Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Comparing the Reed Solomon Code to Two Recently Found Codes for Distributed Storage

Thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

J. van der Krogt

Delft, The Netherlands

March 5, 2019

BSc thesis Applied Mathematics

"Comparing the Reed Solomon Code to Two Recently Found Codes for Distributed Storage"

J. van der Krogt

Delft University of Technology

Supervisor
Dr.ir. J.H. Weber

Thesis committee
Dr. J.L.A. Dubbeldam
Dr. J.G. Spandaw

March, 2019
Delft

Preface

This thesis is the final submission for the Bachelor program of Applied Mathematics at Delft University of Technology. The first time I learned about coding theory was a year ago in the course of applied algebra, a course about coding theory and cryptography. I initially choose the course because of one of my favorite movies of all time *Imitation Game*. This movie only covers cryptography, however, as I went on following the course my interest in coding theory grew even bigger, as I found the methods that were covered fascinating and since the field has a very current application. The field of coding theory is growing rapidly, articles are being published by the month and findings I studied are being implemented while this thesis is being written. This made me enthusiastic for working on this subject.

I would like to thank my supervisor, J. Weber, for his flexible guidance and for his feedback during the process of writing this thesis. I would like to thank Nils Dikken for designing figures 2 and 3, and at last I would like to thank the thesis committee for reading and reviewing this thesis.

Abstract

Erasure codes protect data from being lost as servers tend to fail because of various reasons. Currently Reed Solomon Codes are being used by multiple big companies, however, more promising codes have been described in recent articles. This report compares the (14,10)-RS code (which is among others being used by Facebook), the Piggybacked (14,10)-RS code and the Heptagon-Local Code in terms of storage overhead, reliability and the repair bandwidth. As with most coding methods, most of the times a trade-off is found between the methods. The Heptagon-Local Code has the greatest storage overhead, however, it can repair one failed server, much faster then the other methods.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Current situation and research . . . 1

1.3 Contributions and organizations of the thesis . . . 1

2 Reed Solomon codes **3**

2.1 Concepts of erasure coding . . . 3

2.2 Reed Solomon codes . . . 4

2.3 Decoding . . . 5

3 Piggybacking **7**

3.1 Encoding . . . 7

3.2 Decoding . . . 9

3.2.1 Systematic nodes . . . 9

3.2.2 Parity nodes . . . 10

3.3 Quality . . . 12

4 Multiple failures in a Piggybacked (14,10)-RS **14**

4.1 Two failures in the parity nodes . . . 14

4.2 Two failures in the systematic nodes. . . . 15

4.3 One failure in a systematic node and one failure in a parity node . . . 16

4.4 Total repair bandwidth of two failures . . . 17

4.5 Three and four failures . . . 17

5 Heptagon-Local Code **18**

5.1 Encoding . . . 18

5.2 Decoding . . . 19

5.2.1 One failure . . . 19

5.2.2 Two failures . . . 20

5.2.3 Three failures . . . 20

5.3 Quality . . . 21

5.4 Difference with 2-Replication . . . 22

6 Comparison **23**

6.1 Repair bandwidth and storage overhead comparison . . . 23

6.2 Changing the Heptagon-Local Code . . . 24

6.3 Changing the Piggybacked (14,10)-RS code . . . 24

6.3.1 Making a fair comparison . . . 24

6.3.2 Comparing the Piggybacked (21,10)-RS with the Piggybacked (14,10)-RS 25

7 Conclusion and future work **27**

7.1 Conclusion 27

7.2 Discussion and future work 27

A From basic erasure coding to Reed Solomon Codes **28**

A.1 An introduction to code design 28

A.2 Three ways to modify a code 31

A.3 Code words as polynomials 32

A.4 Cyclic codes 33

A.5 BCH codes 34

A.6 Reed Solomon codes 36

B Repair of two failed nodes in the same group after Piggybacking **38**

1 Introduction

We start this thesis off by discussing why one would want to use and design an erasure correcting code, then we will see what the problem is with the codes that are currently in use by a lot of big companies. To this end a motivation for this thesis will be given. At last we will walk through a list of what each chapter is about and what contributions are made in each chapter.

1.1 Motivation

In today's world the collecting and saving of data plays a tremendous role. A countless amount of companies try to collect data in order to predict future behavior of anything (e.g., customers, terrorist attacks, the weather). These companies store their data into servers which could all be located in the same building, but these servers can also be spread out over different data centers all over the world. In 2014 Google announced their plans for building a data center of 70 hectare in Groningen which at the moment is being expanded to become even bigger. Today Google possesses 15 different data centers, 4 of which are located across Northern Europe, 8 in the United States of America, 1 in Chili and 2 in Eastern Asia. All of these data centers are of course connected with each other to store your photos, your email, your browser and search history, your location and what not. For multiple reasons, sometimes a server node is unavailable. It can fail, it can be undergoing maintenance or it can be busy serving other data demanding purposes. Since the result is the same, namely unavailability of the data, each of which will from now on be called *node failure*.

Research concerning node failures at the Facebook warehouse cluster has been done in [1]. The authors found that a median value of 50 nodes per day that were unavailable for more than 15 minutes. This led to a median value of 180TB cross rack traffic generated per day because of these unavailable servers. Unfortunately the article says nothing about what percentage this is of the whole data center, however, we know that millions of dollars are spent in order to restore the data. It is clear that the need for a system that provides quick and cheap node repair is of great significance to companies like Facebook and Google.

1.2 Current situation and research

Currently most big IT-companies use Reed Solomon Codes to store their data with, as described in [2]. These codes perform great because they are reliable and for other reasons yet to be described, however, in order to repair a failed server a relatively great amount of data has to be downloaded each time. In [2] 8 recent and more promising methods are named. In this thesis we will be looking at two of these methods, Piggybacking and the Heptagon-Local Code, and compare them with the Reed Solomon Code currently in use. In this thesis we ask the question of if it is profitable for a big IT-company to switch from Reed Solomon Codes to one of these two recently found codes.

1.3 Contributions and organizations of the thesis

We now proceed to describe chapter wise what each chapter is about and what contributions are made in this chapter.

Chapter 2: Reed Solomon codes. This chapter provides the information about erasure coding and about Reed Solomon codes that the literature describes, which is essential for this thesis. First the parameters which we will use for the final comparison are introduced, followed by a discussion about when we can compare two codes. Then the definition of a Reed Solomon code is given, followed by the properties of *shortening* a Reed Solomon and at last we will combine erasure coding and Reed Solomon codes by showing how decoding of the (14,10)-Reed Solomon Code works. We will also explain why we choose this code. In Appendix A the same information is given, only in much more detail for the reader with little or no prior knowledge of Reed Solomon codes or erasure coding.

Chapter 3: Piggybacking. This chapter describes what the Piggybacking coding scheme is, as can be found in the literature, by first showing how the encoding algorithm works, followed by the decoding algorithm. Since in chapter 6 we will compare the (14,10)-Reed Solomon Code to the Piggybacked version of the (14,10)-Reed Solomon code, we will use this code to show how the algorithms apply to this example of a code. We will end the chapter with a short analysis of this Piggybacked (14,10)-Reed Solomon code that we can make based on the literature.

Chapter 4: Multiple failures in a Piggybacked (14,10)-RS. Since the literature only tells us about the repair and results of a single failure, we will see in this chapter if the (14,10)-Reed Solomon code also benefits from the Piggybacking scheme when multiple failures occur. Therefore we discuss the optimal repair method for each combination of two failures, followed by a calculation of the total repair bandwidth. Then we discuss a lower bound and an upper bound for 3 and 4 failures.

Chapter 5: Heptagon-Local Code. In this chapter our last code, the Heptagon-Local Code, is introduced and described based on the literature. Again first the encoding algorithm is given, followed by the decoding algorithm. Then the analysis provided by the literature concerning 1, 2 and 3 failures is described. We proceed by looking at if the Heptagon-Local Code can repair more than 3 failures. Since the Heptagon-Local Code uses 2-Replication, we close the chapter off by looking at what the difference between these two codes is and how the code benefits from the steps the Heptagon-Local Code encoding algorithm takes after the replication of the data.

