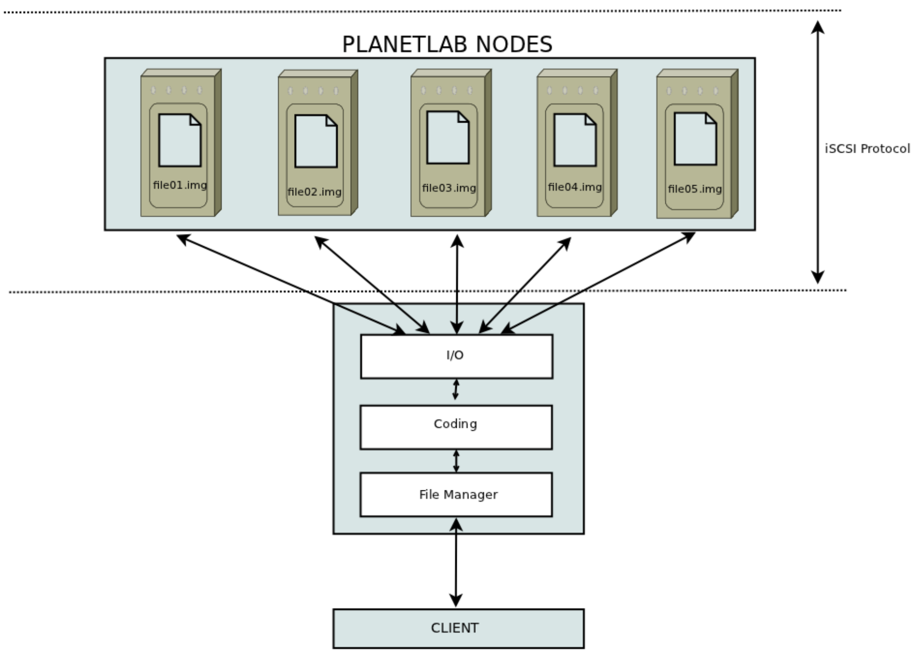


Figure 4.5: Data Distribution on PlanetLab nodes

Let's now see the procedure to install iSCSI on both the nodes (targets) and the client (initiator). First we need to install the corresponding package on every target and on the client: `open-iscsi-utils` [1] for the initiator and `tgt` [32] for the targets.

Once it is done, we need to configure the targets. Let's take the example of a target that has the following IP: 195.37.16.121. Note that the same has to be done for every other target node. We then need to run the following commands [32]:

```
./tgtd
./tgtadm --lld iscsi --op new --mode target --tid 1
              -T iqn.2011-07.com.example:storage.disk0
./tgtadm --lld iscsi --op new --mode logicalunit --tid 1
              --lun 1 -b /home/imperialple_codestore/file00.img
./tgtadm --lld iscsi --op bind --mode target --tid 1 -I ALL
```

The first line of this example runs the iSCSI daemon in the background. We then add a new device controller (which will control what comes in and out of the node via iscsi protocol) named `iqn.2011-07.com.example:storage.disk0`. In the third line we add a new device to this controller which is the IMG file that we created in the same way than before. The last line is just here to tell the iscsi daemon that any client can connect and bind to the controller (and the target device).

We now need to configure the initiator to connect to the target and mount the device in order to be usable by the software. Note that this operation needs to be done for every node. Let's have a look at the following commands [1]:

```
iscsiadm -m discovery -t sendtargets -p 195.37.16.121
iscsiadm -m node -T iqn.2011-07.com.example:storage.disk0
              -p 195.37.16.121 -l
```

The first line is here to tell the iSCSI protocol to connect to the node that has the following IP: 195.37.16.121 and to discover any controller present on it. It will return the name of every controller including `iqn.2011-07.com.example:storage.disk0` which is the one that we created before and we can then use this name to connect. That is what does the second line: it logs into the controller device of the target machine.

One of the advantage of this method is that the target is directly seen as a device by the client and has for example the name `/dev/sdc`. In this case, we don't need to mount it with a `/dev/loop` device as it was the case with the IMG file.

## 4.2 Layers

Now that we have given an overview of the system along with how to use it, we can begin to explain in more details the underlying architecture of the software which is composed of 3 layers that we are going to detail here. As explained before the three layers are the file managing layer, the coding layer and the storage layer. The name of these layers are similar to [11] but does not achieve exactly the same things since their implementation has to deal with much more details namely due to different implementations of RAID. For more information about this, you can refer to [11]. The first one is more technical when the other ones have more to do with algorithms that is why we will give more implementation details for the first one when we will take more time to describe the algorithms for the other layers.

### 4.2.1 File Managing Layer

This class is responsible for the management of any operation within the user folder. It can be any operation such as file creation, file copy, file deletion, file edition, linking, unlinking. As told before, the goal of this layer is to make the user interacting with his folder as with any other folders. That is what motivated the choice for the utilisation of FUSE.

FUSE (Filesystem in User Space) is a free software that provides an API for C++ [30]. It allows to create a Virtual File System without having to modify linux sources or the kernel. Note that a common file system saves data and retrieve it from a disk but virtual file systems do not store the data: they act like a view of an existing device. That is this functionality that we use in our project.

Let's now talk about the implementing part. FUSE provides a library for C++ that enables the developers to create their own virtual file system: that is what has been used to create the file manager layer. We first need to load a set of files:

```
#include <fuse.h>
#include <errno.h>
#include <fcntl.h>
```

The first one contains the definitions for basic functions needed to implement a file system, errno.h contains definitions regarding error numbers and the last one contains definitions of file options.

Once these files have been loaded, it provides every function we need to implement a file system. What we have to do is to modify them in order to make our system working the way we want (which involves a lot of change because of the fact that we code every byte of information). In order not to transform this report in a list of functions, we will only describe here the functions that have a huge importance for fuse and the ones that have been strongly modified to suit our requirements. Note that some of the following are inspired from the `hello_world` example provided on the wiki of fuse project [31].

- **init() function**

  This function is called when the file system is launched. It initializes the system: it will clear the mountdir folder if necessary and performs any action needed with the kernel. In our particular case, we will use it to return the properties of the system which thanks to fuse will be available to all the file operations. Note that here the properties of the system are represented under the form of a data structure `mbr_para` that contains belong others the rootdir and the mountdir path, the number of available disks, the number of disks from which the system should be able to reconstruct a file, the size of each disk, the status of each disk (whether they have failed or not), the fact that we use the cache or not, and other parameters needed for the practical of the implementation but not to explain how our software works.

- **destroy() function**

  The destroy function is called when the file system is closed, it is here to delete everything that was in the folder (but not on the disks, so when we remount the folder the data are still there). Here, this function will be strongly modified. Indeed, if we choose to use the cache, this function will be responsible for flushing any information that where stored in the cache to the physical disk and in any case, it is responsible for storing the spaces freed by the `unlink()` function in an annex file `met` that will be opened at the beginning of each utilisation. As these questions are more related to the cache and the storage layers, we will explain it in the corresponding section.

- **getattr() function**

  This function returns metadata concerning a file specified by its path in a special structure. In our implementation, it is very slightly modified since we only need to retrieve the path of the file in order to send it to the input/output layer and more particularly the `Read_Met` function that is in charge of reading metada information from cache.

- **readdir() function**

  Next very important function is used to read directory contents. Because our implementation is very simple, we will be interested only in `path`, `buffer` and `filler` arguments passed to it. `path` is the path to the directory from which we will have to read our contents, `buffer` will hold the contents, and `filler` is used to add contents to directory. Here the implementation comes directly from the hello_world example we were talking before.

- **open() function**

  Here this function is not modified by our system since there is no coding needed to open a file. By opening a file, fuse just means that we check if the user has enough privileges to read or write a file. Actually reading data is related to the next function.

- **read() function**

  In our project this is one of the most important function. Indeed, reading data is equivalent to retrieve the encoded information from the different nodes and decode it using an erasure code like explained in the overview section. The purpose of this function is then to handle reading request made by fuse. The point is that fuse can ask for any size of reading request: it can be 8 bits as well as 5 MegaBytes but as we explained in the part dealing with Reed

Solomon code or any other codes, erasure codes takes a fixed size input and returns a fixed size output. So the input of our coding layer must have a fixed size (even if it is not the size of the input in the way we have seen before).

That is what this function is dealing with: it first check that the amount of data required by fuse is less than the maximum size fixes by the user in the settings of the system and send it to the decoding function. Otherwise, if the size required is greater than the one set by the user, then the function cut it in several parts and sequentially sends it to the coding layer.

- **write() function**

  The problem is quite similar to the one before. Here the the function is equivalent to sending the message to the nodes so we fist have to encode it (which is done in the coding layer). The amount of data providing by fuse to be written on the servers can be very different so here again we have to use this function to check that is less than the input of the encoding function, otherwise we split it into several parts and send it sequentially to the coding layer.

- **unlink() function**

  The unlink operation is the fuse equivalent for deleting a file except that it does not delete it: it only deletes the link to the actual space. It does not reallocate the reusable space. We then need to change the function in order to keep a pointer on any space that has been used and then deleted. To do so, we define a new structure `free_block` which contains the disk on which the reusable space is available, the number of blocks (of fixed size) that are available and a pointer on the next available block. We use this structure as a linked list. When we initialise it, the first node has only the `NULL` pointer value and each time we add a new node, we redirect the last pointer to the new node. This way, we can keep a track on every space that has been freed.

  Note that when we want to reallocate one of the space that we have freed (which happens when there is no space left on the disk and that we need to come back to the freed space), we need to remove the space from the list and to do so we just redirect the link that came from the previous node to the one that comes after.

  You can also notice that when we reuse a space, it can only be done by the same disk on which the space has been freed.

- **fgetattr() function**

  This function is used when we need to retrieve the file attribute and can be used by fuse instead of the classic `getattr` function. That is why we had to change it in the same way that we did for `getattr`: use the storage layer.

- **main() function**

  Let's now talk about the main function. Even if it is not part of the fuse library, it is very useful for us because it is here that we retrieve all the information that fuse needs to know. In fact, before running the program, the user must fill a `settings` and a `status` file.

  In the first one, he must write the number of available disks along with their location and their size but also other information such as the fact that he wants to use the cache or not. In the second one, we put the status of every disk: 1 if we consider that he is down and 0 if he is still usable. We then use the `main` function to parse these files and give the `fuse_main` function (the one that starts the system and calls for the init function) the required arguments.

As these few examples suggest, the file managing layer is just a bridge between the actual files and the coding layer that we are going to detail.

### 4.2.2 Coding Layer

In the chapter dealing with coding theory, we saw how to add redundancy to a message in order to make it recoverable if some symbols were erased. In the chapter dealing with the Minimum Bandwidth Exact Regenerating code, we have seen how to distribute the symbols of an encoded message on several nodes.

All of these are very beautiful theoretical work but when it comes to implementation, it becomes much more challenging and we need to define how we are going to use these theories. In the following, we will first see how we distribute the data on several nodes in our practical implementation based on the Minimum Bandwidth Exact Regenerating code and in the two other points we will see how we code and decode a message in a much more efficient way (based on an implementation approach).

#### 4.2.2.1 Data Architecture

In the following $n$ will represent the number of available disks and $k$ the number of disks needed to rebuild a file.

As just said before, we have to organize data into fixed size block. The incoming fixe sized blocks will be considered as original blocks. Once we have the $B$ original blocks as required and explained in chapter 3 ( $B = \frac{kd-k(k-1)}{2}$) we can construct $\theta = \frac{n(n-1)}{2}$ blocks thanks to a Reed-Solomon code. These $\theta$ blocks are made from the $B$ original ones and $\theta - B$ other blocks that we will call the encoded blocks. In our choice of algorithm, we also need to introduce blocks that are just copies of the $\theta$ ones in order to enable the repairing.

In the following we will call the whole of these blocks the aggregated block. An aggregated block is then formed by the $B$ original blocks, the $\theta - B$ encoded blocks and the $\theta$ copies.

You can notice that at the beginning we only have 1 original block which is less than the $B$ needed to do the encoding we then pad it with empty blocks to do the encoding and have an aggregated block. Then, every time we have a new block we replace the first empty block by it and we redo the encoding in order to recompute the same aggregated block. We do that until we have the $B$ original blocks. Once it is the case we begin a new aggregated block and we keep doing the same thing.

Using the example from chapter 3 with $n = 5$ and $k = 3$, we still have $B = 4$ and $\theta = 10$, a representation of the different nodes could look like the one of figure 4.6:

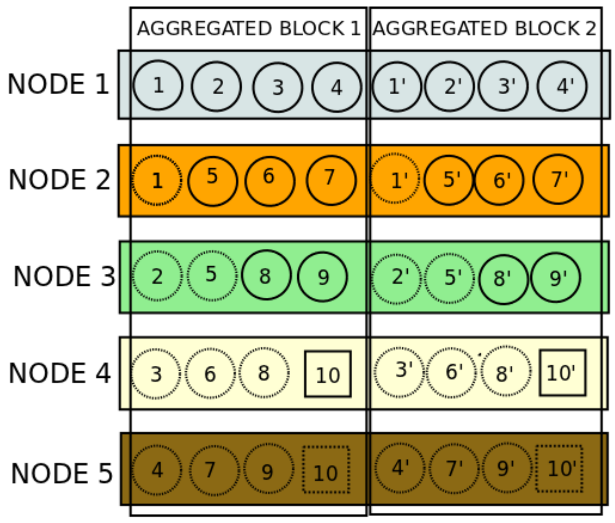


Figure 4.6: Data Distribution

On this figure, we have represented two aggregated blocks and for each of them, you can notice that the circle represent the original blocks, the dotted circle the copies of the original blocks, the square represent the encoded blocks, and the dotted squares represent the copies of the encoded blocks.

The main difference between this figure which represents our practical implementation and the one we saw in chapter 3 is that here we are dealing with blocks where in chapter 3 symbols were stored on the node. We could make correspond a symbol with a block and it would solve the problem but the drawback would be that we will write the symbols (and by doing so the blocks) one by one on the node which will be very slow.

The way we choose to get around this difficulty is that we directly code the blocks. One could say that it is not possible since we choose a symbol to be 1 byte and by doing so the size of the field is limited to $2^8 = 256$ elements and so a block should at most contains 256 symbols (bytes in our case). But when we say encoding a block we do not really encode a block: we mean encoding every symbol (here a symbol is a byte) from every block.

Let's take an example. We define our system such as follows: $B = 4$ and $\theta = 10$ (as in our example) and we choose that the size of our blocks will be 3 bytes. Let's imagine that we have $B$ blocks then we have $B$ first bytes, $B$ second bytes and $B$ third bytes. Thanks to the Reed Solomon we can encode it into $\theta$ first bytes, $\theta$ second bytes and $\theta$ third bytes which will be our $\theta$ blocks. We can then distribute it according to the representation at the top of this page. So in any case the encoding part of the Reed Solomon code will receive $B$ symbols and return $\theta$ symbols that will then be transformed into $\theta$ blocks.

All this work is done by the encoding function of the coding layer named `encode`. It takes as an input a block of size define by the user. Then it is in charge of finding on which nodes and which blocks it has to store the original blocks, the encoded blocks and the copies according to the distribution we have just seen. We will see how to implement this distribution in the next subsection. Then it stores the original blocks and the copies of the original blocks on the corresponding disk and on the corresponding block from that disk using the write function define in the input/output layer. Note that contrary to chapter 3 where every copy made on the different disks are copies of encoded symbols (because all symbols were encoded), here the first $B$ symbols of the encoded words are the same than the original message that is why we can copy them directly before using the encoding function. This is made feasible thanks to the systematic from of the generator matrix (as we are going to see in the Practical Encoding subsection).

Finally, this function compute the encoded block thanks to the generator matrix created as we will see in the Practical Encoding subsection before storing it and its copy on the good disk an the good block using the same write function.

#### 4.2.2.2 Data Distribution

In the previous subsection, we talked a lot about storing a block on a disk according to the distribution matrix $V$ described in the part dealing with Minimum Bandwidth Exact Regenerating Code (that is for example with this matrix that we got the previous figure). We are now going to explain how we implement this matrix or rather how we find the corresponding disk number and block number of a block.

Let's suppose that fuse wants us to store a block that comes in $i$th position. It will not be the $i$th on each of the disks that we have to manage (we will call this position $i$ the absolute position because it is the real position of a block in the stream of the blocks store by fuse when the position on a particular disk will be called the relative position since it depends on the number of the disk). That is why we need in the encoding and decoding function:

- *one function to find the relative position of an original block based on its disk and on its block absolute number.*

  To do so we have to notice that a block cannot belong to a disk if its position on the aggregated block is less than the disk number (in this case we return $-1$). Let's assume that we initialize a counter by making it equal to the size of an aggregated block. To compute the relative position, we just have to notice by looking at the matrix that every time we increment by 1 the number of a disk the position on this disk decrease from is equal to the position on the previous disk plus the counter (reduced from 1).