Chapter 6: Comparison. In this chapter we summarize the found results of each code, which leads to a trade-off between the codes we compare. We ask ourselves if we can change the parameters of both codes in order to see if we can improve its results.

Chapter 7: Conclusion and future work. This chapter concludes the results that are found and gives a list of recommended research topics for the reader.

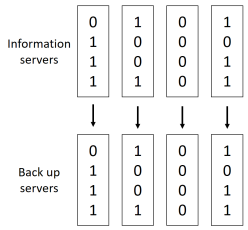
2 Reed Solomon codes

In this thesis we will discuss two recently found codes and we will compare them with the Reed Solomon Code, but before we can do so, we must first understand what erasure coding is and work our way to the definition of a Reed Solomon Code. In this chapter we will only discuss the essential properties needed for the following chapters. Readers with little knowledge about erasure coding or Reed Solomon Codes are referred to Appendix A for a more detailed and step wise explanation. Most results found in this chapter are derived from [3].

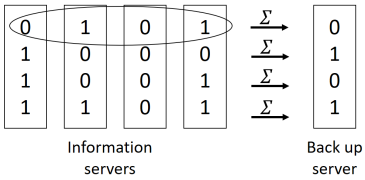
2.1 Concepts of erasure coding

In Chapter 6 we will compare multiple codes. We will begin this section by introducing the two parameters which we will use for this comparison, and we will proceed by defining when we can compare two codes in a fair way. We will discuss all this by using the two relatively simple codes of the next example.

Example 2.1.



(a) *A direct copy of every information server (often called 2-Replication).*



(b) *One parity server based on the sum of each row of bits of the information servers.*

Figure 1: *Two examples of relatively simple codes, coding a message of 16 symbols (here bits).*

The first code stores 32 symbols, which is twice as much as the original 16 symbols the data consisted of. The second code only stores 4 symbols extra. This has to do with the concept of *storage overhead*, which is defined as follows

Definition 2.2. *The storage overhead of a code is the total amount of data it stores divided by the amount of original data it stores.*

In the case of Example 2.1 the first code stores 32 symbols, where the original data was only 16 symbols, which makes the storage overhead 2. The second code stores 20 symbols, so the storage overhead here is only 1.25. This means that the second code scores better in terms of storage overhead. However, there is a trade-off between the two codes and that is where the second parameter has to be considered. When for instance the first information server fails, in the first code we only have to download the 4 symbols of the first back-up server in order to restore the lost information, where in the second code we need to download all remaining 16 symbols (and take the sum of each row). This is has to do with the *repair bandwidth*.

Definition 2.3. *The repair bandwidth is the amount of data that is needed to repair the system of servers. This number is given as a percentage of the size of the original data.*

When 1 server fails, in the first code we need to download the 4 symbols of the copy node when we want to repair this server, which makes the repair bandwidth $4/16 * 100\% = 25\%$. Since this

hold for any arbitrary server, we say that *the repair bandwidth for 1 failure is 25%*, which is not always the case as we will see after this example. For the second code we said that we need to download all remaining 16 symbols which makes the repair bandwidth 100% for 1 failure. This means that the first code scores better in terms of the repair bandwidth when only 1 server fails. When we now look at two failures something interesting happens. The information in the second code is then lost, but in the first code this depends on which combination of servers fail. When for instance node 1 and 2 fail we can simply recover the data from back-up servers 1 and 2. These servers together store 8 symbols, which make the repair bandwidth 50%. However, when server 1 and back-up server 1 fail, this information is lost, so there is no repair bandwidth. That is why, in some cases, we need to include some probability theory in the calculation of the repair bandwidth. We will explain this with an example, using Example 2.1.a.

Example 2.4. *When two failures happen in our code, we have a chance of* $4 * \binom{2}{2}/\binom{8}{2} = 1/7$ *of that being the combination of an original server and its corresponding parity server. In this case we cannot repair the system. In the other 6 out of 7 cases the repair bandwidth is 50%.*

In this code the combination of two failed nodes thus decides whether a system can be restored or not. However, in some codes we will discuss in this thesis, we will even see that different combination of failed servers (which can all be repaired) result in different repair bandwidths. When for instance 1 out of 7 cases mean (not that the data is lost, but) a repair bandwidth of 75%, and the other 6 out of 7 cases give a repair bandwidth of 50%. When a situation like this occurs, we take the weighted average of the repair bandwidths, which in this case is $\frac{1}{7} * 75\% + \frac{6}{7} * 50\% = 53.6\%$. We then say that *the repair bandwidth for 2 failures is 53.6%.*

The storage overhead and the repair bandwidth will in this thesis define the quality of a code. Now we only need to discuss what a fair comparison means. Like we said before, when 2 failures happen, the first code of Example 2.1 still has a chance of 6 out of 7 to be able to be repaired, where the data in second code would be forever lost. You could therefore say that the first code is more reliable. However, the first system of servers consist of 8 nodes, which makes the chance of two nodes failing bigger then when there are only 5 nodes like in the second code. To explain this we work out the situation of Example 2.1.

Example 2.5. *The repair of a random server takes a certain amount of time t, and every server has a certain chance of failing within that time span. This chance we will call* α *and it is of course equal for any server in the system. Then with a system of 8 servers where 1 is down, the chance that at least one other server failing within the repair time interval of size t is* $1 - (1 - \alpha)^7$*, where with a system of 5 servers that chance is* $1 - (1 - \alpha)^4$*. Since* α *is very small, it holds that* $1 - (1 - \alpha)^7 > 1 - (1 - \alpha)^4$*.*

The question arises if we can just compare a system of 5 nodes with a system of 8 nodes. In this case the answer is yes. Since our goal is to secure a certain amount of data, we say that we can compare any two codes in a fair way when they both secure the same amount of *original data*. In this case, both systems store 16 original symbols, which makes it a fair comparison.

2.2 Reed Solomon codes

Reed Solomon (RS) codes are used a lot in practice because of their clever construction. A significant difference to the codes we have discussed before, is that the symbols of a code word were singular bits, where in Reed Solomon codes this is not the case anymore. Reed Solomon codes are codes over the finite field $GF(p^r)$ with p prime, in this thesis $p = 2$,

and r a positive integer. When for instance we work over the field $GF(2^3)$, a symbol consists of 3 bits (which makes 8 possible symbols instead of 2) and all code words contain $2^3 - 1 = 7$ symbols. There are different ways of notating such a symbol, but in this thesis we will be doing that by powers of the primitive element β. This means that when for example we use a $GF(2^4)$, a word can look like this $\beta^8\beta^5 1000000000000$ which makes it the word 0111.1101.1000.0000.0000.0000.0000.0000.0000.0000.0000.0000.0000.0000.0000 in terms of bits. Again, a more detailed explanation can be found in Appendix A. We will now formally define binary RS codes and then we will look at the parameters of the code. We close the section off with a theorem about *shortening RS codes* (see Section A.2 for the definition of shortening). This is done a lot in practice.

Definition 2.6. *A binary Reed Solomon code $RS(2^r, d)$ is a cyclic linear code over $GF(2^r)$ with generator polynomial $g(x) = (\beta^{m+1} + x)(\beta^{m+2} + x)\dots(\beta^{m+d-1} + x)$ for $m \in \mathbb{N}$ and β a primitive element of $GF(2^r)$.*

We see that we can design the distance d of an RS code by choosing at what power of β we stop during building the generator polynomial. To list the parameters of an $RS(2^r, d)$ code:
$n = 2^r - 1$;
$k = 2^r - d$;
The amount of possible words $= 2^{rk}$.

So we see that by choosing r and d, the other parameters immediately follow. In practice RS codes are shortened a lot. We end this section by discussing how this affects the code. When we shorten the $RS(8, 5)$ of the example above. We know from the properties of shortening that both n and k decrease value by 1 where d keeps its value, resulting in $n = 6, k = 2, d = 5$. The amount of code words is still 2^{rk} which is now $2^{3*2} = 16$. Since shortening can be (and is often) done multiple times, we generalize shortening an RS code in the next theorem.

Theorem 2.7. *Shortening an $RS(2^r, d)$ code s times makes the original (n,k,d)-code map to an (n-s, k-s, d)-code. More specifically:*
$n = 2^r - 1 - s$;
$k = 2^r - d - s$;
The amount of possible code words $= 2^{rk}$.

This makes it possible for any Reed Solomon Code in this thesis to exist. For example, if we take the $RS(2^4, 4)$ code and shorten it 1 time, we get a $(n = 14, k = 10, d = 4)$-RS code, notated as the (14,10)-RS code. This is a code we will see a lot in this thesis.

2.3 Decoding

In the next chapters we will look at the (14,10)-RS code because it is used by Facebook as explained later on. The (14,10)-RS code has the property that it can repair any 4 failed nodes, which is quite a lot. The problem with the Reed Solomon Code clearly is not the reliability. The downside of the Reed Solomon Code is however the repair bandwidth, this we will explain in this section. We first introduce some notation. We will from now on notate a systematic symbol (a symbol of the original message) as a_i where in this (14,10)-RS code we have that $i \in \{1, 2, ..., 10\}$. The vector of all systematic symbols will be notated as $\mathbf{a}$. For simplicity we will always be looking at the systematic version of the code. The (14,10)-RS coding scheme adds another 4 symbols by mapping the original 10 symbols in 4 different ways to a new symbol.

Therefore we can look at this the following way: the first parity symbol is $\mathbf{p}_1^T\mathbf{a}$, the second symbol is $\mathbf{p}_2^T\mathbf{a}$, and so on. In other words, there are 4 vectors $\mathbf{p}_i$ with which we multiply our message vector $\mathbf{a}$ in order to get our 4 parity symbols. We call these $\mathbf{p}$-vectors the coding vectors. This can all be summarized in the following table.

Node 1	a_1
$\vdots$	$\vdots$
Node 10	a_{10}
Node 11	$\mathbf{p}_1^T\mathbf{a}$
Node 12	$\mathbf{p}_2^T\mathbf{a}$
Node 13	$\mathbf{p}_3^T\mathbf{a}$
Node 14	$\mathbf{p}_4^T\mathbf{a}$

Table 1: *The result of coding a message of 10 symbols, according to the (14,10)-RS coding scheme.*

Now we said that the downside of Reed Solomon lies in the repair bandwidth. That is because when a systematic node fails, we need to download a parity symbol, together with the remaining systematic symbols in order to subtract these from this parity symbol and be left with the (coded) missing symbol. This means we have to download 10 symbols, which makes the repair bandwidth 100%. The same goes for 2, 3 and 4 failures, with only an extra parity symbol for each extra failure. Also, when a parity server fails, we need to download all systematic symbols in order to calculate the parity symbol again. There is no other way since a parity symbol can not be deduces from other parity symbols. Due to its coding scheme, the repair bandwidth of a Reed Solomon Code is always 100%. In the next chapter we will look at Piggybacking, a method that is added to the RS coding scheme (or any coding scheme in general) with as its goal to decrease the repair bandwidth.

3 Piggybacking

Piggybacking is a method of encoding and decoding an existing code (like RS) with as goal to decrease the repair bandwidth. It was described in [4], we will summarize the information that we need and apply this to our own example of the (14,10)-RS code. This RS code is among others in use by Facebook's f4 BLOB storage system [5]. Then we will discuss the decoding principal and to close off we will review the results of the Piggybacking scheme. The general idea of Piggybacking is to make the parity servers mathematically dependent of more groups of information servers, so that more information can be traced back from one parity server. This may sound vague, that is why in this chapter we will discuss the coding and the decoding algorithm step by step, and we will discuss our example at the same time next to it. Then we will look at the piggybacking of parity nodes together with the decoding results. We discuss the results that these algorithms deliver in the end of this chapter. In [4] three ways of Piggybacking are described. We will be looking at Piggybacking 2, since from [4] Piggybacking 2 seems to be the most promising method. We will simply be calling it Piggybacking.

3.1 Encoding

We will be looking at the encoding algorithm of Piggybacking in this section by looking at the general algorithm and an example of how a (systematic) (14,10)-RS code is encoded. We will do this step by step by first looking at how the concerned step is taken in general, followed by how this step is applied in our example code.

General step 1. Say $r = n - k$. We consider $(2r - 3)$ instances of the base code and let $\mathbf{a}_1$,...,$\mathbf{a}_{2r-3}$ be the messages associated to the corresponding instances. We will divide the k systematic nodes into $(r - 1)$ equal sets $S_1, ..., S_{(r-1)}$. In the case that k is not a multiple of $(r - 1)$, we make nearly equal sets as we will see in this example step.

Example step 1. For our (14,10)-code we know that $n = 14$, $k = 10$ and $r = n - k = 4$. This means we have $2 * 4 - 3 = 5$ instances $\mathbf{a}_1$, $\mathbf{a}_2$, $\mathbf{a}_3$, $\mathbf{a}_4$ and $\mathbf{a}_5$. Since 10 is not a multiple of $4-1 = 3$ we divide the systematic nodes into 3 sets: S_1 consisting of nodes 1 to 4, S_2 consisting of nodes 5 to 7 and S_3 of nodes 8 to 10. The 50 RS-coded symbols and the 20 parity symbols initially look like the situation in Table 2.

General step 2. We define the k-length vectors $\{\mathbf{q}_{i,g}\}_{i=2,g=1}^{r,r-1}$ as

$$\mathbf{q}_{i,g} = \begin{bmatrix} 0 & & & & & & & & & \\ & \ddots & & & & & & & & \\ & & 0 & & & & & & & \\ & & & 1 & & & & & & \\ & & & & \ddots & & & & & \\ & & & & & 1 & & & & \\ & & & & & & 0 & & & \\ & & & & & & & \ddots & & \\ & & & & & & & & 0 & \end{bmatrix} \mathbf{p}_i$$

where the positions of the ones in the diagonal of the $(k \times k)$ diagonal matrix depicted corresponds to the nodes in the S_g-th group. Further, we define the k-length vectors$\{\mathbf{v}_i,\mathbf{v}_i^*\}_{i=2}^{r}$ as

$$\begin{aligned} \mathbf{v}_i &= \mathbf{a}_{r-1} + i\mathbf{a}_{r-2} + ... + i^{r-2}\mathbf{a}_1 \\ \mathbf{v}_i^* &= \mathbf{v}_i - \mathbf{a}_{r-1} = i\mathbf{a}_{r-2} + ... + i^{r-2}\mathbf{a}_1 \end{aligned}$$

S_1 Node 1	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
S_1 Node 2	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
S_1 Node 3	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$
S_1 Node 4	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$
S_2 Node 5	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$
S_2 Node 6	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$
S_2 Node 7	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$
S_3 Node 8	$a_{8,1}$	$a_{8,2}$	$a_{8,3}$	$a_{8,4}$	$a_{8,5}$
S_3 Node 9	$a_{9,1}$	$a_{9,2}$	$a_{9,3}$	$a_{9,4}$	$a_{9,5}$
S_3 Node 10	$a_{10,1}$	$a_{10,2}$	$a_{10,3}$	$a_{10,4}$	$a_{10,5}$
Node 11	$\mathbf{p}_1^T\mathbf{a}_1$	$\mathbf{p}_1^T\mathbf{a}_2$	$\mathbf{p}_1^T\mathbf{a}_3$	$\mathbf{p}_1^T\mathbf{a}_4$	$\mathbf{p}_1^T\mathbf{a}_5$
Node 12	$\mathbf{p}_2^T\mathbf{a}_1$	$\mathbf{p}_2^T\mathbf{a}_2$	$\mathbf{p}_2^T\mathbf{a}_3$	$\mathbf{p}_2^T\mathbf{a}_4$	$\mathbf{p}_2^T\mathbf{a}_5$
Node 13	$\mathbf{p}_3^T\mathbf{a}_1$	$\mathbf{p}_3^T\mathbf{a}_2$	$\mathbf{p}_3^T\mathbf{a}_3$	$\mathbf{p}_3^T\mathbf{a}_4$	$\mathbf{p}_3^T\mathbf{a}_5$
Node 14	$\mathbf{p}_4^T\mathbf{a}_1$	$\mathbf{p}_4^T\mathbf{a}_2$	$\mathbf{p}_4^T\mathbf{a}_3$	$\mathbf{p}_4^T\mathbf{a}_4$	$\mathbf{p}_4^T\mathbf{a}_5$