- *one function to find the relative position of a copy based on its disk and on its block absolute number.*

  To do so we have to notice that a block cannot belong to a disk if its position on the aggregated block is greater than the disk number (in this case we return $-1$). Let's assume that we have a second counter initialize to the number of disk minus 1. To compute the relative position, we just have to notice by looking at the matrix that every time we increment by 1 the disk number then the relative position is equal to the one of the original block plus the counter decrease from 1.

- *one function to find the number of a disk for a block based on its relative position.*

  To do so we just have to notice that is function is the opposite compared to the first one, we then just have to reverse every operation to get the result.

- *one function to find the absolute position of a block based on its disk number and its relative position.*

  To do so we, still by looking at the matrix it is easy that the absolute position of a block based on a disk is the same that at the previous disk plus the disk number decreased from the size of an aggregated block until we arrive to the first disk that will give the absolute position.

- *one function to find the disk number for a copy based on its relative position.*

  For this function, we first need to call the function that gives us the number of a disk for a block based on its relative position. Then knowing that we just have to follow the same reasoning than the previous function.

- *one function to find the absolute position for a copy based on its disk number and its relative position.*

  This function is the easiest one since the absolute positive of a duplicated block is equal to the disk number. We then just have to return it.

With these 6 functions encoding and decoding can store the blocks according the matrix $V$ define in the Minimum Bandwidth Exact Regenerating code from chapter 3.

#### 4.2.2.3 Practical Encoding

In the chapter dealing with the encoding algorithm of the Reed Solomon code, we only explained how we theoretically encode a word (made of symbols). Obviously it was needed for the understanding of the project but it is not really feasible in a real implementation. That is why we are now going to explain how we can encode a word in practice: this is this method that will be used in our implementation.

In the second chapter, we explained that an original word $m$ could be transformed into a coded word $c$ with a simple matrix multiplication: $c = Gm$. This is this technic that we are going to use. We want to use a matrix under its systematic form since the computation will be much faster and that it is always very useful to keep an identity matrix at the beginning of the matrix since that way the fist $B$ symbols of the encoded word will be the same that the original word as we will see in one of the following figure. Now let's remind that the generator matrix for a Reed-Solomon code has the the following form [16]:

$$G = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ \alpha^{1^1} & \alpha^{2^1} & \alpha^{3^1} & \dots & \alpha^{B^1} \\ \alpha^{1^2} & \alpha^{2^2} & \alpha^{3^2} & \dots & \alpha^{B^2} \\ \vdots & \vdots & \vdots & & \vdots \\ \alpha^{1^{\theta-1}} & \alpha^{2^{\theta-1}} & \alpha^{3^{\theta-1}} & \dots & \alpha^{B^{\theta-1}} \end{pmatrix}$$

In order to compute this matrix we first need to compute the multiplication over the the finite field that contains $2^8$ elements. They are implemented exactly in the same way that what we have seen in the second chapter. The only modification made to it is the fact that we store every multiplication over the finite field result in a table in order to make the computation more fast and not having to compute several times the same result.

Once this is done we need to compute the matrix we have just talked about. This is quite easy since we just have to compute the second line and multiply each element by itself to get the other lines. That is exactly what is done in the `codmat` function.

The thing is that this matrix is not very useful in this actual state because we need to keep the identity matrix at the beginning of the matrix in order to make appear the original message as the first $B$ symbols of the encoded word: we need to transform this matrix into its systematic which is done by the `systematic` function. This function is nothing more than a Gaussian Reduction except that the operations take place over the finite field with $2^8$ elements.

The principle is to go through each columns $i$, find a nonzero elements $G(j,i)$ in this column and swap the row in order to have this element in position $G(i,i)$. Now, for each element in row $i$ that is not in column $i$, we need to make it zero. Suppose that this is column $j$, and the element at $G(i,j) = e$. Then we want to replace all of column $j$ with $(col - j + col - i * e)$ (where $col$ is the number of columns). We then desire to have row $B$ be all ones. To do that, we multiply every column $j$ by $\frac{1}{G(B,j)}$. Then we multiply every row $j$ by $\frac{1}{G(j,j)}$ to keep the identity matrix. Finally, we'd like the first column to be all ones. To do that, we multiply every row by the inverse of the first element. Once this is done, we have the following matrix:

$$G = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & * & * & \dots & * \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & * & * & \dots & * \end{pmatrix}$$

This $\theta * B$ matrix is very useful to decode the encoded message because the $B$ first lines are the identity matrix and in the remaining part, the first row and the first columns are all ones.

When we do the multiplication of a message $m$ by this matrix we obtain the same message $m$ at the beginning of the encoded message which is much more efficient than the original matrix $G$ since we only need to copy the first $B$ symbols and only after do some computation to find the encoded symbols. You can see an illustration here:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & * & * & \dots & * \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & * & * & \dots & * \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \end{pmatrix} = \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \\ c_1 \\ c_2 \\ \vdots \\ c_{\theta-B} \end{pmatrix}$$

Finally, the `reducedcodmat` returns only the last $\theta - B$ rows which allows the encoding function to just have to do the multiplication of the last rows with the original to get all the $\theta - B$ encoded block. Note that the first $B$ can directly be copied without using the matrix.

**Toy Example**

Let's assume that we have $n = 4$ nodes and that we want to retrieve the file thanks to $k = 3$ nodes, we can then take the following systematic generator matrix over the finite field $\mathbb{F}_8$:

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Then the original word $M = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix}$ will become:

$$\begin{aligned} GM &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} \\ &= \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_1 + m_2 + m_3 \end{pmatrix} \end{aligned}$$

#### 4.2.2.4 Practical Decoding

Here again is a completely different approach to decode an encoded word than seen in the theoretical part (but contrary to this part, it will only deal with erasures and not errors).

First, let's come back to the following equation: $Gm = c$. We then have the following system:

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & * & * & \dots & * \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & * & * & \dots & * \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \end{pmatrix} = \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ m_B \\ c_1 \\ c_2 \\ \vdots \\ c_{\theta-B} \end{pmatrix}$$

Now, as explained when we were dealing with the rebuilding of a file using Minimum Bandwidth Exact Regenerating code, $n - k$ nodes fail (so we still have $k$ remaining nodes) we can still have access to $B$ symbols which means that $\theta - B$ symbols have been erased.

We can reflect these erasures in the system gave before [13]: we just have to delete every line of the system that corresponds to the erased symbols of the encoded message. We then get the following system $G'm = c'$ where $G'$ and $c'$ represent the matrix $G$ and the encoded word $c$ without the erasure symbols. Note that $G'$ becomes a square $B * B$ matrix and as it is a subset of a linear permutation of a Vandermonde Matrix it is non singular which means that we can compute its inverse. We then just have to multiply the encoded word by this new matrix to get the original word:

$$m = c'G'^{-1}$$

Let's now talk about the implementation. The function named `decomat` in charge of deleting any rows that correspond to an erased symbol and the function named `inv` is in charge of the computation of the inverse of this matrix using a Gaussian Elimination. As this last function is not really trivial we are going to detail it a bit more.

We first need to convert the matrix into an upper triangular matrix. To do so, we apply the following algorithm: for every column $i$ find a non zero element and swap two lines to put it in $(i, i)$ position, then we multiply the whole row by the inverse of this element. Now for each $j > i$, we add $M(j, i) * M(i, )$ to $M(j, j)$ (it will put to zero every element other than (i,i) that is on the same row). Now that we have an upper triangular matrix, we just need to take each line and multiply it by the inverse of its first non zero element to get the inverse matrix.

Once this is done the `decode` just has to do the multiplication of this matrix with encoded word to get the original word (and then the corresponding block).

**Toy Example**

Let's assume that we have $n = 4$ nodes and that we want to retrieve the file thanks to $k = 3$ nodes, we can then take the previous systematic generator matrix G. We also make the assumption that we want to retrieve the file thanks to node 1, 2 and 4.

The following system:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} m_1 \\ m_2 \\ m_3 \\ m_1 + m_2 + m_3 \end{pmatrix}$$

then becomes:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} m_1 \\ m_2 \\ m_1 + m_2 + m_3 \end{pmatrix}$$

We then need to invert the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Which gives us:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 7 & 7 & 7 \end{pmatrix}$$

And finally, we compute the following multiplication to get the result:

$$\begin{aligned} M &= \begin{pmatrix} m_1 & m_2 & m_1 + m_2 + m_3 \end{pmatrix} \begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} m_1 & m_2 & m_3 \end{pmatrix} \end{aligned}$$

### 4.2.3 Storage Layer

Now that we have talked about the file manager and the coding layers, we still have to understand how the storage layer works.

At the beginning of the program we can make a choice between two options: the first one is that we choose not to use the cache, in this case there is nothing to explain since the program will just write on the disk as it always does. The second is obviously the case when we choose to use the cache and here there is much more to explain and that is the goal of this subsection.

There are 2 main possible operations in our implementation of the cache layer: reading a block and writing a block.

First let's have a look at the description of our cache which comes from [34] and in which was found the write-policy that we are going to explain.

A cache is made up of entries. Each entry contains data that are the copies of the data that are stored on the real storage device. Each entry also has a tag, which specifies the identity of the data on the storing of which the entry is a copy. Here the tag is simply the position of the data on the disk.

When the coding needs to access data stored on the disk, it first checks the cache. If an entry can be found with a tag that corresponds to the one of the disk, then we return the data in the cache rather than the one on the disk. This situation is known as a cache hit. In the opposite case, when we check the cache and that we cannot find the corresponding data, it is a cache miss. In this case, we fetch the data from the disk and copy it into the cache for the next access.

When a system writes data to the cache, it must at some point write these data to the disk as well simply because the size of the cache cannot be as big as the one of the disk. The timing of this write is controlled by the write policy we are going to explain in the next paragraph.

The write policy we have chosen is the write-back policy because it is the most commonly used. In a write-back cache, writes are not immediately stored on the disk. Instead, the cache tracks which of its locations have been written over and marks these locations as dirty. The data in these locations are written back to the disk when those data have to be deleted from the cache.

To sum up, when we want to read data from the cache and put it in a buffer, we first verify that it exists on the cache, if it does we directly return the value to the buffer. Otherwise, we have to allocate space for the new data on the cache and we distinguish 2 cases: if the location is not dirty we fetch the data from the disk to the cache and mark it as non-dirty and if the location is dirty (not already stored on the disk), we store what the location contained on the disk and then we fetch the new data in order to come back in the first case.

If we want to write data from a buffer to the cache, we first verify that it exists on the cache, if it does we directly overwrite it and we mark it as dirty. If it does not exist, we have to reallocate spaces for the new data on the cache and again we still distinguish two cases: if the location is not dirty we write the data from the buffer to the cache and mark it as dirty and if the location is dirty (not already stored on the disk), we store the previous data that the location contained on the disk and then we write the new data from the buffer to the cache in order to come back in the first case.

You can find everything that has been explained before on the figure 4.7:

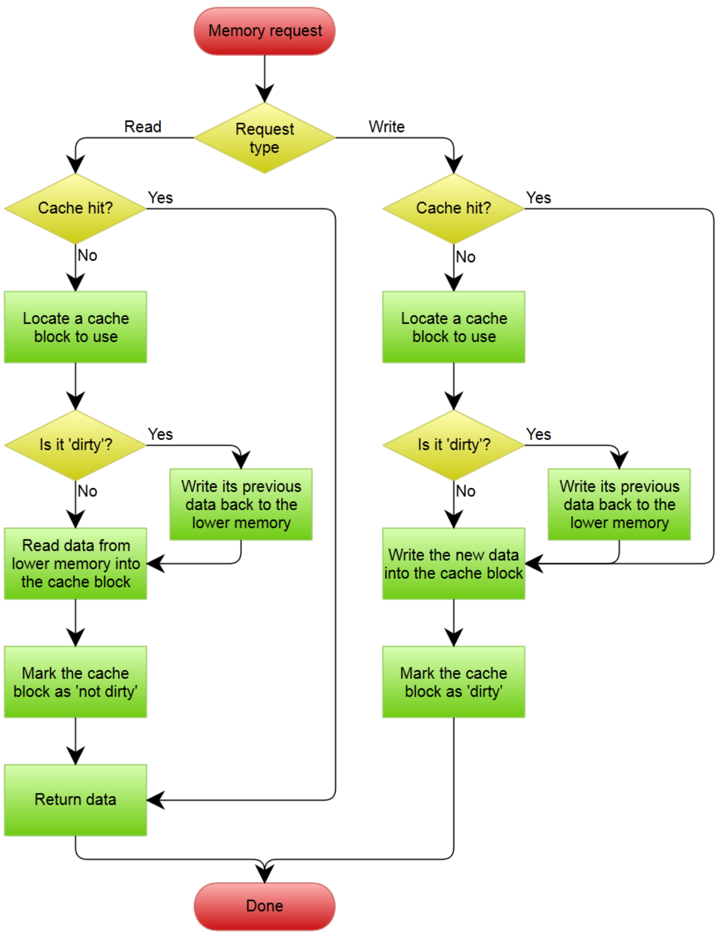

Memory request
Request type
Read
Write
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Return data
Cache hit?
Yes
No
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Write the new data into the cache block
Mark the cache block as 'dirty'
Done

Figure 4.7: Cache Layer Representation

Our implementation is the exact application of this method with an entry per disk.

### 4.2.4 Repairing Function

This part of our implementation is the easiest one. Indeed, the only thing we have to do is using the following algorithm: for every block of the disk that we want to rebuild, we first need to know if this block is a copy of an original or an encoded block. If it is an original or original encoded block we then need to find the block number and the disk of the corresponding copy thanks the function described in the coding layer (data distribution subsection). If it is a copy of an original or of an encoded block we then find the the block number and the number of the disk of the corresponding original block.

Then in both case, we just need to copy what is contained on the space corresponding to the number of the disk and the number of the block to the block of the new disk.

## 4.3 Issues

The solutions chosen in the previous sections were not the most obvious ones but as it is a student project there are some limitations regarding the available resources including the fact that the `sudo` command is not available and that has some consequences. This section is here to quickly explain the choices made that were not necessarily obvious but that were needed according to this particular restriction.

### 4.3.1 Use of sudo on the laboratories computers

Here the problem is that even if fuse operates in user folder and does not need access to the kernel, we still need the `sudo` command to run it. With the CSG, we have looked for different solutions including using other APIs than fuse but it appeared that it was the only free project that allows to manage file as if he was using a real file manager. As the main goal of this project was to study the feasibility of an Minimum Bandwidth Exact Regenerating code in a real environment, the use of fuse was compulsory.

The option we chose was then to use a Virtual Machine on which the use of `sudo` will be possible on one of the laboratories machine. But there are consequences to this choice. If you have a look back to section 4.1.3.1, you can notice that we have to find the IMG files to mount them. The problem is that now we have to mount this file onto the virtual machine and not on the real machine. So we still proceed as in section 4.1.3.1 using ssh protocol and once our files appear on our real computer (the one on which is stored the client virtual machine), we can mount it using KVM (Kernel Based Virtual Machine [27]) on which we can install Virtual Machine which has an option to mount IMG file directly as disk.

Using the command `fdisk -l` on the virtual machine will show this disk mounted as `/dev/sdb` and we just have to use it as described before.

### 4.3.2 Kernel problems using PlanetLab nodes

First thing to know about PlanetLab is that it runs a modified version of Fedora distribution on which iSCSI is not installed by default. So we tried to install it using `yum` (the fedora package manager) but the only available package was `iscsitarget`. Once installed and running an error always appeared regarding the accessibility of a kernel module. With the help from Thom Haddow from Imperial Department of Computing, we finally discovered that it was because `iscsitarget` wanted to use inaccessible function of the PlanetLab which obviously are not editable.

We then looked for a package that was able to operate without the kernel and finally found `tgtadm` that had to be manually installed in order not using the kernel. That was the solution that solved one of the biggest problem of this project.

# Chapter 5

# Experiments and Results

As said before, the goal of this project is to study the usability of a system such as the Minimum Bandwidth Exact Regenerating code in real conditions. That is the aim of this chapter: compare the performance of our system with what could be expected from a real distributed storage system.

For that, we first need to present the context in which took place the experiments. Then, we will test the impact of the different hardware components on the system.

## 5.1 Experiments Limitations

In this chapter, we will always have to keep in mind that we are dealing with experiments which implies some imperfections (namely on the hardware components) and on the top of that there are also limitations due to our special environment (computer laboratories and use of PlanetLab)

### 5.1.1 Environment Limitations

The goal of the following sections will be to measure the impact of hardware components on the system. But these performances has to be read carefully since no environment is perfect. In this subsection we are going to explain for every component what will be the limitations due to our environment.

#### 5.1.1.1 CPU

In the Impact of the Hardware components section, the first component to be tested will be the CPU. Obviously, we need to control the frequency in order to do some tests that we will detail in the appropriate section.

To do so, we need to use a software called `cpulimit` [14] that limits the percentage of CPU that a specific program can use. The problem is that we need the `sudo` command in order to run this tool. So here the solution was rather easy since it is the same problem than in the previous chapter where we need `sudo` command to run FUSE. So the solution is the same: use a Virtual Machine. So in the next experiments dealing with CPU, we always have to keep in mind that we are running a Virtual Machine. Note that it does not create any new problem since when we install the Virtual Machine you can configure the CPU parameter so we know what will be the CPU frequency.

#### 5.1.1.2 Bandwidth

The second component that will be tested will be the impact of a bandwidth limitation (uploading or downloading) on the three main functions of the system: uploading a file, downloading it and finally the repairing of a node. So here again, we need to control the bandwidth. Here we have several available tools to limit the bandwidth: `trickle` or `wondershaper`. The first one limits the bandwidth of a single program when the second one limits the bandwidth of the whole operating system.

Here, we are going to be confronted to a lot of different problem. First, we have to decide on which part of our system we want to put a bandwidth limitation: the nodes (store on PlanetLab) or the client (store on a Imperial College computer). First idea was to limit the bandwidth on the PlanetLab nodes because testing the impact of the bandwidth on the node repairing (data would not have to transit by the client in order to go from a node to another) but it appears that this solution was not feasible. Indeed, after talking with the PlanetLab nodes support, it appears that any actions that involve a change on the ethernet card of a PlanetLab node is forbidden. So we had to forget about this option.

The only remaining option is to limit the uploading and downloading bandwidth of the client (note that we are still working on a Virtual Machine installed on a real machine in the laboratories). Here the problem is quite different but the result is the same. If we want to change the configuration of the Virtual Machine network we directly need to have access to parameter of the ethernet card of the real machine so it is also impossible.

So finally, the only option that was feasible is to use a laptop and connect it on the WiFi Imperial College network. Here there are no constraints except that we cannot exceed 5 GigaBytes of downloading per day and that the file use to test are, as we will later, quite big so it limits the number of experiments per day but it is not crippling.

#### 5.1.1.3 Hard Drives

The last test on the hardware components will deal with the Hard Drives. To do so, we will do two kind of tests: first the difference between a common hard drive and with a SSD flash hard drive. Second will be the impact of the prioritisation of the process on the disk using `ionice` [24].

In both case, the limitations of these experiments will be the same as for the CPU limitations: we do not have access to the `sudo` command and then need to use a Virtual Machine

### 5.1.2 Hardware Limitations

This section is just here to remind that no components are perfect and so any experiments are subject to imperfections. By this, we mean that performances are not steady: during a certain amount of time the transfer speed of a disk can for example decrease and increase. The same holds for the CPU and the Bandwidth.

On the top of that, we can add the fact that we are not alone on the system. Regarding for example the CPU, the Operating System can perform background tasks and so need a percentage of the CPU frequency (note that we do not have to worry about the CPU performance on the nodes since their goals is only to store data, they do not have to do any significant computations).

Regarding the Bandwidth, both on PlanetLab and on the college network, we are not alone and we need to take into account that other might use whether or not a huge amount of bandwidth.

At last, the disks are also a problem since the Operating System can use it and more particularly on Planet, other users that are on the nodes can use the same disk and by doing so changing the data transfer.

In order to bypass these imperfections, the solution we found is to do our experiments using rather big files (here we will transfer an iso file of an Ubuntu distribution that is 643 MegaBytes large).

Using this principle, the experiments will be quite long and by doing so the perturbations due to the other users or the operating system will be limited since the result will be the mean on the very long time. Of course every experiment is made several times (between 5 and 10) and we take the mean of the values we obtained.

The drawback of this solution is that the number of experiments that we can do per day will be limited by the 5 GigaBytes provided by the Imperial College network.

## 5.2 Impact of the hardware components

In this section we are going to describe the different experiments that have been done on the hardware components but also on the cache system that we have implemented and trying to analyse them.

Note that in all the hardware section, we will take $k = 3$ and $n = 7$ to compare the result and the size of the file used is 643 MB which once we have had the redundancy gives us a total amount of data of 900.2 MB.

Also in both the hardware and parameters section, the size of the input file will be 643 MB for the uploading and downloading operations. When for the repairing section, we will deal with disk of 1 GB and we repair the integrality of this disk (not just the used space) since it is in the specification of a distributed storage system.

### 5.2.1 CPU

Let's begin with the test regarding the CPU impact. In the requirement of our system we said that the nodes were only storage nodes and so they are not supposed to do any computations so we only need to test the impact of a CPU limitation on the client.

To do so, the nodes will then be stored on the Laboratories computers over the ethernet network of the imperial college as explained in section 4.1.3.1 (we want to get rid of the limitations of the bandwidth so we cannot use the PlanetLab for this experiment).

So as explained before we are going to use `cpulimit` on the client virtual machine. To do so, we first run the software as usual:

```
./mbr -f <rootdir> <mountdir>
```

Then we get its `PID` by using the following command:

```
ps aus | grep mbr
```

Let's assume that the PID is 17730 then we can limit the utilisation of the CPU by the software by 50% using the following command:

```
cpulimit -p 17730 -l 50
```

By doing so, every process that uses the PID 17730 will be limit to 50% of the CPU when using it. So the different experiments will consist in changing the CPU percentage available for our system and see how it evolves.

The following figures show the impact of the CPU usage on the uploading of a file on the system, the downloading of a file from a system and a repairing operation. Note that the configuration of the client machine is as follows: 512 MB of RAM, IDE disk (7500 rpm) and intel Core i5 @ 3.20GHz CPU.

Note that the RAM is voluntary low since we do not want that the operating system uses it to cache the data.

#### 5.2.1.1 Uploading

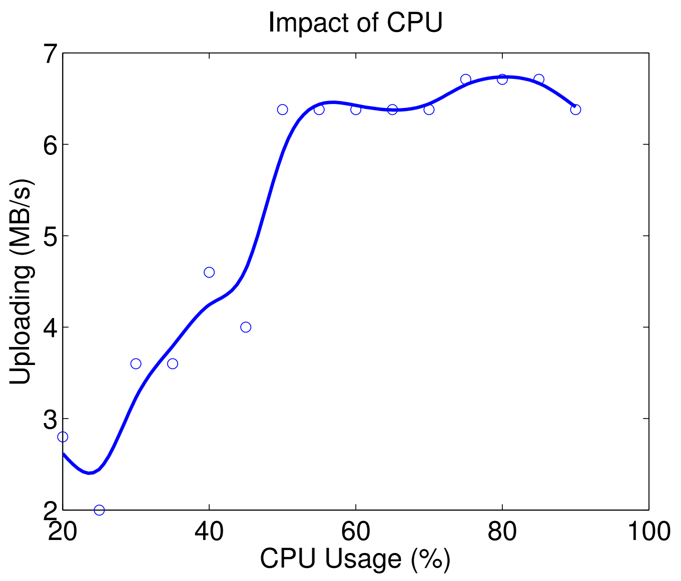


Figure 5.1: Impact of the CPU on the uploading speed

On the graph from figure 5.1 we can understand that the CPU is very important when its frequency is low and the more the frequency is high and the less it has an impact.

Now that we have these results it is quite understandable that in the process of uploading a file the coding part (using the generator matrix for example) is the part that is the most CPU consuming so when we limit this parameter we limit the speed of the encoding which limits then the data transfer and for example the Gaussian Elimination takes more and more time to be done.

However after 40% of the CPU usage the speed is quite steady and this is quite normal since the encoding process is a sequential process so the only way to make it even faster would be to parallelize it by for example using several encoding bocks.

Let's now run the same experiment (limiting the percentage of CPU available for our program) but when trying to download a node from the nodes to the client machine.

#### 5.2.1.2 Downloading

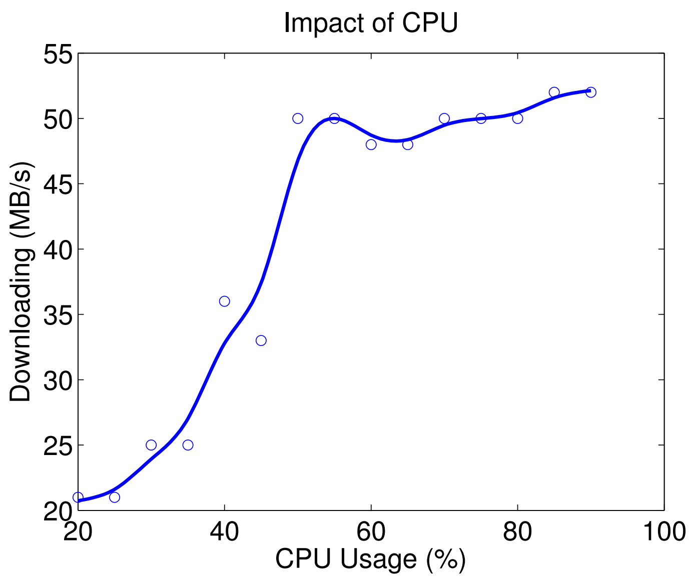


Figure 5.2: Impact of the CPU on the downloading speed

The problem in figure 5.2 is quite the same than before. In the downloading operation of a file, the decoding process becomes very slow when we decrease the CPU frequency.

Furthermore, the inversion of the Vandermonde matrix takes a lot of time too. It is normal since the computation over the finite field are more complex than operations with normal integers.

Here again we can notice that after a while the speed does not increase any more which is due to the fact that the decoding is sequential. Compared to the last graph the limit speed comes a bit later which proves the fact that the decoding is more complicated and so more time consuming than the encoding.

Now that we have seen the impact of the CPU on the downloading and uploading speed, we can try to run the same kind of experiment with the repairing function.

#### 5.2.1.3 Repairing

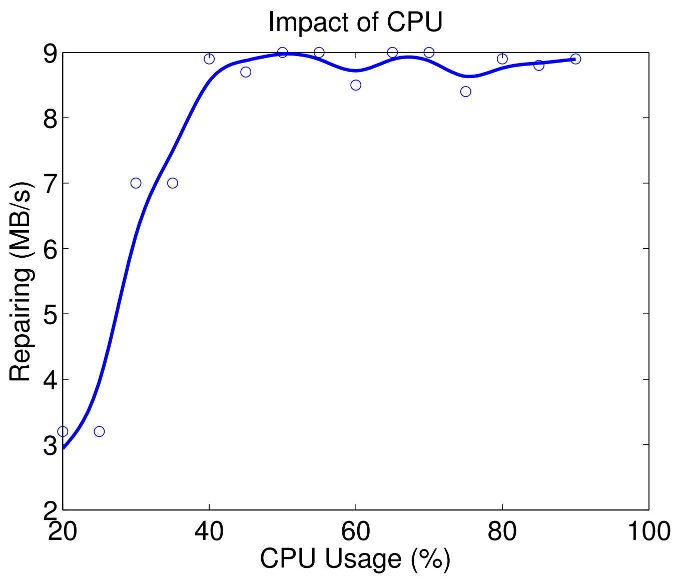


Figure 5.3: Impact of the CPU on the repairing speed

The result from 5.3 was much more surprising than the two previous one. At first sight, this operation was really heavy for the disk but not for the CPU since it is essentially copy from several nodes to another one does not include encoding or decoding.

But according to this graph we can see that the CPU does have an impact on the repairing when the percentage of available CPU is low. The answer can be found in the distribution of the data on different nodes. Indeed for each block, the program need to find the position for several nodes so it has to go through the distribution graph a huge number of time which implies a huge number of `for` loop. This the part that is CPU consuming in this implementation (again only when a low percentage of the CPU is available but this could happen on less recent computers).

### 5.2.2 Bandwidth

In this section, we are going to limit the bandwidth of the client machine in order to see how the system will react. As we want to test the bandwidth in real conditions (not over an ethernet network), we will use PlanetLab nodes as illustrates in 4.1.3.2. As told in part 5.1.1.2, we first try to use `wondershaper` and `trickle` but the problem of these two software is that they use preloading which that every data transmitted over the network will be in a buffer and only after a certain amount of time it will be transferred. Doing it this way we have much less control and understanding of what happens.

That is why we chose to use the script found in [12] where the author explains how to create a system that respect a certain Quality of Service which means that they create a system such as if there are several users, the bandwidth will not be completely used by only of them. The script allows the administrator to put a particular downloading and uploading speed for every user. We will use this script in a different way since it will allow us to limit the bandwidth. In fact we will use it as if there was only one user on the system that can only use the bandwidth to a particular bandwidth.

Here is the script that we use and that has been found in [12] (details of what the script does are in the comments, we refer to [12] for more details):

```
#!/bin/bash
# Change the following values
DOWNLINK=800
UPLINK=220
DEV=eth0
# Clean input and output manager
tc qdisc del dev $DEV root 2> /dev/null > /dev/null
tc qdisc del dev $DEV ingress 2> /dev/null > /dev/null

########## Uplink #############

tc qdisc add dev $DEV root handle 1: cbq avpkt 1000 bandwidth 10mbit

# We create different priority classes: the default class
# Main class
tc class add dev $DEV parent 1: classid 1:1 cbq rate ${UPLINK}kbit \
allot 1500 prio 5 bounded isolated
#  Bulk Priority
tc class add dev $DEV parent 1:1 classid 1:10 cbq rate ${UPLINK}kbit \
allot 1600 prio 1 avpkt 1000
# bulk and default class 1:20 - gets slightly less traffic,
# and a lower priority:
tc class add dev $DEV parent 1:1 classid 1:20 cbq rate $[9*$UPLINK/10]kbit \
allot 1600 prio 2 avpkt 1000
# Both classes managed by sfq:
tc qdisc add dev $DEV parent 1:10 handle 10: sfq perturb 10
tc qdisc add dev $DEV parent 1:20 handle 20: sfq perturb 10

# Filters
# SSH and IP protcols are redirected to 1:10 priority class
# 1:10 :
tc filter add dev $DEV parent 1:0 protocol ip prio 10 u32 \
match ip tos 0x10 0xff flowid 1:10

tc filter add dev $DEV parent 1:0 protocol ip prio 11 u32 \
match ip protocol 1 0xff flowid 1:10
# Same goes for ACK packets
tc filter add dev $DEV parent 1: protocol ip prio 12 u32 \
match ip protocol 6 0xff \
match u8 0x05 0x0f at 0 \
match u16 0x0000 0xffc0 at 2 \
match u8 0x10 0xff at 33 \
flowid 1:10
# Everything else is considered as inactive so we redirect it to the 1:20 priority class
tc filter add dev $DEV parent 1: protocol ip prio 13 u32 \
match ip dst 0.0.0.0/0 flowid 1:20
```

```
########## Downlink #############

tc qdisc add dev $DEV handle ffff: ingress
# Rejects everything that is coming in too quickly
tc filter add dev $DEV parent ffff: protocol ip prio 50 u32 match i
```

We can use it by changing the parameters at the beginning of the script and then just running it. Note that it will affect the speed of the whole machine (not only the one of our software). The advantage of this script is that it deals much better with preloading and that at least we can understand what is happening.

Let's now really talk about the experiments. In a first time, we will limit the uploading speed and look what are the consequences on the uploading of a file, on the downloading of a file and on the repairing of a node over an internet network. We will then limit the downloading speed and see what happens for the same elements. Note that for the repairing, we choose to rebuild the failed node on the client machine: we then download from every node to the client machine.

#### 5.2.2.1 Uploading Limit

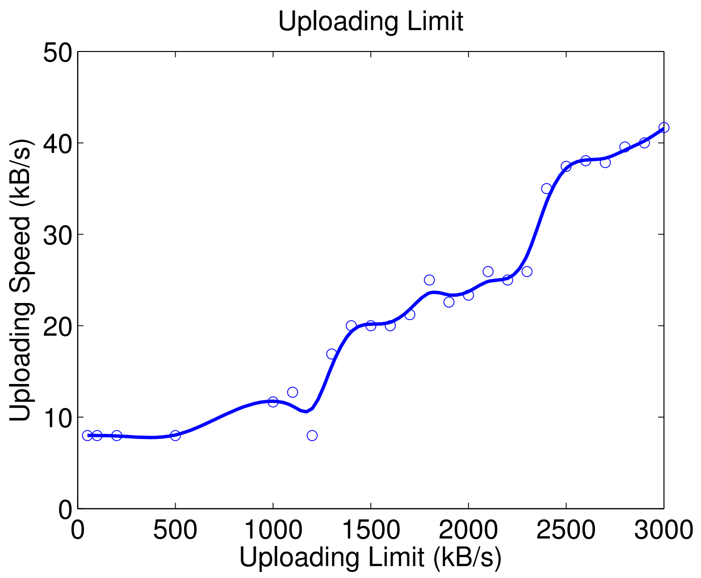


Figure 5.4: Impact of the uploading bandwidth on the uploading speed

The curve from figure 5.4 represents the impact of an uploading limitation of the client when uploading a file. First thing we can notice is that the uploading is very low compared with the limit. Here it can be easily explained because: the uploading limit is the limit for the client which means that it is the available speed for the 7 nodes of the experiment. So it is easily understandable that the speed is divided by 7.