Table 2: *50 original symbols initially encoded according to the RS scheme. Here $a_{i,j}$ stands for the symbol in the i^{th} node and the j^{th} instance. The vector $\boldsymbol{p}_i$ is coding vector corresponding to the base function (in this case the Reed Solomon encoding scheme) of the i^{th} parity server. The vector $\boldsymbol{a}_j$ is the message vector consisting of all systematic symbols of the j^{th} instance.*

Example step 2. We first define the vectors $\mathbf{q}$:

$$
\begin{array}{lcccccccccc}
\mathbf{q}_{2,1} & = & [p_{2,1} & \dots & p_{2,4} & 0 & \dots & 0 & 0 & \dots & 0] \\
\mathbf{q}_{2,2} & = & [0 & \dots & 0 & p_{2,5} & \dots & p_{2,7} & 0 & \dots & 0] \\
\mathbf{q}_{2,3} & = & [0 & \dots & 0 & 0 & \dots & 0 & p_{2,8} & \dots & p_{2,10}] \\
 & \vdots & & & & & & & & & \\
\mathbf{q}_{4,1} & = & [p_{4,1} & \dots & p_{4,4} & 0 & \dots & 0 & 0 & \dots & 0] \\
\mathbf{q}_{4,2} & = & [0 & \dots & 0 & p_{4,5} & \dots & p_{4,7} & 0 & \dots & 0] \\
\mathbf{q}_{4,3} & = & [0 & \dots & 0 & 0 & \dots & 0 & p_{4,8} & \dots & p_{4,10}\,]
\end{array}
$$

Followed by the vectors $\{\mathbf{v}_i,\mathbf{v}_i^*\}$:

$$
\begin{aligned}
\mathbf{v}_2 &= \mathbf{a}_3 + 2\mathbf{a}_2 + 4\mathbf{a}_1 \\
\mathbf{v}_2^* &= \mathbf{v}_2 - \mathbf{a}_3 = 2\mathbf{a}_2 + 4\mathbf{a}_1 \\
\mathbf{v}_3 &= \mathbf{a}_3 + 3\mathbf{a}_2 + 9\mathbf{a}_1 \\
\mathbf{v}_3^* &= \mathbf{v}_3 - \mathbf{a}_3 = 3\mathbf{a}_2 + 9\mathbf{a}_1 \\
\mathbf{v}_4 &= \mathbf{a}_3 + 4\mathbf{a}_2 + 16\mathbf{a}_1 \\
\mathbf{v}_4^* &= \mathbf{v}_4 - \mathbf{a}_3 = 4\mathbf{a}_2 + 16\mathbf{a}_1
\end{aligned}
$$

General step 3. We Piggyback the base code in the following two parts.
Part 1:

Example step 3. The first $(r-2=2)$ instances of the nodes stay the same and the last three instances are transformed like described above. Note that in node 12 we have that $i-2=0$ and $r-i=2$, in node 13 we have $i-2=1$ and $r-i=1$ and in node 14 we have $i-2=2$ and $r-i=0$. This gives us

The first $r-2$ instances of node $k+i$ are not changed, they remain looking like:

$\mathbf{p}_i^T\mathbf{a}_1$	...	$\mathbf{p}_i^T\mathbf{a}_{r-2}$

In the $r-1$-th instance we store the following information: $\mathbf{p}_i^T\mathbf{a}_{r-1} + \sum_{j=1,j\neq i-1}^{r-1}\mathbf{q}_{i,j}^T\mathbf{v}_i^*$

The next $i-2$ instances are Piggybacked like:

$\mathbf{p}_i^T\mathbf{a}_r + \mathbf{q}_{i,1}^T\mathbf{v}_i$	...	$\mathbf{p}_i^T\mathbf{a}_{r+i-3} + \mathbf{q}_{i,i-2}^T\mathbf{v}_i$

And the last $r-i$:

$\mathbf{p}_i^T\mathbf{a}_{r+i-2} + \mathbf{q}_{i,i}^T\mathbf{v}_i$	...	$\mathbf{p}_i^T\mathbf{a}_{2r-3} + \mathbf{q}_{i,r-1}^T\mathbf{v}_i$

Part 2 is subtracting the last $r-2$ instances from the $r-1$-th instance:

$\mathbf{q}_{i,i-1}^T\mathbf{a}_{r-1} - \sum_{j=r}^{2r-3}\mathbf{p}_i^T\mathbf{a}_j$

Node 1	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
⋮	⋮	⋮	⋮	⋮	⋮
Node 10	$a_{10,1}$	$a_{10,2}$	$a_{10,3}$	$a_{10,4}$	$a_{10,5}$
Node 11	$\mathbf{p}_1^T\mathbf{a}_1$	$\mathbf{p}_1^T\mathbf{a}_2$	$\mathbf{p}_1^T\mathbf{a}_3$	$\mathbf{p}_1^T\mathbf{a}_4$	$\mathbf{p}_1^T\mathbf{a}_5$
Node 12	$\mathbf{p}_2^T\mathbf{a}_1$	$\mathbf{p}_2^T\mathbf{a}_2$	$\mathbf{p}_2^T\mathbf{a}_3 + \sum_{g=1,g\neq 1}^{3}\mathbf{q}_{2,j}^T\mathbf{v}_2^*$	$\mathbf{p}_2^T\mathbf{a}_4 + \mathbf{q}_{2,2}^T\mathbf{v}_2$	$\mathbf{p}_2^T\mathbf{a}_5 + \mathbf{q}_{2,3}^T\mathbf{v}_2$
Node 13	$\mathbf{p}_3^T\mathbf{a}_1$	$\mathbf{p}_3^T\mathbf{a}_2$	$\mathbf{p}_3^T\mathbf{a}_3 + \sum_{g=1,g\neq 2}^{3}\mathbf{q}_{3,j}^T\mathbf{v}_3^*$	$\mathbf{p}_3^T\mathbf{a}_4 + \mathbf{q}_{3,1}^T\mathbf{v}_3$	$\mathbf{p}_3^T\mathbf{a}_5 + \mathbf{q}_{3,3}^T\mathbf{v}_3$
Node 14	$\mathbf{p}_4^T\mathbf{a}_1$	$\mathbf{p}_4^T\mathbf{a}_2$	$\mathbf{p}_4^T\mathbf{a}_3 + \sum_{g=1,g\neq 3}^{3}\mathbf{q}_{4,j}^T\mathbf{v}_4^*$	$\mathbf{p}_4^T\mathbf{a}_4 + \mathbf{q}_{4,1}^T\mathbf{v}_4$	$\mathbf{p}_4^T\mathbf{a}_5 + \mathbf{q}_{4,2}^T\mathbf{v}_4$