Once that this is made clear, the results look much more plausible. The curve under high limit speed is completely predictable since at this stage the bandwidth is the limiting factor of our system. It just confirms that the comportment of our system supports high uploading speed. Note that on this figure like on the followings ones the perturbation on the curve are a normal symptom of a real network where the network traffic is not always the same.

On the other hand, the results under low speed limitation are less predictable since the curve is steady around 500kB/s. It is due to the fact that not only the data are transmitted to the network, we need also bandwidth for the details of the IP protocol and more important the SCSI command (remind that we use package `open-iscsi-utils` package to transmit our data over the internet). So when the bandwidth get limited there is less and less place for our data in the bandwidth since most of the spaces is used by the iSCSI commands.

By this experiment we have then highlighted one of the limitation of a real system or at least from the software.

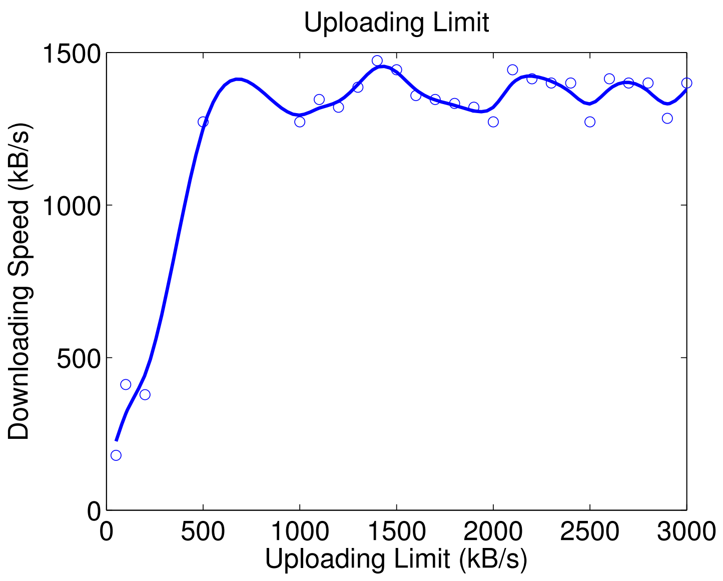


Figure 5.5: Impact of the uploading bandwidth on the downloading speed

This second curve from figure 5.5 is here to show the impact of the uploading on the downloading operation. At first sight this reasoning can appeared to be useless since the downloading of a file does not require uploading so the curve should be steady but as you can see on the curve the results are not that simple. On the left end of the graph the curve is not steady and it begins from a minimum and then increase up to the limiting speed of 500 kB/s. Note that as before, the uploading limit has to be divided by 7 in order to take the 7 nodes into account.

The reasons for that are not that simple but we can see two beginning of answers: the first one is related to what we said before, the data contained in the file are not the only one that we need to transmit over the network. iSCSI protocol and IP protocol also need bandwidth to make our system usable and they need both uploading and downloading bandwidth.

The other reason that we can see to justify this behaviour is that when we read a file (and then download it) we need to compute the positions of the blocks that we need to read. This position is then sent to the node and these extra data are limiting the process of downloading since of the node does not know the position he has to send it can not send it and the we can not download it.

Once the uploading limit becomes higher (the right part of the curve) then everything is coming back to the normal and the curve becomes steady (limiting only by the downloading speed).

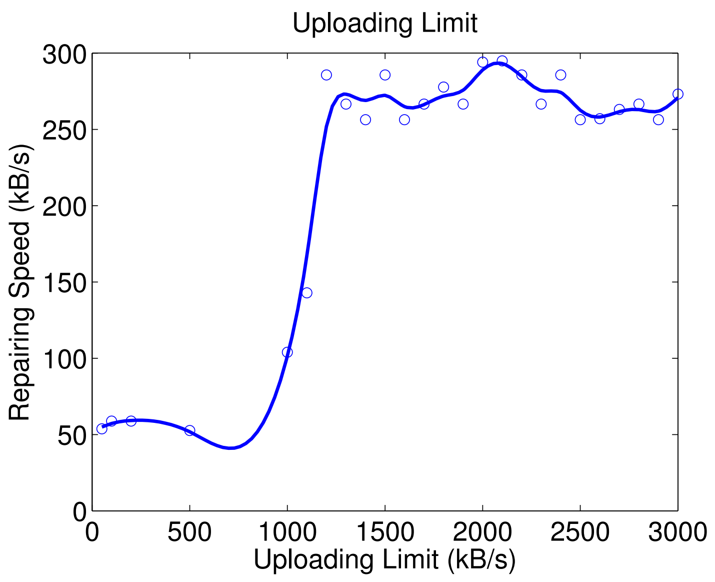


Figure 5.6: Impact of the uploading bandwidth on the repairing speed

This figure from 5.6 is very similar to the previous in the way that we can see on the curve that first the speed increase and then it stays steady (except for the perturbation). As before, as here the repairing consists only in downloading from every node to the client machine the uploading should not have any effects. However, until up to 400 kB/s it has an impact and the reasons are the same as the ones seen before. Except that they even have more influence since we have to send more information to the nodes in order to reconstitute the whole node: we are calling them more often so the beginning of the curve is more steady than in the precedent case.

#### 5.2.2.2 Downloading

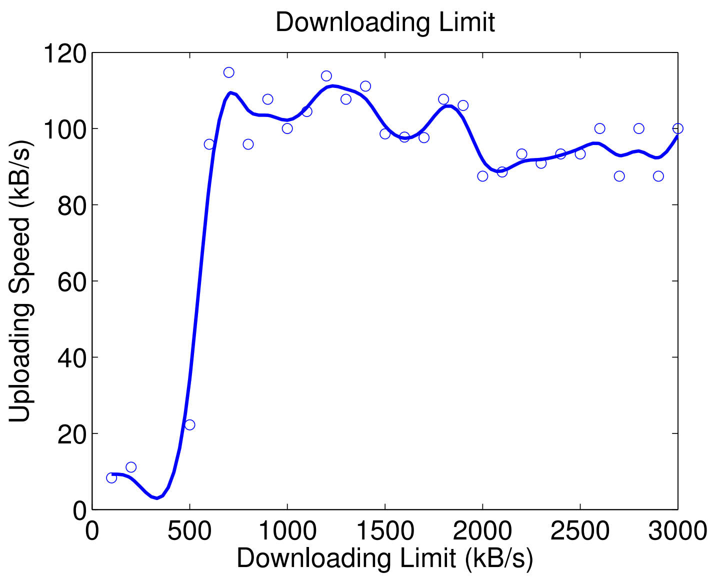


Figure 5.7: Impact of the downloading bandwidth on the uploading speed

For the downloading limit, the same protocol than for the uploading limit was observed.

The graph from figure 5.7 describes the impact of a downloading limitation speed on the behaviour of our system. As we can see the problem is exactly the same than when we limit the uploading speed when downloading a file: the curve should be steady but it is not. It definitely shows a limit of a practical implementation compared with the theoretical approach.

However here the problem seems less imposing: the curve comes more rapidly to its steady state. It confirms what we were saying before. When we limit the uploading speed and downloading a file, we have two phenomenons to take into account: the iSCSI protocol and the fact that we send to the nodes the position and the size of the block that we want. Here (uploading a file under downloading limitation speed) the situation is different: we do not need to care about the position and the size of the block (since the uploading speed is not limited). The only thing that can explain the behaviour of the system under low downloading limit is the fact that the information included in the protocol have more trouble to come from the nodes to the client.

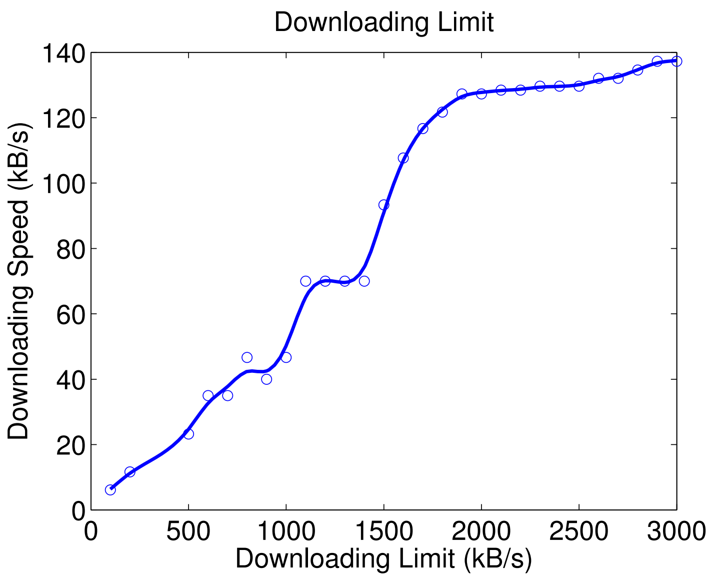


Figure 5.8: Impact of the downloading bandwidth on the downloading speed

The figure 5.8 is the representation of the downloading speed of a file when we limit the downloading speed. This experiment looks quite satisfying since the system evolves as it should. Indeed, we can see that except for the perturbations the speed progress regularly with the downloading limitation speed which is what we expected since at this stage the downloading speed is the limiting factor (indeed the disk are faster than it and we have shown before that the CPU speed was much faster than what is needed for this kind of system).

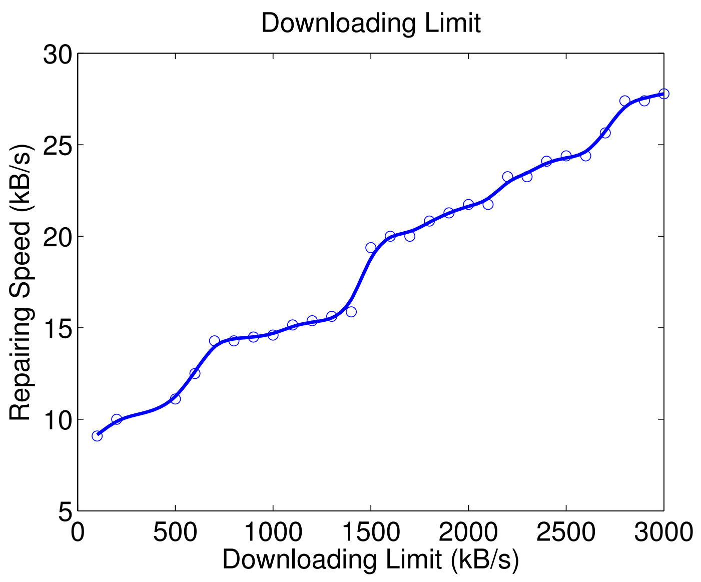


Figure 5.9: Impact of the downloading bandwidth on the repairing speed

This figure 5.9 is the representation of the repairing speed of a node when we limit the downloading speed. Here again there is no need for a lot of details since it corroborates what we could expect. Indeed, as here the repairing of a node consist in downloading the data from all the nodes to reconstruct the node on the client machine, the limiting parameter is the downloading bandwidth so it is completely normal than the repairing speed evolves regularly with the downloading speed limit.

### 5.2.3 Hard Drive

This section will be organised in two parts. In the first, we will get a computer that has a SSD Flash Hard Drive. As it was possible to get only one we will have to store the IMG files on only one machine which is not optimal but at least we can compare the performance of this device with a classic disk in the same condition (IMG files on the same machine). In a second time, we will try to look at the impact of the prioritisation of the task on the disks in order to have a better understanding of the importance of this peripheral. Note that in this case we don't need PlanetLab nodes any more so we come back in the configuration of part 4.1.3.1.

#### 5.2.3.1 Use of Flash Memory

In this subsection, we are going to look at the impact of a SSD drive. The reason why we are doing that is very simple: more and more computers are equipped with this technology. And even if it is still very expansive, it might very replace the common hard drives in a few years.

First let's have a quick reminder of what is a SSD: a Solid-State drive is a data storage device that uses solid-state memory in order to store data. The main difference with the common HDDs is the fact that they are not electromechanical so there are not any moving parts which make them much faster (namely in terms of disk access) and more resistant to physical shock but they are much more expensive and can only support a limited number of writes over the life of the device.

To fix the ideas concerning the performance of a SSD compared with an actual HDD, in table 5.1 is a quick overview of the data transfer rate [4]:

| | SSD | HDD |
|---|---|---|
| Transfer Rate (MB/s) | 65 | 500 |

Table 5.1: Disk Performances

Let's now see how we are going to test these devices. With the same machine than before we are going to create the IMG files and then mount it on the `/dev/loop/` devices. We are going to do the same on the machine that has a SSD device (and the exactly the same configuration regarding the CPU and the RAM). As we said before this cannot be optimal since every node is stored on the same computer so the transfer rate will be strongly affected but this experience is more here for the comparison between the 2 devices so as long as the experiments are done in the same conditions on the 2 machine the result should be quite coherent.

The table 5.2 describes the downloading speed of a file, the uploading speed of a file and the repairing speed of a node on the 2 devices:

| | Uploading Speed (MB/s) | Downloading Speed (MB/s) | Repairing Speed (MB/s) |
|---|---|---|---|
| SSD | 11.94 | 179.2 | 5.1 |
| HDD | 17.4 | 3.6 | 7.2 |

Table 5.2: Performance of our system with HDD and SSD

As you can see in the table, there is very few difference between SSD and HDD for the uploading operation and this is quite surprising. Our conclusion is that this phenomenon is partially due to the influence of the cache layer which minimises the number of writing disk access during the writing of data to the node. This has the same influence on the repairing speed since the SSD device is only slightly better than the HDD device compared with the performances predictable from the transfer rates table. Furthermore, as discussed during the presentation of this project, it appears that the performances of a SSD device regarding the writing are not as good as the one expected from table 5.2 and as 7 nodes are stored on the same disk this phenomenom is strongly amplified.

But as expected there is a really huge gap between the SSD and the HDD regarding the downloading speed which corresponds to the performance from the previous table.

#### 5.2.3.2 Prioritisation

As said before we will use `ionice` in order to control the disk access. This is made feasible since thanks our virtual machine, the different IMG files that represent the nodes are seen directly as hard drives on the Virtual Machine so every time we will try to read or write on one of this disk it will appear for `ionice` as a real disk access when in fact it a distant disk access. This technic will allow us to control the disk prioritisation of every node.

In order to use `ionice`, we first need to run our software normally then get its `PID` thanks to the following command:

```
ps aus | grep mbr
```

If the `PID` is for example 1234, we call `ionice` with the command:

```
ionice -c 3 -p 1234
```

The `-c` option is here to specify the priority given to the process by the user. 1 will be a `real time` priority, 2 a `best-effort` priority and 3 an `idle` priority.

The result of the prioritisation appears in table 5.3:

| | Downloading Speed (MB/s) | Uploading Speed (MB/s) | Repairing Speed (MB/s) |
|---|---|---|---|
| Real Time | 9.18 | 47.7 | 5.9 |
| Best Effort | 6.63 | 50.23 | 5.40 |
| Idle | 7.79 | 50.2 | 7.02 |

Table 5.3: Impact of prioritisation on the system performance

As you can see this experiment did not give us any significant results but after having a look at this result we can conclude that these results are normal. Indeed, while the system is running there is no other process that requires access to the disk so any kind of prioritisation will give the same speed since our system will be high-priority (since he is alone).

## 5.3 Impact of the system configuration

In the previous part we saw the impact of the hardware on the system but we must not forget that the performances also depend on the way a user is going to use our software so in this section we are going to have a look to every parameter that the user can modify and that could have an impact on the performance.

In the following experiment, we fix $k = 2$ and we change the parameter $n$ to see what are the consequences on the uploading and downloading speed. Note also that we are still in the configuration from chapter 4.1.3.1.

### 5.3.1 Impact of the number of nodes

Obviously the first parameter that the user can modify is the number of nodes and that is why we are beginning with it.

#### 5.3.1.1 Uploading Operation

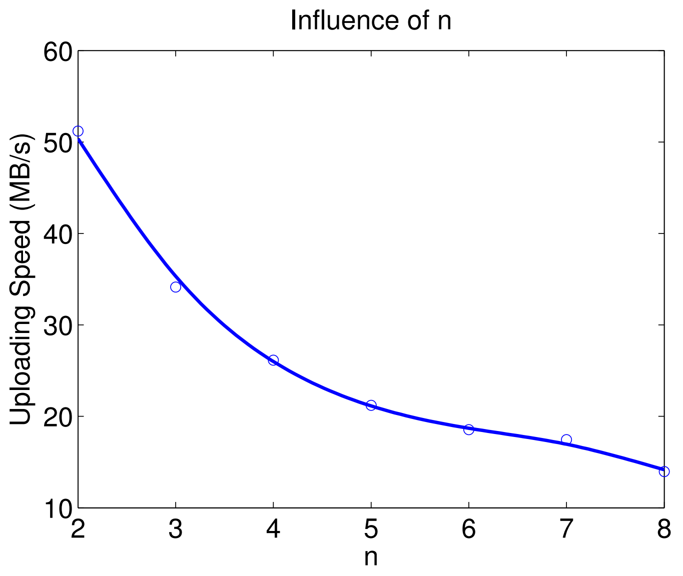


Figure 5.10: Impact of n on the uploading speed

The figure 5.10 represents the influence of the number of available nodes on the uploading speed of a file. Here we can see that the speed decreases with the parameter $n$. So the question is why does it have an impact when in our computation of the speed we take into account that the size of an output aggregated block depends on $n$ (for example a file of size 512 MB will have an output of 716 MB for $k = 3$ and $n = 7$ but will be different for $n = 6$). As in the computation of the speed we take this into account this curve should have been an horizontal line.

The possible reason for that it comes from the fragmentation of the disk. Indeed, every time we are writing a block on the physical disk of one of the node. It has to look for a new location on the disk with enough space to store the block. There are several possibilities: either he finds a space on the disk but the next time it will have to look further. Either it does not found enough space and in this case it fractions the data and then put it in different locations.

This operation takes time and could explain this performance decrease since when we increase the number of nodes (for a fixed $k$) the size of output data (in other words the number of blocks) increases which means that the disk node has to look for more space.

To illustrate this argument, you can find the size of the output data for an input data of size 512 MB and a fixed $k = 2$ in table 5.4:

| n | Size of the Output Data (MB) |
|---|---|
| 3 | 1024 |
| 4 | 1228.8 |
| 5 | 1462.85 |
| 6 | 1706.67 |
| 7 | 1954.90 |
| 8 | 2205.53 |
| 9 | 2457.6 |

Table 5.4: Size of the output data for an input data of size 512 MB and a fixed $k = 2$

This table was computed thanks to formula given in [20] which says that for a file of size $B$, the data to be transferred will be $\frac{2Bn(n-1)}{2k(n-1)-k^2+k}$. It confirms our idea in the way that there is a huge variation of the output size with the number of nodes.

#### 5.3.1.2 Downloading Operation

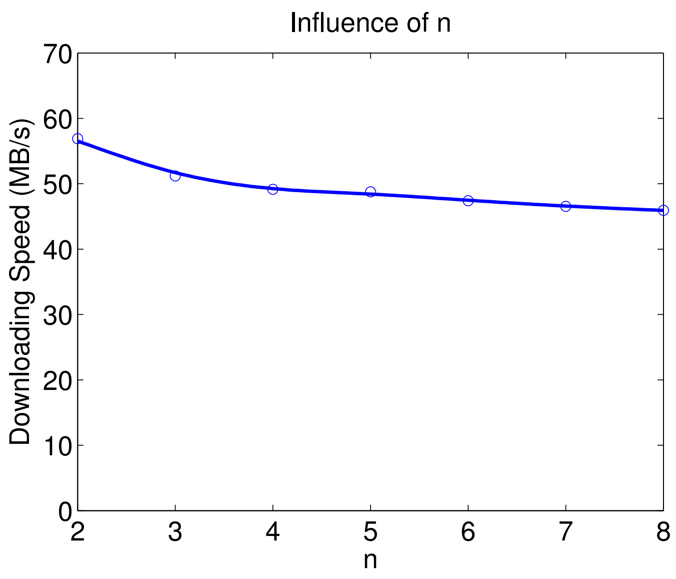


Figure 5.11: Impact of n on the downloading speed

The figure 5.11 represents the influence of the number of available nodes on the downloading speed of a file. Here there is no surprise like on the previous graph: the curve is not really an horizontal line as it should be but it is quite stable around 50 MB/s which is the approximate transfer rate of a data disk.

So this experiment allow us to conclude that there is no real influence from the number of nodes on the downloading speed and that over an ethernet network (where there is no real limitation on the bandwidth) the limiting factor of our system is the transfer data rate of every disk.

#### 5.3.1.3 Repairing Operation

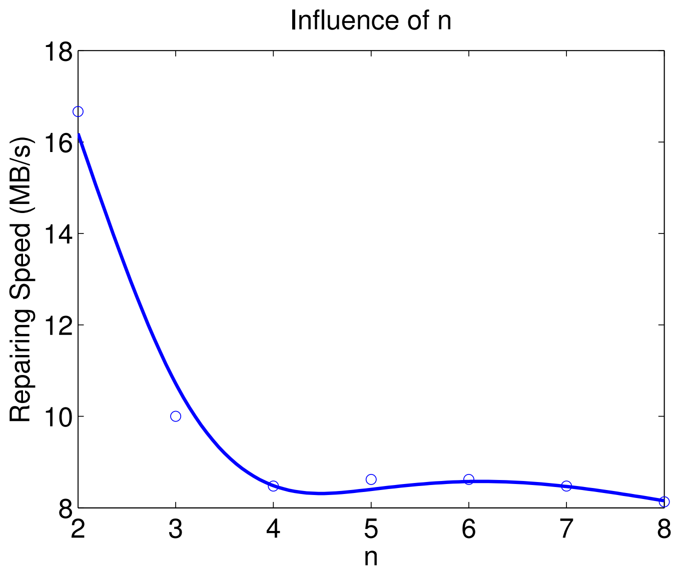


Figure 5.12: Impact of n on the repairing speed

The figure 5.12 represents the influence of the number of available nodes on the repairing speed of a node. As on every figure of this section regarding $n$ (and the next one dealing with $k$), the result should be an horizontal line. This the case for the end of the curve so it is quite logical but there is the left part of this curve which is not a straight line. It is due to the fact that when there is only 2 nodes, this 2 nodes are the exact copy from one to the other so when we need to repair one of them we just need to make a copy of a disk. There is no more computations require and that is why as soon as there are 3 available nodes things come back to the normal.

### 5.3.2 Impact of the number of nodes from which the system has to be able to retrieve a file

Now that we have seen the impact on the performance by the number of nodes we are going to see the impact made by the parameter $k$ (the necessary number of nodes to retrieve a file).

In the following experiment, we fix $n = 7$ and we change the parameter $k$ to see what are the consequences on the uploading and downloading speed.

#### 5.3.2.1 Uploading Operation

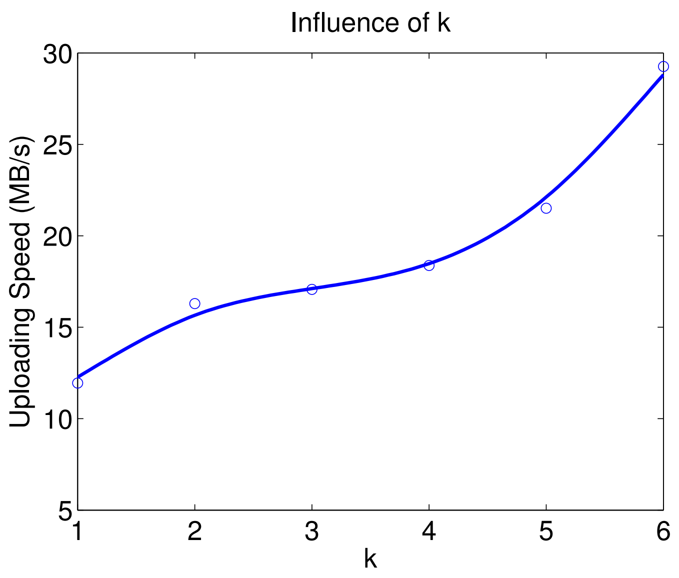


Figure 5.13: Impact of k on the uploading speed

The figure 5.13 represents the influence of the $k$ parameter on the uploading speed of a file. It confirms what we saw in the previous section regarding the influence of $n$ on the uploading speed. Indeed, here again the result should be an horizontal line but we can see that it increases even by taking into account the size of the output data. It confirms our theory on the fact it is due to the fragmentation of the data which increases much faster when there is a huge amount of data that has to be transferred. Here the size of the output data decreases with $k$ so there is less and less fragmentation and then the speed increases.

You can find the size of the output data for $n = 7$ and an input size of 512 MB in the table 5.5:

| k | Size of the Output Data (MB) |
|---|---|
| 1 | 3584 |
| 2 | 1228.8 |
| 3 | 1462.85 |
| 4 | 1706.67 |
| 5 | 1954.90 |
| 6 | 2205.53 |

Table 5.5: Size of the output data for an input data of size 512 MB and a fixed $n = 7$

This table was computed thanks to formula given in [20] which says that for a file of size $B$, the data to be transferred will be $\frac{2Bn(n-1)}{2k(n-1)-k^2+k}$. It confirms our idea in the way that there is a huge variation of the output size with the parameter $k$.

#### 5.3.2.2 Downloading Operation

Let's now do the same for the downloading operation:

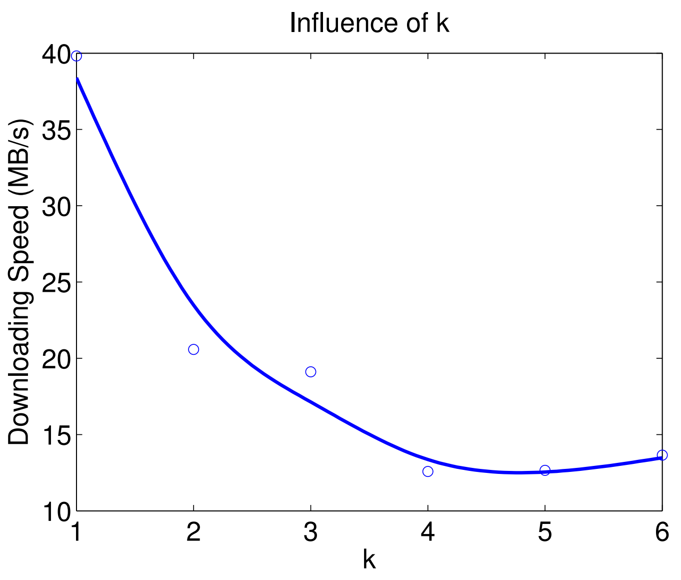


Figure 5.14: Impact of k on the downloading speed

The figure 5.14 represents the influence of the $k$ parameter on the downloading speed of a file. It is very similar to the one we got regarding the influence of $n$ on the downloading speed regarding the right part of the curve which is pretty stable around 15 MB/s and this is what was expected since the $k$ parameter should not have any influence on any operation.

But when $k$ is only equal to 1, we can notice that the speed is much higher. To understand this phenomenon we need to come back to the definition of $k$ which is the number of nodes from which the system can retrieve a file. In this case, this number is 1 so it means that the system can recover from only 1 node and by doing so every information data is on every disk. That is what increases the downloading speed: we are just copying a file from 1 node to our system.

#### 5.3.2.3 Repairing Operation

To finish with the impact of $k$ on the system, we need to study its influence for the repairing operation.

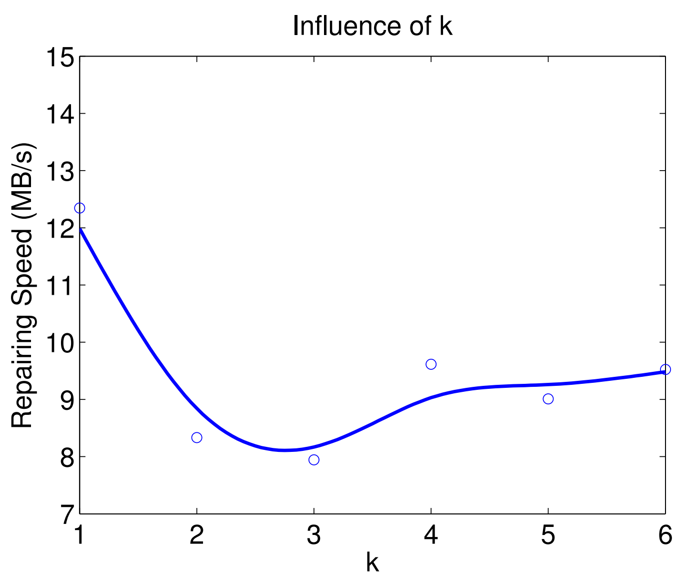


Figure 5.15: Impact of k on the repairing speed

The figure 5.15 represents the influence of the $k$ parameter on the repairing speed of a node. Here we can see that the results match pretty well the expectation. Except for the $k = 1$ case there is very few influence from $k$ on the system. The justification of this unexpected result for $k = 1$ is again the fact that as $k = 1$ we then need only to copy what contains only one disk to another and we do not have to select which block comes from which node: so the speed is increased.

### 5.3.3 Impact of the cache

A parameter that will not have thought at the first sight but that might have a huge importance is the cache size that is attributed in our input/output layer. The following experiment will increase the size of the cache allocating in the storage layer and look at the impact on the uploading and downloading speed.

Note that in order to have a larger range for the cache of the size, the size of the RAM on the client virtual machine was set to 4 GB.

#### 5.3.3.1 Uploading Operation

Let's start with the uploading operation and the following experiment.

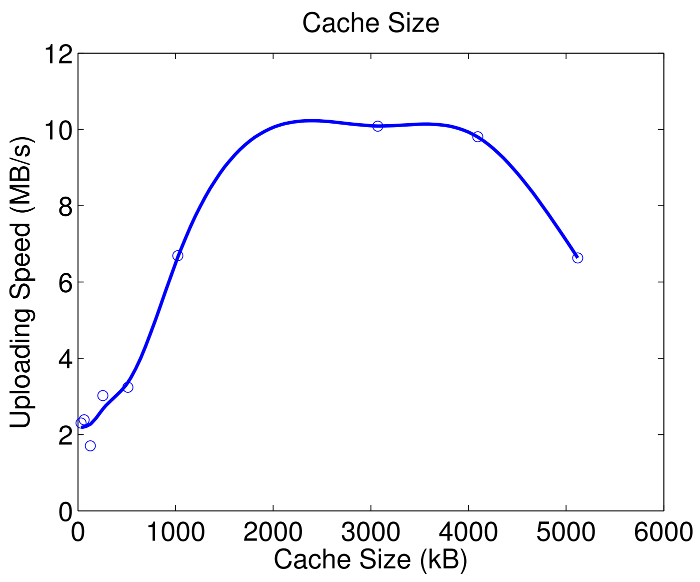


Figure 5.16: Impact of cache size on the uploading speed

The figure 5.16 represents the influence of the cache size on the uploading speed of a file This figure may be one of the most important of this chapter. Indeed it shows the impact that can have our input/output implementation on our system. As you can see we can distinguish two parts on the curve: one when it increases and the the other one when it decreases.

Let's begin with the first part and let's see why the speed increases with the cache size. In the definition of the cache that we gave in the software architecture when we are writing a block to the disk we first go through the cache memory (allocated by the software). If there are no read operations then the cache memory will only flush the data to the disk when it is full. Here we are uploading a file so there are no read operations and so we only access the disk when the cache is full. By doing it this way we are doing much less disk access than if we were directly writing on the disk for every block. So we don't have to wait for the average 3 ms latency time of a disk at each access which greatly increases the speed.

You can also notice that there is a part that increases much quicker before 1 GB and a part that increases slower. This slow down is due to the fact that we have limited the number of access but now we are limited by the transfer rate which can not be bypassed by the cache.

On the other hand, there is a decrease after 3 GB of cache size. Even if at first sight there are no apparent reasons for this phenomena, we have to remind that the size of the RAM is limited to 4 GB so when we try to create a cache in our system of size 4 or 5 GB there is not enough space and the machine has to free memory and then reallocate it in order to be able to provide such a big space and this process takes time. As you can see the maximum speed appears just before that the size of the cache equals the one of the RAM. This experiment was repeated with different size of cache and this property was always verified.