Applying the second step of this algorithm gives us

Node 1	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
⋮	⋮	⋮	⋮	⋮	⋮
Node 10	$a_{10,1}$	$a_{10,2}$	$a_{10,3}$	$a_{10,4}$	$a_{10,5}$
Node 11	$\mathbf{p}_1^T\mathbf{a}_1$	$\mathbf{p}_1^T\mathbf{a}_2$	$\mathbf{p}_1^T\mathbf{a}_3$	$\mathbf{p}_1^T\mathbf{a}_4$	$\mathbf{p}_1^T\mathbf{a}_5$
Node 12	$\mathbf{p}_2^T\mathbf{a}_1$	$\mathbf{p}_2^T\mathbf{a}_2$	$\mathbf{q}_{2,1}^T\mathbf{a}_3$ - $(\mathbf{p}_2^T\mathbf{a}_4 + \mathbf{p}_2^T\mathbf{a}_5)$	$\mathbf{p}_2^T\mathbf{a}_4 + \mathbf{q}_{2,2}^T\mathbf{v}_2$	$\mathbf{p}_2^T\mathbf{a}_5 + \mathbf{q}_{2,3}^T\mathbf{v}_2$
Node 13	$\mathbf{p}_3^T\mathbf{a}_1$	$\mathbf{p}_3^T\mathbf{a}_2$	$\mathbf{q}_{3,2}^T\mathbf{a}_3$ - $(\mathbf{p}_3^T\mathbf{a}_4 + \mathbf{p}_3^T\mathbf{a}_5)$	$\mathbf{p}_3^T\mathbf{a}_4 + \mathbf{q}_{3,1}^T\mathbf{v}_3$	$\mathbf{p}_3^T\mathbf{a}_5 + \mathbf{q}_{3,3}^T\mathbf{v}_3$
Node 14	$\mathbf{p}_4^T\mathbf{a}_1$	$\mathbf{p}_4^T\mathbf{a}_2$	$\mathbf{q}_{4,3}^T\mathbf{a}_3$ - $(\mathbf{p}_4^T\mathbf{a}_4 + \mathbf{p}_4^T\mathbf{a}_5)$	$\mathbf{p}_4^T\mathbf{a}_4 + \mathbf{q}_{4,1}^T\mathbf{v}_4$	$\mathbf{p}_4^T\mathbf{a}_5 + \mathbf{q}_{4,2}^T\mathbf{v}_4$

Table 3: *The result of 50 original symbols encoded according to the RS scheme, followed by the Piggybacking scheme.*

3.2 Decoding

3.2.1 Systematic nodes

In this section we will first be looking at the repair of the arbitrary systematic node l. We will do this in the general sense, and in terms of the example we Piggybacked above. After having done this, we will see how we can efficiently repair a failed parity node. The algorithm for a systematic nodes goes as follows:

General step 1. Download all data from the last $(r-2)$ instances in nodes $\{1,...,k+1\}\backslash\{l\}$. With this data we can recover the data from the concerning instances of node l.
Example step 1. We download the data from instances 4 and 5 in nodes $\{1,...,11\}\backslash\{l\}$. Now there are only 3 instances of node l left to recover.

General step 2. In the beginning of the encoding algorithm, we divided all systematic nodes into groups S_g. This had as a result that the data from node l was stored in $q_{i,g}$ for some g.

As we can see from the algorithm and the example above, every node $\{k+2, ..., k+r\}$ has precisely one symbol containing a $\mathbf{q}$ vector with a non-zero l^{th} component. We download these $r-1$ symbols. We subtract the components with $\{\mathbf{a}_r, ..., \mathbf{a}_{2r-3}\}$ out of these symbols.
Example step 2. Say our node $l = 7$. Then l is in S_2. We now have to download instance 4 from node 12, instance 3 from node 13, and instance 5 from node 14 because they contain the $\mathbf{q}_{i,2}$-vectors. From step 1 we know $\mathbf{p}_4^T\mathbf{a}_4$, so we subtract this out of the first symbol we just downloaded. We do the same for the other two symbols. We are now left with $\mathbf{q}_{2,2}^T\mathbf{v}_2$, $\mathbf{q}_{3,2}^T\mathbf{a}_3$ and $\mathbf{q}_{4,2}^T\mathbf{v}_4$.

General step 3. We download all symbols from the systematic nodes that are in the same group as l. We subtract these out of the previously downloaded symbols. We are now left with $r-1$ independent linear combinations of $\{\mathbf{a}_{1,l}, ..., \mathbf{a}_{r-1,l}\}$. From this we can decode our erased data.
Example step 3. We continue with our situation where $l = 7$. We now download the symbols in instances 1, 2 and 3 from nodes 5 and 6. We subtract all factors with $\{a_{i,j}\}_{i=1,j=5}^{3,6}$. We are left with

$$\begin{cases} p_{2,7}(a_{3,7} + 2a_{2,7} + 4a_{1,7}) \\ p_{3,7}a_{3,7} \\ p_{4,7}(a_{3,7} + 4a_{2,7} + 16a_{1,7}) \end{cases}$$

where we know what the p elements are. This means we can decode the last three symbols, with only having downloaded 29 symbols in total.

Node 1	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$		
Node 2	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		
Node 3	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$		
Node 4	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$		
Node 5					
Node 6					
Node 7	x	x	x	x	x
Node 8	$a_{8,1}$	$a_{8,2}$	$a_{8,3}$		
Node 9	$a_{9,1}$	$a_{9,2}$	$a_{9,3}$		
Node 10	$a_{10,1}$	$a_{10,2}$	$a_{10,3}$		
Node 11	$\mathbf{p}_1^T\mathbf{a}_1$	$\mathbf{p}_1^T\mathbf{a}_2$	$\mathbf{p}_1^T\mathbf{a}_3$	$\mathbf{p}_1^T\mathbf{a}_4$	$\mathbf{p}_1^T\mathbf{a}_5$
Node 12	$\mathbf{p}_2^T\mathbf{a}_1$	$\mathbf{p}_2^T\mathbf{a}_2$	$\mathbf{q}_{2,1}^T\mathbf{a}_3$ - $(\mathbf{p}_2^T\mathbf{a}_4 + \mathbf{p}_2^T\mathbf{a}_5)$		$\mathbf{p}_2^T\mathbf{a}_5 + \mathbf{q}_{2,3}^T\mathbf{v}_2$
Node 13	$\mathbf{p}_3^T\mathbf{a}_1$	$\mathbf{p}_3^T\mathbf{a}_2$		$\mathbf{p}_3^T\mathbf{a}_4 + \mathbf{q}_{3,1}^T\mathbf{v}_3$	$\mathbf{p}_3^T\mathbf{a}_5 + \mathbf{q}_{3,3}^T\mathbf{v}_3$
Node 14	$\mathbf{p}_4^T\mathbf{a}_1$	$\mathbf{p}_4^T\mathbf{a}_2$	$\mathbf{q}_{4,3}^T\mathbf{a}_3$ - $(\mathbf{p}_4^T\mathbf{a}_4 + \mathbf{p}_4^T\mathbf{a}_5)$	$\mathbf{p}_4^T\mathbf{a}_4 + \mathbf{q}_{4,1}^T\mathbf{v}_4$	

Table 4: *An overview of the 29 symbols we downloaded. An empty entry represents a symbol we have downloaded and an x marks a missing symbol that we were trying to restore.*

3.2.2 Parity nodes

We only discussed the repair of a systematic node so far, so we now will move on to the repair of a parity node. Now note that in the previous subsection we did not use the parity symbols from the first $r-1(=3$ in our example) instances of the first parity node (here node 11) for the repair of any systematic node. This means that we can modify these symbols without decreasing

the repair bandwidth for the repair of systematic nodes. Our goal of modifying these symbols will be to decrease the repair bandwidth for the repair of parity nodes, because now the repair bandwidth for parity nodes is still 100%. We will look at how we can do this in our example of the (14,10)-RS code.