#### 5.3.3.2 Downloading Operation

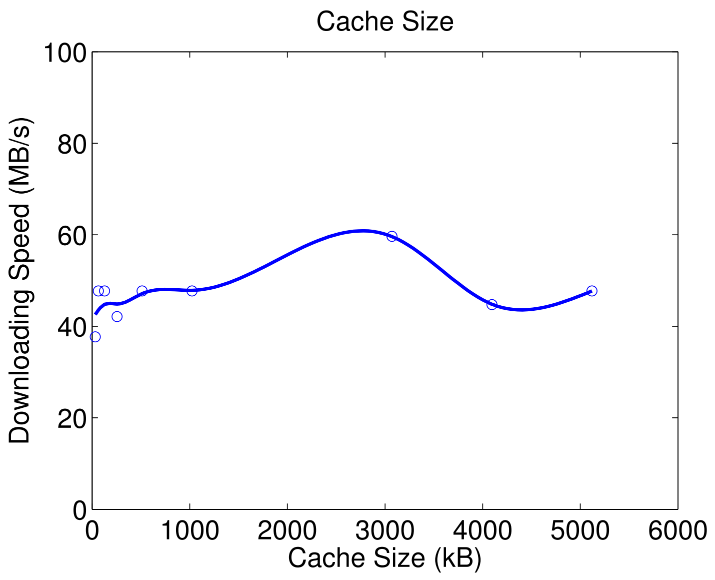


Figure 5.17: Impact of cache size on the downloading speed

The figure 5.17 represents the influence of the cache size on the downloading speed of a file. We saw that the cache has a great impact of the uploading of a file however there are very few improvement of the speed when we use the cache when we are downloading a file from the node.

So what can explain this difference ? As we said before the improvement made in the previous part allow us to reduce the number of disk access when we write a file but here we want to read it. So what happens when we read a file that does not happen when we write it ? From our cache definition, when the user request a read operation, it is handled by the cache layer that checks if the block is available in the cache. As here it is never available we then need it to retrieve it from the physical memory and that for every block so our cache layer becomes useless since it accesses the disk exactly the same number of time than without it.

#### 5.3.3.3 Repairing Operation

Let's finish this experiment part with the one from figure 5.18:

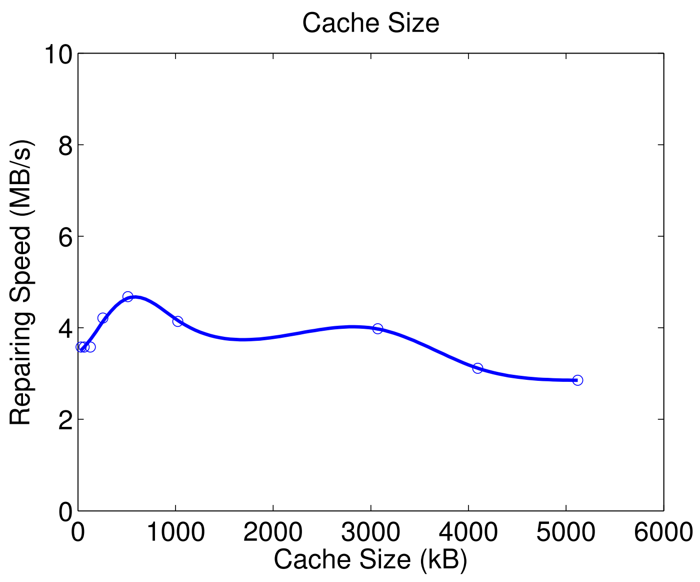


Figure 5.18: Impact of cache size on the repairing speed

The figure 5.18 represents the influence of the cache size on the repairing speed of a node. Here again we can notice that the cache size has a very limited impact on the system performance. The reasoning to explain that is exactly the same as the previous case.

Indeed, the repairing is constituted of the 2 operations: read and write. So the write is actually improved by the cache but it becomes limited by the number of access disk made by the read operation and as the algorithm is sequential: every write happens right after a read the write operation is limited because of the read operation.

## 5.4 Conclusions

This section is here to summarize the different conclusions that were made in the previous of this chapter. We will first give some general remarks regarding the whole set of experiences, then analyse in more details the hardware section and then the parameters section.

### 5.4.1 General Remarks

So first let's recall the main goal of this project: is it feasible to realize a distributed storage system based on an Minimum Bandwidth Exact Regenerating ? Indeed, in a context where there are a lot of literature regarding the use of erasure codes in distributed storage and very few implementation of them, it was becoming interesting to know if it was feasible in practice and not only in theory. According to this project, the answer to this question is positive since as we are going to detail in the next paragraph the speed in which we are able to achieve downloading, uploading or repairing operations are very reasonable.

Indeed, according to the experience given from these experiments, on a normal utilisation the average speed in order to upload a file is approximately 15 MB/s (note that we still refer to the speed of the output in order to compare the system for different parameters. The downloading is approximately 40 MB/s and the repairing speed is approximately 7 MB/s. The main fear at the beginning of this project was to have ridiculously low transfer speed such as 1 kB/s or so. Obviously that is not the case and it shows that all the theory found in the literature can be put into practice in a real system.

Note that download operations generally have an higher speed than upload operations, mainly because the system can only download one copy of each native block without the need of accessing other code blocks, or duplicate blocks.

### 5.4.2 Hardware Components

Ultimately this section has for goal to say which piece of hardware is to be preferred if we want to increase the performance of the system but in order to do that we first have to distinguish 2 different utilisations: over an ethernet network (where there is no limitation) or over the internet (where we are limited by the speed of our bandwidth).

- **Utilisation over an ethernet network**

Over an ethernet network, the available bandwidth between the nodes is really huge so we have only to take into account the other hardware components since it will never become the limiting part of the system.

So let's come back to the experiment and first on the CPU ones. As we have seen, except if we are only using less than half the power of the CPU, the system will not face any slowing down (neither for the downloading, the uploading or the repairing). So it means that except for old computers there is absolutely no need in upgrading the CPU frequency.

It then means that the piece of hardware that needs to be upgraded is the disk. The problem is that there is a disk on every node and so the performance is equivalent to the performance of the weakest of the disk. As we have seen with the cache section, reducing the number of access to the disk greatly increase the performance and only after comes the need for a higher data transfer rate. So if we want to increase the performance of our system the first thing to do is to find disk that a better access time and then a better transfer rate. But we have seen this problem may very well become obsolete one day with the use of SSD device. However a question persists regarding the use of such a device: we know that its lifetime is limited by the number of writes made on it so we can wonder how pertinent it is to use it in a distributed storage system where the goal is to write a lot of data.

- **Utilisation over the internet (utilisation on PlanetLab nodes in our case)**

Here the problem is that we choose to use to distribute our storage over distant nodes so we have to use an IP protocol for SCSI command (iSCSI). So let's now have a look back on the experiments. First thing to notice is that the internet bandwidth of the client is the limiting element. Indeed, it has to be divided into the servers so there is no point having 8 servers that have a huge internet bandwidth if the client has not 8 times that speed.

Another point we can notice thanks to the experiments regarding the bandwidth is that must distinguish the uploading bandwidth and the downloading. And one important notice we made about that is that even if we want a system that download a file at a very high speed (but we judge that it is not necessary that it uploads at a high speed because for example we can upload during the night at home but we need the data quickly when we are at work the day after), then we must not neglect the uploading bandwidth since if it is too slow it also limits the downloading speed. The same reasoning holds in the opposite case.

### 5.4.3 Parameters

This section was much more surprising than expected. Indeed, according to the theory, that should have been no impact from $k$ and $n$ on the performance of the system but in real case the results are pretty different. Indeed, the thing we have to remember is that the fragmentation of the disk nodes has an important part in the result when the size of the output data increases and this is what limits the performance

Another good surprise is the fact that the cache layer of our implementation has a great influence on the system performance. The main interest of this layer appears when we are uploading a file but it is very efficient as long as we stay in the limit of the RAM since it limits the impact of the disk access latency (before hitting the data transfer limit).

# Chapter 6

# Twin Codes Extension

This chapter could have taken place into the conclusion as an opening or a future work because time was missing to implement this idea and including it in our system. However there are a lot of things to explain such it will not have fitted in the conclusion and by working on it, some ideas have emerged and make it worth to write a chapter about the Twin Codes idea [19].

## 6.1 Presentation

Twin Codes are a very recent idea developed in [19] that are based on codes such as Reed Solomon and that is the main reason we began to get interested in it.

This section will begin by highlighting the context in which this algorithm was introduced and then give a general idea of what is the use of twin codes.

### 6.1.1 Context

As we said before, we are in a context where we have $n$ nodes and we want to be able to retrieve a file from $k$ out of the $n$ original ones. We also want to regenerate (or repair) a failed from the remaining $n-1$.

As we have seen in the experiences, this operation is very bandwidth consuming and that is why algorithm such as the Minimum Bandwidth Exact Regenerating code from [20] have been developed in order to minimize the require bandwidth to repair a node.

As we have seen, this minimization requires a trade-off between the required bandwidth and the storage that we need to store on every node as you can remind from the curve from figure 6.1 [7]:

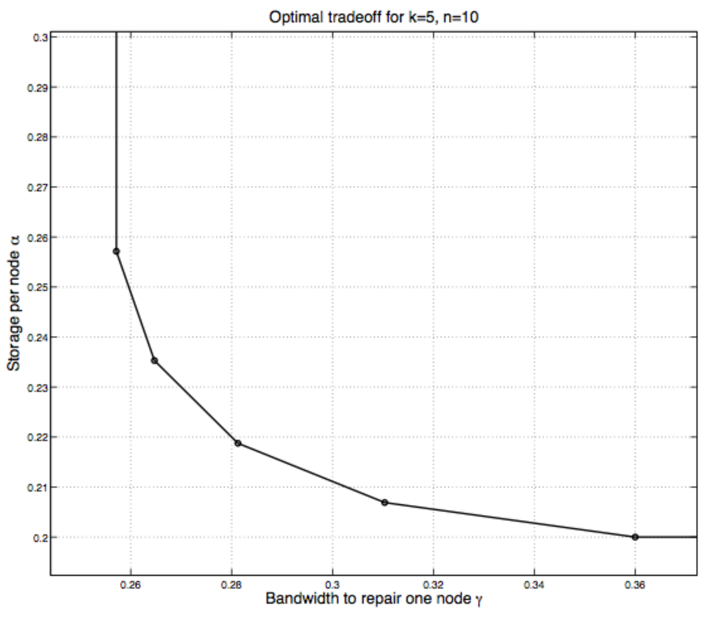


Figure 6.1: Trade-off curve between storage $\alpha$ and required bandwidth $\gamma$ for $k = 5$ and $n = 10$

The code that we have implemented operates at the MBR point of this curve which means that it minimize the bandwidth but absolutely not the storage. More and more codes are developed in the literature to operate at this point and very few are implemented. Some are for example using interference alignment [28], or product matrix [18]. On the other hand, most other points of this curve are shown to be not achievable according to [26] where the authors show the non-achievability of the interior points on the storage-bandwidth tradeoff under exact-repair is established, with the possible exception of points within the immediate vicinity of the MSR point.

However, it could have been very useful if we could minimize both the storage and the bandwidth: we would not have to care any more about which of the two has to be minimize first. We can also notice that there is a lot of literatures regarding erasure codes such Reed-Solomon codes, BCH codes, Goppa Code and Hamming code. So a natural question could be, if there are so many codes, maybe there is a way to arrange them in order to minimize both the bandwidth and the storage because as we have seen in the experiment conclusions the output data size has a huge impact on the performance (without talking about the fact that it will require less space to store a file and then it is less expansive).

That is what is proposed in [19] by their authors with their proposition of the twin codes. As it will be detailed in the next subsection the idea is to use two codes rather than one to increase the performance of the system. According to them, they will both minimize the bandwidth and the storage. And that is what they show in the figure 6.2 extracted from [19] and that represents the bandwidth/storage trade-off where $k = 10$ and $n = 19$:

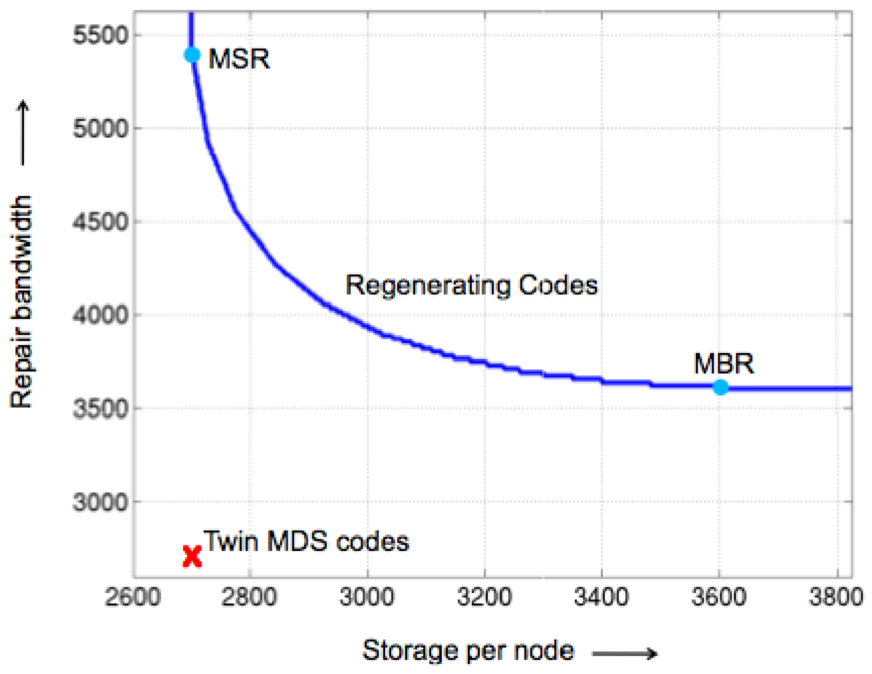


Figure 6.2: Trade-off for the Twin Codes

Now that we have seen what the authors propose to achieve let's take a look at how they plan to achieve it.

### 6.1.2 Main Idea

The Twin-code framework is developed to suit a distributed storage network and has mainly two goals. The first one is the data reconstruction of a file and the second one is the repairing of a failed node.

Under this framework, there is a key idea on which we need to insist in a first time: in our previous system we had $n$ different nodes that were all used by the same code $C$ (Reed-Solomon code) but in the following the $n$ nodes will now be separated into 2 different types. In order to make it more clear we will call the first ones nodes of type $T_1$ and the other ones will be nodes of type $T_2$. In order to get the data stored on $T_1$, we need to encode the original data $M$ with a code $C_1$ and in order to get the data stored on $T_2$ we first do a permutation of the same original message $M$ and then we encode it with a code $C_2$ that can be the same as $C_1$.

As we will see in the next section this construction has 2 interesting properties:

- we can repair a node of a certain type thanks to a subset of nodes from the other type
- the client machine can rebuild the original data thanks to a subset of nodes of the same type.

You can see a representation of these two properties thanks to the following figures:

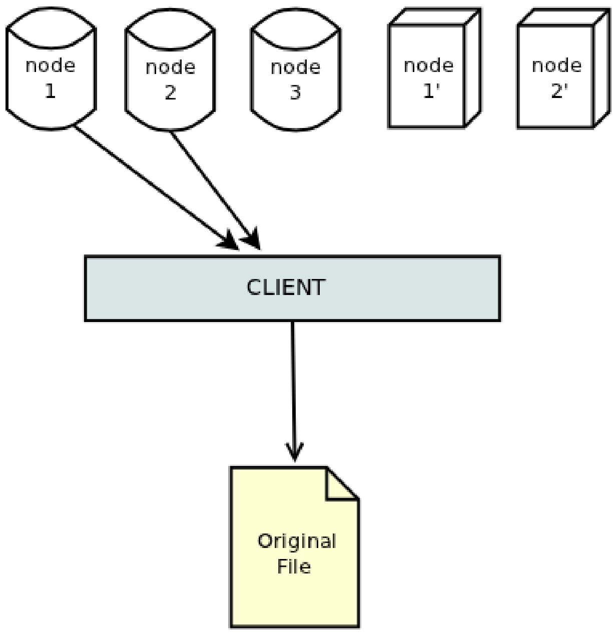


Figure 6.3: Downloading Operation for the Twin Codes

On the drawing from figure 6.3 we can see that the client can rebuild the original file thanks to a subset of the cylinder node.

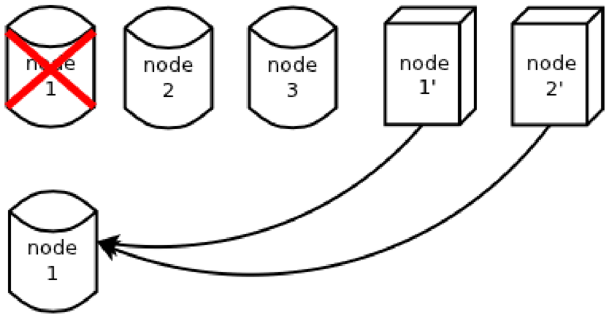


Figure 6.4: Repairing Operation for the Twin Codes

On the drawing from figure 6.4 we can see that the node 1 from the cylinder type has failed and that we can reconstruct thanks to a subset of the other type.