We take 2 systems of piggybacked (14,10)-RS codes, both consisting of 5 instances and 14 nodes, where we will store those 2 times 5 instances in the same 14 nodes. For simplicity we will notate the parity symbols by their location (e.g., the symbol in node 12, instance 3 will be notated as (12,3)). This gives us the following situation:

Node 1	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$	$a_{1,8}$	$a_{1,9}$	$a_{1,10}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Node 10	$a_{10,1}$	$a_{10,2}$	$a_{10,3}$	$a_{10,4}$	$a_{10,5}$	$a_{10,6}$	$a_{10,7}$	$a_{10,8}$	$a_{10,9}$	$a_{10,10}$
Node 11	(11,1)	(11,2)	(11,3)	(11,4)	(11,5)	(11,6)	(11,7)	(11,8)	(11,9)	(11,10)
Node 12	(12,1)	(12,2)	(12,3)	(12,4)	(12,5)	(12,6)	(12,7)	(12,8)	(12,9)	(12,10)
Node 13	(13,1)	(13,2)	(13,3)	(13,4)	(13,5)	(13,6)	(13,7)	(13,8)	(13,9)	(13,10)
Node 14	(14,1)	(14,2)	(14,3)	(14,4)	(14,5)	(14,6)	(14,7)	(14,8)	(14,9)	(14,10)

Table 5: *A schematic display of the result of Piggybacking the (14,10)-RS code. Two separate systems of 50 original symbols are stored next to each other and the parity symbols are named after their location.*

The Piggybacking now means that we take the sum of (12,4), (13,4) and (14,4) and add this to (11,6) and the same goes for instance 5 to 7 and for instance 3 to 8. It may look like the order of which instance piggybacks to which instance is a bit strange, but it is actually not. We will see this later on. We now have the following situation.

...	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$	$a_{1,8}$	...
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
...	$a_{10,3}$	$a_{10,4}$	$a_{10,5}$	$a_{10,6}$	$a_{10,7}$	$a_{10,8}$	...
...	(11,3)	(11,4)	(11,5)	(11,6) $+\sum_{i=12}^{14}(i,4)$	(11,7) $+\sum_{i=12}^{14}(i,5)$	(11,8) $+\sum_{i=12}^{14}(i,3)$	...
...	(12,3)	(12,4)	(12,5)	(12,6)	(12,7)	(12,8)	...
...	(13,3)	(13,4)	(13,5)	(13,6)	(13,7)	(13,8)	...
...	(14,3)	(14,4)	(14,5)	(14,6)	(14,7)	(14,8)	...

Table 6: *The result of Piggybacking the parity nodes of the first system onto the parity nodes of the second system.*

We can now look at the repair scheme. When node 11 fails we still need to download all systematic symbols (repair bandwidth is 100%). When node 12 fails we download the systematic symbols of instance 1 and 2 for the recovery of (12,1) and (12,2). We do the same for instance 6 to 10 for the recovery of the parity symbols in these instances. Now, for the recovery of (12,3) we only need to download the piggybacked (11,8) $+\sum_{i=12}^{14}(i,3)$ symbol, together with (13,3) and (14,3). We subtract all known symbols and are left with (12,3). We recover (12,4) and (12,5) in a similar way. Thus, the important improvement we made is that we do not need the systematic symbols of instances 3, 4 and 5. We now have downloaded only 70 systematic symbols together with 9 parity symbols. This makes 79 instead of 100, meaning a repair bandwidth of 79%. Since the repair of nodes 12, 13 and 14 are similar, the repair bandwidth for parity nodes in 1 failure is now $\frac{1}{4} * 100\% + \frac{3}{4} * 79\% = 84.25\%$.

The question arises if adding more systems to each other than just the 2 we just did, leads to an even lower repair bandwidth. The answer is yes. When we look for example at 3 systems, we can also piggyback instances 9 and 10 to instances 11 and 12, like we did with 4 and 5 to 6 and 7. The only important difference now is that we cannot piggyback instance 8 to instance 13 like we did with 3 to 8. This is because, like we have seen above, when we piggybacked instance 3 to 8, we saved downloading the 10 systematic symbols from instance 3. We however needed the systematic symbols of instance 8, to subtract these from the piggybacked (11,8) $+\sum_{i=12}^{14}(i,3)$ for the repair of (12,3). Since we already need the systematic symbols of instance 8, there is no use in trying to piggyback the parity node symbols of instance 8 to another instance. In general, we can always piggyback the 4^{th} and 5^{th} instance of a system to the first and the second of the subsequent system, but we can only piggyback the third instance of each odd system to the third instance of the subsequent even system. Since the repair bandwidth apparently depends on the number of systems we add together, we want to know what the repair bandwidth for a single failed parity node is when we add m systems together. In [4] a formula is given for the repair bandwidth in the general case. When we take this formula with the fact that in our case $r = 4$ and $k = 10$, we get

$$\text{Repair bandwidth for 1 failed parity node} = \left(25 + \frac{97.5m + 42}{2m}\right)\% \tag{1}$$

From this, together with the fact that the repair bandwidth for node 11 is always 100%, we can deduce that the repair bandwidths for nodes 12, 13 and 14 are all

$$\text{Repair bandwidth for node 12} = (28/m + 65)\% \tag{2}$$

We will look at the impact of m on the repair bandwidth of the code in total in the next section.

3.3 Quality

From the previous example we see that the storage overhead of the Piggybacked (14,10)-RS code remains $14/10 = 1.4$. Remember from Section 2.3 that with Reed Solomon the repair bandwidth is always 100%, which in this case means that we would need to download 50 symbols per system when a node fails. In the previous section we saw that when node 7 failed we needed just 29 symbols instead of 50. Note that when one of the first four nodes fails, we need to download three symbols extra to restore the failed node. So with 4 out of 10 systematic nodes we reduce the amount of data download (and therefore the repair bandwidth) with 36%, and with 6 out of 10 systematic nodes we reduce it with 42%. That is an average repair bandwidth of 60.4% for one failed systematic node instead of the 100% we would have had by using RS. We also saw that for 1 failed parity node, our repair bandwidth depends on how many systems m we combine. We will now look at what impact m has on the repair bandwidth of node 12 (and thus 13 and 14), on the group of parity nodes, and on the code in total in the following table. For the calculations of node 12 and the group of parity nodes we will use Equations 1 and 2, and for the code in total we will use Equation 1 where this is the average repair bandwidth for 4 out of 14 nodes with 10 out of 14 nodes having an average repair bandwidth of 60.4%.

	m=2	**m=10**	**m=100**	**lim$_{m\to\infty}$**
Node 12	79%	67.8%	65.28%	65%
Parity nodes	84.25%	75.85%	73.96%	73.75%
Piggybacked (14,10)-RS code	67.21%	64.81%	64.27%	64.21%

Table 7: *The effect of increasing m on the repair bandwidth.*

We see that increasing m does not change much when it comes to the total repair bandwidth of the Piggybacked (14,10)-RS code. For simplicity of the next chapters we will for the rest of this thesis take that $m = 2$, and therefore from now on the Piggybacked (14,10)-RS code has a repair bandwidth for 1 failure of 67.21%. Since nothing about multiple failures is described in the literature, we will look at that in the next chapter.

4 Multiple failures in a Piggybacked (14,10)-RS

When looking at 2 failures happening at once, we have a couple of options. The two failures might both happen in systematic nodes, or they might both happen in the parity nodes. Also, we could have one failure in a systematic node and one in a parity node. We proof for all three possibilities what is the lowest possible repair bandwidth and then we proceed by giving a lower bound and an upper bound for the repair bandwidth of 3 and 4 failures.

4.1 Two failures in the parity nodes

If multiple failures happen in only the parity nodes, we can always repair this because we created these nodes from only the systematic nodes. Moreover, we can repair up to 4 failed parity nodes. However, if we would use only the systematic symbols, we would always have to download all systematic symbols for repair, which means a repair bandwidth of 100%. To this end we piggybacked the parity nodes in the previous chapter, which made it possible to do a more efficient repair. In this section we ask ourselves if this piggyback design also decreases the repair bandwidth for two failures. Remember that we Piggybacked with $m = 2$ meaning that we Piggybacked with 2 systems like in Table 6. We know from the previous chapter that when node 11 fails, the repair bandwidth is always 100%. So we only have to check if we can find a more efficient repair when any combination of nodes 12, 13 and 14 fails. Our first claim is that when a single parity node 12, 13 or 14 fails, we have only 2 options for repair: we repair by downloading all systematic symbols, or we use the Piggybacked symbols in node 11 instance 6, 7 and 8.

Proof. Since we know from the previous chapter that we can repair more efficiently by using the piggybacked parity symbols, we will proof the claim by looking at the situation where we have not piggybacked the parity symbols, and proof that we can now only use the systematic symbols. We will do this for node 12, since 13 and 14 are identical.
We first look at the lost symbol of instance 1, which is $\mathbf{p}_2^T\mathbf{a}_1$ (see Table 3). We know that we cannot trace back $\mathbf{a}_1$ from for example $\mathbf{p}_3^T\mathbf{a}_1$, because the multiplication of the two vectors gave just 1 symbol. Thus, we cannot make $\mathbf{p}_2^T\mathbf{a}_1$ from $\mathbf{p}_3^T\mathbf{a}_1$ because these symbols are made from different vectors $\mathbf{p}_r$. Now that is important, because we can also see from Table 3 that when node 12 fails, all terms with (parts of) $\mathbf{p}_2$ are lost and only terms with the other $\mathbf{p}$-vectors are left. In other words, the piggybacking was done per node, and not across nodes, which means that symbols of a parity node are not stored in any other node. Therefore when node 12 fails, this information is completely lost and we cannot use other parity symbols to restore this. Thus, we need to download all systematic symbols. □

Now that the claim has been proven, we will look at 2 failures again. The claim has been proven for 1 failures, but when now 2 failures happen, we have even less symbols left, which means that we know that we still have at maximum the same 2 options for decoding. Since the first option gives a repair bandwidth of 100%, we will try to use the second option: using the Piggybacked parity symbols. We first observe that since the symbols in the failed nodes 12 and 13, instance 1, 2, 6, 7, 8, 9 and 10 are not piggybacked, we have to download all systematic symbols of these instances. This is 70 symbols, and the parity symbols of these instances are now all restored. Our only possibility of saving downloading symbols for the repair of node 12 and 13, instances 3, 4 and 5 is by using the Piggybacked symbols. To this end we proceed by downloading these piggybacked symbols in node 11, instance 6, 7 and 8. These are

$$(11,6)+\sum_{i=12}^{14}(i,4) \qquad (11,7)+\sum_{i=12}^{14}(i,5) \qquad (11,8)+\sum_{i=12}^{14}(i,3)$$

Now, since we already downloaded the systematic symbols of instances 6, 7 and 8, we can subtract (11,6), (11,7) and (11,8). Our goal is now to restore (12,3), (12,4), (12,5), (13,3), (13,4) and (13,5) from these summations. Because of similarity we only look at the repair of instance 3 which is done by looking at the the parity symbol instance 8, node 11. We are left with one symbol representing $(12,3)+(13,3)+(14,3)$. We can only download (14,3), since the other two are missing which leaves us with one equation with two unknowns. We also know that the missing information is stored nowhere else except for the entire 10 systematic symbols in this instance. Thus we conclude that we cannot restore these two lost symbols, other than downloading all systematic symbols. Therefore, our best option is to not use the piggybacked symbols, but to download all systematic symbols in the first place. In conclusion, the repair bandwidth for 2 failed parity nodes is 100%.

4.2 Two failures in the systematic nodes.

When reviewing this situation, we see note that the two failures can be in the same, or in different group(s) S_g. In both situations we look at $m=1$ and then see that this is also optimal for $m=2$.

1. Two failures happening in the same group. When two failures happen in the same group, we can still save download, giving a repair bandwidth of 86%. Here will look at the repair algorithm. A proof that this is in fact the most efficient repair method is given in Appendix B. We look at the situation where nodes 6 and 7 (both of group S_2) have failed. Same as in the repair of one failed systematic node, we start by downloading the parity symbols from instance 4 from node 12, instance 3 from node 13, and instance 5 from node 14 because they are the only symbols that contain $\mathbf{q}_{i,2}$. These are

$$\mathbf{p}_2^T\mathbf{a}_4+\mathbf{q}_{2,2}^T\mathbf{v}_2 \qquad \mathbf{q}_{3,2}^T\mathbf{a}_3-(\mathbf{p}_3^T\mathbf{a}_4+\mathbf{p}_3^T\mathbf{a}_5) \qquad \mathbf{p}_4^T\mathbf{a}_5+\mathbf{q}_{4,2}^T\mathbf{v}_4$$

This leads us to the unavoidable downloading of the 8 remaining symbols from both instances 4 and 5 in order to subtract these from the terms. Now we are left with only the parts with a $\mathbf{q}$-vector. Remember that $\mathbf{q}_{i,2}$ means that these terms consist of only symbols in group S_2. Now we download the three remaining systematic symbols from node 5 and we subtract these from the parts with the $\mathbf{q}$-vectors which leaves us with the following:

$$\begin{cases} p_{2,6}(a_{6,4}+a_{6,3}+2a_{6,2}+4a_{6,1})+p_{2,7}(a_{7,4}+a_{7,3}+2a_{7,2}+4a_{7,1}) \\ p_{3,6}(a_{6,3}+a_{6,4}+a_{6,5})+p_{3,7}(a_{7,3}+a_{7,4}+a_{7,5}) \\ p_{4,6}(a_{6,4}+a_{6,3}+4a_{6,2}+16a_{6,1})+p_{4,7}(a_{7,4}+a_{7,3}+4a_{7,2}+16a_{7,1}) \end{cases}$$

These are three equations with 10 unknowns. In order to be able to solve this, we will download at least 7 more parity symbols, which gives us 7 more equations. For the most efficient repair we download from node 11 instances 3, 4 and 5, we download from node 12 instances 1 and 3, from node 13 we download instance 1 and from node 14 we download instance 3 (see Table 3). These together with the systematic symbols of instances 1, 3, 4 and 5 suffice to repair, and they save downloading 7 systematic symbols, thus giving a the repair bandwidth of 86%.

2. Two failures happening in different groups. Now, we look at when the two systematic failed nodes are not in the same group S_g, let's say in our example nodes 1 and 7 failed.

Remember that we look at $m = 1$ and afterwards we can see we have an optimal repair method which does not require the download of the 3 piggybacked parity symbols of node 11 instances 6, 7 and 8. Thus, $m = 2$ is similar. Then these symbols lay in groups S_1 and S_2. We will need 10 equations, because we have 10 missing symbols. The first step is the same as in the previous example: we download all parity symbols with a $\mathbf{q}$-vector of groups 1 and 2, and we download the systematic symbols belonging to these groups, and we download all systematic symbols of instances 4 and 5 in order to subtract these. Now we have 10 unknown symbols in 6 equations. Now again observe that node 11, instance 4 and 5 are 'free' equations, which means that these are always optimal for download. We now have 8 equations, meaning that we only have to add at least two more equations. From the systematic nodes we only have the symbols left in group 3 (nodes 8, 9, 10) in instances 1, 2, 3. We can not add another equation without downloading new systematic symbols, so we need to add a parity symbols that requires download from instance 1, 2, or 3. We want to avoid downloading a parity symbol that requires the download of the systematic symbols in group 3. Thus, we download node 11 and 12 of instance 1, 2 or 3. One can check that it does not matter which instance we choose. We saved downloading 6 symbols. We have to download 44 symbols, which gives a repair bandwidth of 88%. Similarly, when two systematic nodes from groups 1 and 3 fail, we also have a repair bandwidth of 88%, and when the two nodes are of groups 2 and 3, the repair bandwidth is 84%.

4.3 One failure in a systematic node and one failure in a parity node

The last combination we look at is where one node is a systematic node and one is a parity node. There are two things we have already learned: when the parity node is node 11, the repair bandwidth is always 100%, and for the other parity nodes it holds that using the piggybacked symbols in node 11 instance 6, 7 and 8 is the only possibility of getting a repair bandwidth less than 100%. We look at the situation where nodes 1 and 12 fail.