The problem of this framework is that it combines several code and that we have to transform the original message so it is more delicate to apply in a real environment but it has other advantages: as we have seen in the previous section it enables the system to minimize the amount of storage on every node and it also allows to minimize the bandwidth to repair a node. By doing so it operates at a point which was unreachable for a a classic Regenerating or even for the algorithm we have studied during this whole project and that is this property that we want to use to improve our system.

Note that there is a main drawback that the authors from [19] are not dealing with which is the fact that in other distributed storage system based on erasure codes we need $k$ nodes out of $n$

to retrieve the original but here we need $k$ nodes from the same type of nodes so we need $k$ nodes out of $n_1 < n$ nodes which is a constraint more difficult to achieve.

## 6.2 Algorithm

Now that we have seen what this framework wants to be able to do we need to see how it can be done. To do so, we are first going to see how we create a system with 2 codes then how we reconstruct the data and how we repair a node and at last we will see a quick toy example in order to get a better understanding of the theoretical point o view.

### 6.2.1 System Generation

Let's first consider the notations that we will use:

- $n_1$ and $n_2$ represent the number of nodes of type $T_1$ and $T_2$ with $n = n_1 + n_2$
- $B$ is the number of symbols from the finite field $\mathbb{F}_q$ that constitute the original message
- $C_1$ and $C_2$ represent codes that can transform a message of size $k$ from the finite field $\mathbb{F}_q$ symbols into an encoded message of $n_1$ or $n_2$ symbols from the finite field $\mathbb{F}_q$ which will allow to correct errors.
- $G_1$ and $G_2$ represent the generator matrices of the previous codes that have a $n_1 * k$ size and a $n_2 * k$ size. And will call the $l$th row of $G_1$: $g_{(1,l)}$ and the $l$th row of $G_2$: $g_{(2,l)}$

Now that we have made these few definitions clear, we are going to explain how we construct the distributed storage thanks to these 2 codes.

First thing to do is coming back to the original message: we divided into $k^2$ blocks which by doing so limit the size of the incoming message to a size $B = k^2$ (as for the Minimum Bandwidth Exact Regenerating Code if the size of the file is bigger than $B$, we split it into several chunks and encode sequentially each chunks).

Once this is done we can represent the original message into a matrix of size $k * k$ that we will name $M_1$. Once we have this matrix, we also need its transpose that we will denote:

$$M_2 = M_1'$$

We then need to know which data we are going to send to the nodes of the system. In this framework we decide that the $l$th nodes of type $T_1$ will store the result of:

$$g_{(1,l)} M_1$$

and the $l$th nodes of type $T_2$ will store the result of the following multiplication:

$$g_{(2,l)} M_2$$

In other words, the data store on the nodes of type $T_1$ are the data from $M_1$ encoded thanks to the code $C_1$ and the nodes of type $T_2$ store the data from $M_2$ (which are a permutation of $M_1$) encoded thanks to the code $C_2$.

Using this method, every node stores exactly $k$ symbols from $\mathbb{F}_q$ and this is the minimum a node can store.

In the following, we will explain how we can reconstruct the original data and how we can repair a node thanks to erasure codes.

### 6.2.2 Data Reconstruction

In this section we need to explain how the client machine can rebuild a file thanks to a number $k$ of the same type nodes. Let's assume that the client can connect to $k$ different nodes of one type (either $T_1$ or $T_2$), we then have access to $k^2$ symbols. Here we suppose that the chosen node type to reconstruct the data is $T_1$ (but the same applies for $T_2$) and we also suppose than the $k$ nodes that we use are the $k$ first ones but it would be the same with any combination with any combinations of $k$ nodes out of $n_1$. We then have access to the following multiplication matrix:

$$S = \begin{pmatrix} g_{(1,1)} \\ g_{(1,2)} \\ \vdots \\ g_{(1,k)} \end{pmatrix} M_1$$

Let's now focus on the $k$ first symbol of the first column of the matrix $S$. They are the $k$ symbols corresponding to the encoding of the $k$ symbols of the message $M_1$. If we follow the same reasoning for every line of this matrix we then understand that every column of the matrix $M_1$ has an encoded version in the $S$ matrix.

It then means that all the $n_1$ columns of the matrix $S$ can be decoded independently from the other columns. We can see every column as an encoded word of $n_1$ symbols but on which $n_1 - k$ have been erased so our erasure code $C_1$ can correct it to recover the $n_1$ encoded symbols and then the $k$ original symbols of every column of the matrix $M_1$.

By this process we can then rebuild the original file and we can note that the same holds for the nodes of type $T_2$.

### 6.2.3 Exact Regeneration

In this section we need to explain how the system can repair itself when one of its node fails. As we have seen before, it is possible to repair from a node failure of any type when we have access to a sufficient subset of nodes of the other type.

Here we will suppose that a node of type $T_1$ fails and that we want to repair it thanks to a subset of nodes of type $T_2$. As we have told in the section regarding the system construction, every node contains $k$ symbols so in order to minimize the bandwidth (downloading at most 1 symbol of each node) we need to connect to $k$ nodes of type 2. This simple observation allow us to find the size of the subset we were talking about.

We first need to define which symbols we need to retrieve from the other nodes. If the node that has failed is the node number $h$. Then we need to download from a way or another the following symbols:

$$g_{(1,h)} M_1$$

As before we suppose that it connects to the $k$ first nodes of type $T_2$ but it would not have changed anything for any combination of $k$ nodes out of the $n_2$ available ones. Each of these nodes then send to the failed node $h$ the following result:

$$g_{(2,l)} M_2 g'_{(1,h)}$$

Note that these values are known since $M_2 g_{(2,l)}$ is stored on node $l$ and that $g_{(1,h)'}$ is just the $h$th column of the encoding matrix which is known by the client.

As we have this value for every of the $k$ nodes we connect to, we can then build the following vector of $k$ symbols:

$$\begin{pmatrix} g_{(2,1)} \\ g_{(2,2)} \\ \vdots \\ g_{(2,k)} \end{pmatrix} M_2 g'_{(1,h)}$$

We then are in the same situation as in the data reconstruction case. We can see the resulting vector $M_2 g'_{(1,h)}$ has an encoded word of size $n_2$ from which $n_2 - k$ have been erased and we can correct it using the code $C_2$, we then get:

$$M_2 g'_{(1,h)}$$

Note here that we retrieve the following information: $M_2 g'_{(1,h)}$ which is not yet what we are looking for since it is: $g_{(1,h)} M_1$. But to finally get what we want we just need to compute the transpose of $M_2 g'_{(1,h)}$. Indeed:

$$\begin{aligned} (M_2 g'_{(1,h)})' &= g_{(1,h)} M'_2 \\ &= g_{(1,h)} M_1 \end{aligned}$$

Which is exactly what we were looking for and we then get the repairing of a node.

## 6.3 Toy Example

The previous section was very theoretical and that is we why chose to have a second look at it but this time using a toy example in order to make clear what happens during the 3 steps of this framework.

### 6.3.1 System Generation

Let's assume that we have $n = 9$ available nodes. We split them into the two types $T_1$ and $T_2$ such as we have $n_1 = 4$ and $n_2 = 5$. We then have to choose that the number of nodes from which the system should be able to rebuild the original data is $k = 3$ which supposes that $B = 9 (= k^2)$. We then have to choose the finite field that has more than 5 elements so $2^3$ since it has to be a power of 2 in order to be able to use a Reed-Solomon code for 5 nodes. Let's now define the matrices we need for the following of the example:

$$M_1 = \begin{pmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \\ m_7 & m_8 & m_9 \end{pmatrix}$$

$$M_2 = \begin{pmatrix} m_1 & m_4 & m_7 \\ m_2 & m_5 & m_8 \\ m_3 & m_6 & m_9 \end{pmatrix}$$
$$G_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$
$$G_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$$

Note that the two generator matrices below are Reed-Solomon Generator Matrices found in the literature and that we can the apply the algorithm explained in chapter 4.2.2.4 that consists in deleting the rows corresponding to missing symbols and then invert the matrix to compute the original message.

According to the definition of our system the $l$th nodes of type $T_1$ will store the result of:

$$g_{(1,l)}M_1$$

Which gives us the distribution for nodes of type $T_1$ represented in table 6.1:

| Node 1 | Node 2 | Node 3 | Node 4 |
|---|---|---|---|
| $m_1$ | $m_4$ | $m_7$ | $m_1 + m_4 + m_7$ |
| $m_2$ | $m_5$ | $m_8$ | $m_2 + m_5 + m_8$ |
| $m_3$ | $m_6$ | $m_9$ | $m_3 + m_6 + m_9$ |

Table 6.1: Contents of the nodes of type $T_1$

The $l$th nodes of type $T_2$ will store the result of the following multiplication:

$$g_{(2,l)}M_2$$

which gives us the distribution for the nodes of type $T_2$ represented in table 6.2:

| Node 1 | Node 2 | Node 3 | Node 4 | Node 5 |
|---|---|---|---|---|
| $m_1$ | $m_3$ | $m_1 + m_2 + m_3$ | $m_1 + 2m_2 + 3m_3$ | $m_1 + 3m_2 + 2m_3$ |
| $m_4$ | $m_6$ | $m_4 + m_5 + m_6$ | $m_4 + 2m_5 + 3m_6$ | $m_4 + 3m_5 + 2m_6$ |
| $m_7$ | $m_9$ | $m_7 + m_8 + m_9$ | $m_7 + 2m_8 + 3m_9$ | $m_7 + 3m_8 + 2m_9$ |

Table 6.2: Contents of the nodes of type $T_2$

Now that we have seen how to build a distributed storage thanks to 2 different codes, let's how we can rebuild a file thanks to it.

### 6.3.2 Data Reconstruction

This subsection will take into account what was explained in the corresponding section of the theoretical approach by illustrating it with the same example than before.

In this example we will take the case where we want to rebuild the file thanks to 3 nodes of the type $T_2$. So let's come back to the table 6.3:

| Node 1 | Node 2 | Node 3 | Node 4 |
|---|---|---|---|
| $m_1$ | $m_4$ | $m_7$ | $m_1 + m_4 + m_7$ |
| $m_2$ | $m_5$ | $m_8$ | $m_2 + m_5 + m_8$ |
| $m_3$ | $m_6$ | $m_9$ | $m_3 + m_6 + m_9$ |

Table 6.3: Contents of the nodes of type $T_2$

Here we assume that we want to retrieve the message $M$ thanks from nodes 1, 2 and 4 but the technic holds for every other combination of 3 nodes. The client machine has then access to the following information (found in nodes 1, 2 and 4):

$$S = \begin{pmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_5 \\ m_1 + m_4 + m_7 & m_2 + m_5 + m_8 & m_3 + m_6 + m_9 \end{pmatrix}$$

Now we are going to follow the same protocol than in chapter 4.2.2.4 but this time for each column of the previous matrix. So first we need to delete the rows from $G_1$ and invert its result as explained in 4.2.2.4. We will call the result $H$.

$$\begin{aligned} H &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}^{-1} \\ &= \begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

Now for the first column of $S$, we got:

$$\begin{pmatrix} m_1 & m_4 & m_1 + m_4 + m_7 \end{pmatrix} \begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} m_1 & m_2 & m_7 \end{pmatrix}$$

For the second column of $S$, we get:

$$\begin{pmatrix} m_2 & m_5 & m_2 + m_5 + m_8 \end{pmatrix} \begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} m_2 & m_5 & m_8 \end{pmatrix}$$

And for the last column, we finally have:

$$\begin{pmatrix} m_3 & m_6 & m_3 + m_6 + m_9 \end{pmatrix} \begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} m_3 & m_6 & m_9 \end{pmatrix}$$

As expected, we succeed in rebuilding the original matrix $M_1$ such as:

$$M_1 = \begin{pmatrix} m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 \\ m_7 & m_8 & m_9 \end{pmatrix}$$

### 6.3.3 Exact Regeneration

Let's now have a look to the last part of the system: the repairing of a node. In this example, let's assume that the first node of type $T_2$ fails, we then need to reconstruct it thanks to $k = 3$ out of the $n_1 = 4$ nodes of type $T_1$. In this example, we will take the case where the system wants to repair itself from nodes 1, 2 and 4 from type $T_1$. So we need to retrieve $m_1$, $m_4$ and $m_7$.

According to the theoretical part, we get access to: $g_{(1,1)}M_2g'_{(2,1)}$, $g_{(1,2)}M_2g'_{(2,1)}$ and $g_{(1,4)}M_2g'_{(2,1)}$ which means that node 1 gives us $m_1$, node 3 gives us $m_5$ and node 4 gives us $m_3 + m_6 + m_9$. We can see that as the following vector:

$$U = \begin{pmatrix} m_1 & m_4 & m_1 + m_4 + m_7 \end{pmatrix}$$

Now, as explained in the theoretical part we are going to decode it using the same method as in the previous section:

$$\begin{pmatrix} m_1 & m_4 & m_1 + m_4 + m_7 \end{pmatrix} \begin{pmatrix} 1 & 0 & 7 \\ 0 & 1 & 7 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} m_1 & m_4 & m_7 \end{pmatrix}$$

So we have the $k$ necessary symbols for the reconstruction of the failed node.

This example concludes the algorithm part of this chapter and we are now going to see in the next section what are the advantages and the drawbacks of this framework. And then how we can use it in our system.

## 6.4 Usability with Minimum Bandwidth Exact Regenerating code

This section will not be the more detailed of this document but it can really be a big interest for someone who wants to keep working on this kind of subject by trying to improve existing system.

### 6.4.1 What can be gained using this framework ?

This subsection could have been named: what are the properties from the Twin Codes that we would like to transfer to the Minimum Bandwidth Exact Repairing code ? Indeed there are several good properties that are interesting for any Distributed Storage systems.

- **The first important property can be found in the node repairing part of this framework.**

When we are trying to repair a node, we need to get trough each of the $k$ selected nodes and each of them are independent so they do not have to get any knowledge regarding the $k-1$ other nodes. This greatly reduces the communication requirements between the nodes (even if it was already the case for our system). So the only thing that needs a node is the encoding vector of the node that has failed.

But, as we are using Reed-Solomon code then the Generator Matrices $G_1$ and $G_2$ of the codes $C_1$ and $C_2$ are Vandermonde Matrices, which means that each encoding vector can be described by only one element from the Finite Field $\mathbb{F}_q$ (since every element of this vector is a power of this element).

By doing so, we can then limit the communication between the nodes.

- **Twin Codes have also a good property regarding the distribution of the data.**

To understand that, we first need to come back on the fact that to repair a node the Minimum Bandwidth Exact Regenerating code needs $n - 1$ remaining nodes. But in the case of the Twin Codes, we only need $k$ nodes of the other types.

So when we want to distribute the data, we do not need to send it to the $n = n_1 + n_2$ nodes but instead we transmit the data to $k$ nodes of type $T_1$, then every node of type $T_2$ are treated as if they were failed nodes so then can be repaired by the $k$ nodes of type $T_1$. And finally, we repair the $n_1 - k$ remaining nodes of type $T_1$ thanks to $k$ nodes of type $T_2$.

The consequence of this property is the fact that the traffic will be more uniform across the network: the client will not have to distribute a huge amount of information every time we want to add new data, it will send the data to only $k$ nodes and then the data will be resend to the other nodes. Thus, as the code used in our project, it minimizes the required bandwidth during the reparation of a node but in addition of this property, the amount of data transferred during the distribution process is reduced to the minimum possible and this is this property that we would like to add to our system.

- **More than 2 node types**

In this chapter we have explained how to create a distributed storage with two node types but nothing prevents us to use more than 2 types. In this case, the dimension of the message matrix will have to be chosen equal to the number of types). Even if the consequences of such a choice have not been studied, it could be an interesting way to improve a system.

Note that the fact that Twin Codes minimize both the Bandwidth and the Storage has not been discussed here since we had already detailed it in the presentation section.

### 6.4.2 What are the constraints of this framework ?

Obviously, this system can not only have advantages, otherwise we would have used it in our project instead of the Minimum Bandwidth Exact Regenerating code. And this subsection is here to explain them and compared them to our own system.

- **Twin Codes are less flexible**

Indeed, there are several that the twin codes framework force us to do: the first one is that the size of the data has to suit in a matrix which makes us choose: $B = k^2$. It is not so worrying since in the Minimum Bandwidth Exact Regenerating code the size has to be $B = \frac{k(n-1)-k(k-1)}{2}$ but it forces us to split the file in much more chunks since the size is smaller. And it can have an impact for very large files. The other problem which is more preoccupying is the fact that we need the nodes to do some computations. Indeed, if we have a look back to the repairing node, we can see that we need to compute $g_{(2,l)}M_2g'_{(1,h)}$ which is annoying since in the definition of our system the nodes are only here as storage devices. Of course, nothing prevents from changing this requirement but it is a supplementary constraint compared with the Minimum Bandwidth Exact Regenerating code.

- **Twin Codes needs $k$ nodes of one nodes type**

As said before, there is a main problem with this framework which is the fact to retrieve a file we need at least $k$ out of $n_1$ or $n_2$ nodes, which means that we need at least $k$ node of the same type. But, if out of the $n = n_1 + n_2$ original nodes we have only $k$ nodes there are very few chances for them to be from the node type but we would rather have something like for example $k/2$ nodes of type $T_1$ and $k/2$ nodes of type $T_2$ which is a problem that does not appear for the Minimum Bandwidth Exact Regenerating code.

### 6.4.3 How can it be include in the Minimum Bandwidth Exact Regenerating code ?

Now that we have seen the advantages and drawbacks of both system, we are now going to explain how we would have combined them if the project was given more time. And then we will detail what we think this modification could add to the existing systems.

#### 6.4.3.1 Combination of the two systems

The idea we had to combine these two systems comes from a simple remark: in the twin codes construction, we split the the original into a matrix of size $k * k$ and then we code every row of $k$ symbols by multiplying by a matrix to have the data of the first node type and then we do the same to have the data of the second node type.

In a way this is very similar to what we do in Minimum Bandwidth Exact Regenerating code: in order to get what is in the node we take a vector of size $\frac{(n-1)*k-k*(k-1)}{2}$ that contains all the data and to get the first symbol stores on every node, we multiply it by a matrix of size $\frac{(n-1)*k-k*(k-1)}{2} * n$ that only contains 0 and 1.

So this systems look like the twin codes except for the fact that every row depends on each data and data the matrix changes for every row we want to encode (it is more a family of matrix). So the question is to know if we can combine two constructions of the system. The answer to this question is yes since, a simple permutation of the distribution matrix $V$ detailed in 3.2.2 we can create a kind of twin codes system where every server of a type can rebuild the ones of the other type.

Note that this idea does not come from any reference and that it should be experimented before going into further conclusions.

#### 6.4.3.2 Expected Properties

There are several obvious properties that will be given to our system and that we are going to list in the rest of this section

- The first good property of this system is that we are still able to minimize the bandwidth when repairing a node (since we do not change the procedure for that).
- Thanks to the Twin Codes, we are able to minimize the data amount of storage on each node.
- Another advantage of a system such as described before is the fact that the matrix multiplication will only include multiplication by 1 or 0 which is much more efficient when we implement it on a machine and much faster on a CPU point of view.

  This property will have another consequence which is much more important and which is the fact that we will not need to have nodes able of arithmetic computation: using 0 and 1 we do not need them to compute the computation of the vector $g_{(2,l)}M_2g'_{(1,h)}$ (seen in 6.2.3) over the finite field (thing that was needed in the Twin Codes Framework).
- We also benefit of the data distribution of the twin codes in the way that, as explained before, we will now be able to send the original encoded information to only $k$ nodes rather than the $n_1 + n_2$ original and then consider the rest of them as node to repair.

  Note that it also means that we do not need $n - 1$ nodes to repair a failed one any more but just $k$ nodes of the other type. It is a big improvement compared to the Minimum Bandwidth Exact Repairing code.
- The last property, we would like our system to have compared with the two original ones is the fact that contrary to the Twin Codes, we need only $k$ out of $n = n_1 + n_2$ nodes to retrieve a file rather than $k$ out of $n_1$ or $n_2$ nodes which will be a great improvement for the original systems.

Even if we have more doubts regarding this last property, the other ones look very plausible and they seem a very good direction for further work in this area.

That being said, it will still be necessary to proceed in some tests (such as the one done in the Experiments and Results chapter) to for example know if we have the same performance regarding the bandwidth and the data storage. But also if the fact that only using 0 and 1 as explained in the third bullet point of this section will have an impact on the encoding performance.

It will also be necessary to take into account the impact of the different hardware components on the system. That way, we will be able to know different properties of our system such as the limiting hardware factor or the impact of the parameters given to our system.

Obviously, it will still be important (as it was in this project) to verify the feasibility of such a system which is so far only theoretical.

# Chapter 7

# Conclusion

In this conclusion, we are going to summary everything that has been done during this project and explained in this document in order to see what remains to do as a future work.

In this project we have studied how to create a finite field. Then we have seen how to use it in order to create an erasures code and more particularly a Reed-Solomon code. Thanks to that we have finally been able to introduce the use of erasure codes in distributed storage. It has allowed us to study a new kind of system: Minimum Bandwidth Exact Regenerating code. At this stage, the question was to know if such a system that was based on beautiful theories but still unimplemented was feasible in real conditions.

The implementation of such a system was then detailed in chapter 4. We are well aware that there always is a better implementation but this one has the merit to show that such a system is feasible and thanks to it we were able to run experiments that lead us to some interesting results. First and most obvious one is that such a system is feasible and could be used under real circumstances. We learned that the CPU was not really important in this kind of system even if some operations could seem time consuming. We have also highlighted the fact that neither uploading bandwidth nor downloading bandwidth should be neglected. And maybe the most unexpected conculsion, that the fragmentation of the disk has a huge impact on the performance.

Finally, we introduced another kind of distributed storage system based on twin codes thanks to which we have proposed a new framework based on both the Twin Codes and the Minimum Bandwidth Exact Regenerating code.

Obviously such a subject is too big for only one project and the points detailed previously were the one on which we choose to focus on but there are still some points that will be worth a lot of interests in a future work. In order to list these points we are going to distinguish what stays to be done from an implementation point of view and then from a theoretical point a view.

Regarding the implementation detailed in chapter 4, we have for example seen that the cache used was a write-back policy and we could be interesting to implement a write-trough policy in order to compare the impact on our system. Parallelization of algorithm such as the encoding and decoding parts of the system could also be envisaged.

Another point that would be interesting to develop in the future is based on the fact that in this implementation we chose to separate the repairing function from the other functions of the system. This choice was made in order to simplify the user work since he can manually choose when and how to repair a node but it could be interesting to work in that direction and merge the two parts of the implementation in order for the system to detect and repair automatically the failed node. This decision could have big impacts on the performance of the system and namely on the CPU consumption.

An important part of Distributed Storage system has not been developed in this project: security management. Processes such has authentification, confidentiality and data encryption have an impact on the performance and should be taken into account in such an implementation.

To end with the implementation part, we can notice that other tests could have been done by mixing the subsections of chapter 5: it could have been interesting to have some knowledge of the impact on the system of the cache size when the bandwidth is limited. Another example could have been to understand the influence of the CPU when the cache size is limited.

Let's now conclude by a more theoretical opening: in the last chapter we propose a new framework which has stayed at a draft state and it is worth to look further in this direction namely to know if the conjunctions that we made appear to be plausible or not. In any case, many areas for future improvement have been highlighted by our approach and the association of several systems are not to be excluded as an improvement of existing systems.

# Appendix A

# Settings Example

```
n_param=7
k_param=3
block_param=4096
cache_param=0
0.dev=/dev/loop1
0.free_offset=0
0.dev_size=1048576000
1.dev=/dev/loop2
1.free_offset=0
1.dev_size=1048576000
2.dev=/dev/loop3
2.free_offset=0
2.dev_size=1048576000
3.dev=/dev/loop4
3.free_offset=0
3.dev_size=1048576000
4.dev=/dev/loop5
4.free_offset=0
4.dev_size=1048576000
5.dev=/dev/loop6
5.free_offset=0
5.dev_size=1048576000
6.dev=/dev/loop7
6.free_offset=0
6.dev_size=1048576000
```

In this example, we specify to the system that there are 7 available nodes and that we want to be able to recover from only 3. We also say that we want to use the cache by setting the `cache` option to 0.

From the 5th line to the end, we provide to the system the name of the disks that we want to use, the number of the first available byte and finally the size of the disks in bytes.

Here we are using disks of 1 GB that are considered as empty so the offset begins at 0.

All this information have to be put in the file named `settings` in the current folder of the software and you can find how to use these parameters as well as the ones rom appendix $B$ in appendix $C$.

# Appendix B

# Status Example

The following example represents a set of nodes where none of them have failed:

```
0
0
0
0
0
0
0
```

We can also use the following example where the 3 first nodes defined in the `settings` have failed:

```
1
1
1
0
0
0
0
```

These parameteres have to be set in the `status` file.

# Appendix C

# User's Guide

If one wants to test our software he can using the following procedure. Note that this user's guide is based on the settings and status examples provided in appendix *A* and *B*. Since it is based on FUSE, some steps to be followed are the same as in [11].

## C.1 Nodes Creation

As explained in chapter 4, we first need to create the IMG files that will be mounted on the `/dev/loop` devices. For that we can run the following commands:

```
dd if=/dev/zero of=file1.img bs=1M count=1000
dd if=/dev/zero of=file2.img bs=1M count=1000
dd if=/dev/zero of=file3.img bs=1M count=1000
dd if=/dev/zero of=file4.img bs=1M count=1000
dd if=/dev/zero of=file5.img bs=1M count=1000
dd if=/dev/zero of=file6.img bs=1M count=1000
dd if=/dev/zero of=file7.img bs=1M count=1000
```

Once they are created we need to verify that nothing is already mounted on the devices and then actually mount them by running the following commands:

```
losetup -d /dev/loop1
losetup /dev/loop1 file1.img

losetup -d /dev/loop2
losetup /dev/loop2 file2.img

losetup -d /dev/loop3
losetup /dev/loop3 file3.img

losetup -d /dev/loop4
losetup /dev/loop4 file4.img

losetup -d /dev/loop5
losetup /dev/loop5 file5.img

losetup -d /dev/loop6
losetup /dev/loop6 file6.img

losetup -d /dev/loop7
losetup /dev/loop7 file7.img
```

## C.2 Uploading and Downloading Files

Once this is done you can then run the sofware using:

```
./mbr -f <rootdir> <mountdir>
```

where `<mountdir>` is the folder where the user can put its file and `<rootdir>` is the place where the files stored are listed. Be careful to always put the `-f` since it is the only way to run FUSE in the foreground (we will come back on FUSE later in this document). You can also use `-d` to run the debugging mode of FUSE.

Assuming that you have created a file named `test` in the current folder of the software and we are using the first example of appendix $B$, you can run the following command in another terminal:

```
cksum test
cp test ./mountidr
ls -l ./mountdir
cp ./mountdir/test ..
cksum ../test
```

These commands will upload and download the `test` file to the nodes and the `cksum` tool will allow you to verify that the file downloaded is the same than the one uploaded.

## C.3 Simulation of node failures

If one wants to verifiy that the system can retrieve the file even if some nodes have failed, he needs to first following the previous steps. Then he stops the program and edit the `status` file as explained in the second example of appendix $B$.

In a first terminal, he needs to restart the program using the same command than before. He then runs the following commands in another terminal:

```
cp ./mountdir/test ..
cksum ../test
```

By doing so, he can verify that even if the first 3 nodes fail, he can still retrieve its data. He can then stop the software.

## C.4 Nodes repairing

Finally, if only one node has failed (you can simulate it by editing the `status` file and put a 1 on the first line and 0 on the 6 other lines), he can run the following commands to repair it:

```
dd if=/dev/zero of=file8.img bs=1M count=1000
./repairing file8.img
losetup -d /dev/loop1
losetup /dev/loop1 file8.img
```

The first line create the file on which the new node will be stored. Then we repair it using the `repairing` function. And finally we unmount the failed device to mount the new one. Once this is done, do not forget to edit the `status` file by putting 0 on every line.

# Bibliography

[1] A. Aizman and D. Yusupov. Open-iscsi. `http:://http://www.open-iscsi.org/`.

[2] E. Berlekamp. Algebraic coding theory. *Aegan Park Press*, 1984.

[3] E. R. Berlekamp. Factoring polynomials over finite fields. *Bell System Technical Journal*, 1967.

[4] S. BLOBIFY. Disque ssd. `http://www.disque-ssd.info/`, 2011.

[5] R. Charette. Google gmail's "lost emails" restored to life. `http://spectrum.ieee.org/riskfactor/telecom/internet/google-gmail-lost-emails-restored-to-life`, Mars 2011.

[6] A. G. Dimakis. Network coding for distributed system. `http://www.cs.uci.edu/bin/pdf/seminarseries2k9/Dimakis.pdf`, 2009.

[7] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *IEEE International Symposium on Information Theory*, April 2010.

[8] Robinson Enterprises. Welcome to chrysocome.net. `http://www.chrysocome.net/`.

[9] PlanetLab: An Overlay Testbed for Broad-Coverage Services. B. chun and d. culler and t. roscoe and a. bavier and l. peterson and m. wawrzoniak and m. bowman. *Newsletter ACM SIGCOMM Computer Communication Review*, 2003.

[10] T. Hansen and G. L. Mullen. Primite polynomials over finite fields. *Mathematics of Computation*, 1992.

[11] Yuchong Hu, Chiu-Man Yu, Yan Kit Li, Patrick P. C. Lee, and John C. S. Lui. Ncfs: On the practicality and extensibility of a network-coding-based distributed file system. *Proceedings of the 2011 International Symposium on Network Coding*, July 2011.

[12] B. Hubert. Howto du routage avance et du controle de trafic sous linux. `http://www.zone-h.fr/files/5/Routage_avance_et_controle_trafic_sous_linux.pdf`.

[13] M. S. Manasse, C. A. Thekkath, and A. Silverberg. A reed-solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage. *Microsoft Research*, 2009.

[14] A. Marletta. Cpu usage limiter for linux. `http://cpulimit.sourceforge.net/`.

[15] K. Z. Meth and J. Satran. Design of the iscsi protocol. *20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[16] W. Perkins. *Reed-Solomon Codes and Their Application.* Stephen B. Wicker Vijay K. Bhargava, 1994.

[17] W. Peterson. Encoding and error correction procedures for the bose-chaudhuri codes. *Transactions on Information Theory*, 1960.

[18] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact- regenerating codes for the msr and mbr points via a product-matrix construction. *IEEE Transactions on Information Theory*, 2011.

[19] K. V. Rashmi, Nihar B. Shah, and P. Vijay Kumar. Enabling node repair in any erasure code for distributed storage. *IEEE International Symposium on Information Theory*, June 2011.

[20] K. V. Rashmi, Nihar B. Shah, P. Vijay Kumar, and Kannan Ramchandran. Explicit construction of optimal exact regenerating codes for distributed storage. *Proc. IEEE Information Theory Workshop*, October 2009.

[21] I. Reed and G. Solomon. Polynomial codes over certain finite field. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[22] O. Rioul. Corps finis. `http://perso.telecom-paristech.fr/~rioul/documents/200609corpsfinis.pdf`, 2006.

[23] P. Rubin, D. MacKenzie, and S. Kemp. dd. *Linux Manual*, 2006.

[24] P. Rubin, D. MacKenzie, and S. Kemp. ionice. *Linux Manual*, 2006.

[25] P. Rubin, D. MacKenzie, and S. Kemp. losetup. *Linux Manual*, 2006.

[26] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran. Distributed storage codes with repair-by-transfer and non- achievability of interior points on the storage-bandwidth tradeoff. *IEEE Transactions on Information Theory*, 2011.

[27] H. Solomon, A. Kivity, P. Anderssons, and A. Liguory. Kernel based virtual machine. `http://www.linux-kvm.org/page/Main_Page`.

[28] C. Suh and K. Ramchandran. Exact-repair mds codes for distributed storage using interference alignment. *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2010.

[29] A. Tanenbaum and M. Van Steen. Distributed systems: Principles and paradigms. *Prentice Hall*, 2002.

[30] FUSE Project Team. Filesystem in userspace. `http:://www.http://fuse.sourceforge.net/`.

[31] FUSE Project Team. Hello world. `http:://www.http://sourceforge.net/apps/mediawiki/fuse/index.php?title=Hello_World`.

[32] F. Tomonori. tgt project. `http:://www.http://stgt.sourceforge.net/`.

[33] J.H. van limit. *Introduction to Coding Theory.* Springer, 1999.

[34] Wikipedia. Dirty cache. `http://www.en.wikipedia.org/wiki/Dirty_cache`.

[35] Y. Wu, A. G. Dimakis, and K. Ramchandran. Deterministic regenerating codes for distributed storage. *Proc. Allerton Conference on Control, Computing and Communication*, September 2007.