Step 1. We start off by downloading the systematic symbols of instance 1, 2, 6, 7, 8, 9 and 10, together with the parity symbols of node 11, instance 6, 7 and 8, because we need to do this for the repair of node 12. We can now restore the failed symbols from instances 1, 2, 6 and 7 by downloading the parity symbols of these instances in node 13 and 14.

Step 2. In order to use the three symbols that are piggybacks for the parity symbols (node 11, instance 6, 7 and 8), we download the symbols of nodes 13 and 14 in nodes 3, 4 and 5. Now we have restored all symbols of the first system of node 12.

Step 3. Now there are 9 symbols left to be restored. Since the two systems cannot work together anymore, because the only symbols that were piggybacked across the two systems are already used, we treat the two systems apart from each other. The first system (where 3 symbols are still missing) can be optimally solved by using the two already downloaded symbols of nodes 13 and 14 in instance 4, together with the download of node 11 instance 5. In order to use these, we must download the systematic symbols of instance 3 nodes 2, 3 and 4. We have now completely restored the first system, and saved downloading of 6 symbols in instance 3. In the second system there is no systematic symbol left to be saved, which means we just have to download enough parity symbols in order to restore the 6 missing symbols.

In conclusion, when parity node 12, 13 or 14 fails together with a systematic node from group 1, the repair bandwidth is 94%. When the systematic node is of group 2 or 3 however, the repair is bandwidth is 93%, because we can save the download of 7 symbols in system 1.

4.4 Total repair bandwidth of two failures

We want to calculate the total repair bandwidth of two failures. In order to do so we use the found repair bandwidths for each possible combination:

$$\text{Repair bandwidth of two failed parity nodes} = \frac{\binom{4}{2}}{\binom{14}{2}} * 100\%$$

$$\text{Repair bandwidth of two failures in the same group} = \frac{\binom{4}{2} + \binom{3}{2} + \binom{3}{2}}{\binom{14}{2}} * 86\%$$

$$\text{Repair bandwidth of two failures different groups} = \frac{\binom{4}{1}\binom{3}{1} * 2 * 88\% + \binom{3}{1}\binom{3}{1} * 84\%}{\binom{14}{2}}$$

$$\text{Repair bandwidth of a combination} = \frac{\binom{10}{1}\binom{1}{1} * 100\% + \binom{4}{1}\binom{3}{1} * 94\% + \binom{3}{1}\binom{3}{1} * 2 * 93\%}{\binom{14}{2}}$$

When we calculate these and we add them together, we get

$$\text{Repair bandwidth for two failures} \approx 91.23\%$$

4.5 Three and four failures

We also want to know about the events of 3 and 4 failures, however due to time constraints during the writing of this thesis only a lower bound and an upper bound will be given here. We first note that in [4] it is explained that Piggybacking keeps the MDS property of a code intact which is the case for our (14,10)-RS code. From this we deduce that we are always able to restore the code when 4 failures happen. Since when 4 failures happen, we only have 100 symbols left, we find an upper bound of a repair bandwidth of 100%. Therefore, the upper bound for the repair bandwidth for 3 failures is also 100%. We also have a lower bound, because the repair bandwidth for 2 failures is 91.23%, which means that the repair bandwidth of 3 or 4 failures can never be lower. This gives us:

$$91.23\% \leq \text{repair bandwidth for three failures} \leq \text{repair bandwidth for four failures} \leq 100\%$$

5 Heptagon-Local Code

The Heptagon-Local Code differs from Piggybacking in a way that Piggybacking takes an original coding scheme (like Reed-Solomon) and adds an extra layer of encoding, where the Heptagon-Local Code is a coding scheme itself. In this chapter we will look at the Heptagon-Local Code as it is described in [6] and we will compare these results with 2-Replication, since the Heptagon-Local Code uses 2-Replication in its coding scheme. Before we can do so, and like in the previous chapter, we will start looking at the encoding scheme, followed by the decoding scheme and in conclusion we will look at the important parameters we need to compare this code to our previous codes.

5.1 Encoding

We start the coding scheme off with the 20 symbols we want to encode. We add a parity symbol which stores the sum of the 20 symbols. This symbol p will be called the *local parity* symbol. We initialize a complete graph of 7 nodes, thus 21 edges. Each of the 21 symbols is assigned to an edge, which stands for the two nodes the symbol will be stored in. This means that we end up with 7 nodes, which each stores 6 symbols, as shown in Figure 2.

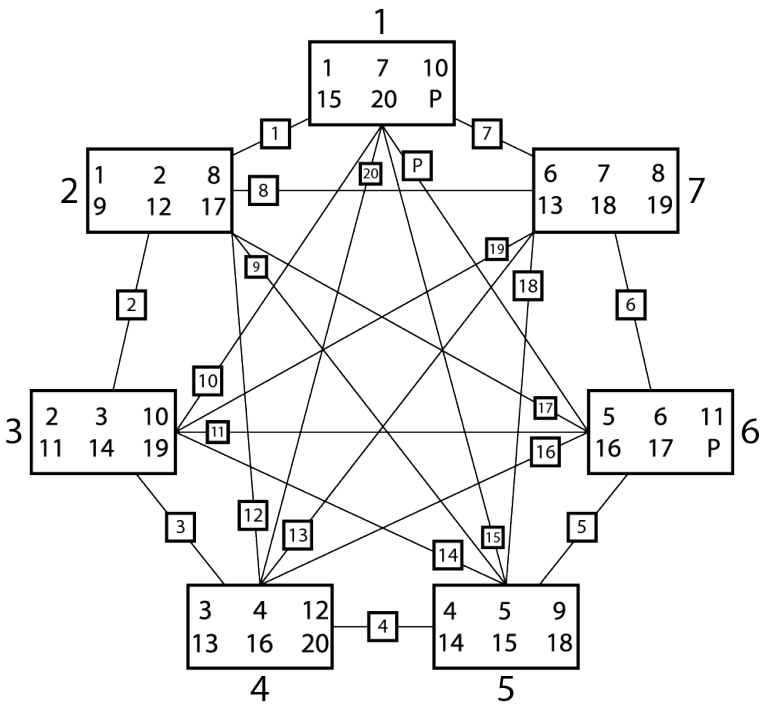


Figure 2: *The construction of a heptagon. The vertices stand for the 7 nodes in which the symbols will be stored, the edges show which two nodes that particular symbol will be stored in.*

This might look like 2-Replication, however, the way the symbols are stored makes this method more reliable than in 2-Replication. We will review this later on. To finish the encoding process, we take the original 40 symbols of 2 different heptagons and we calculate two parity symbols. These operations rely on Galois Field arithmetic. The first parity symbol is the result of adding the 40 symbols. The second parity symbol is the result of adding cyclic shifts of the 40 symbols in the following manner: $s_1 + \pi(s_2) + \pi^2(s_3) + ... + \pi^{39}(s_{40})$, where s_i is the ith original symbol. (The reader with little knowledge about cyclic shifts is referred to Appendix A.4.) The two parity symbols will be called the *global parity* symbols and they will be stored in the global parity node. We add all this together like shown in Figure 3.

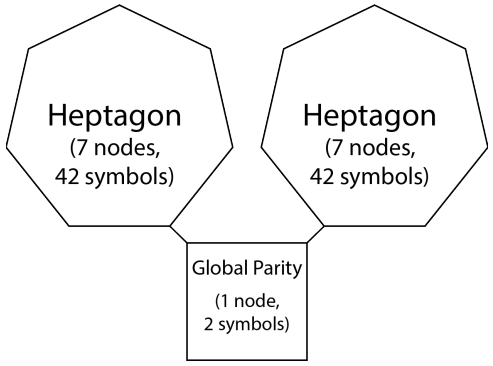


Figure 3: *The addition of two heptagons and one global parity node to complete the encoding scheme.*

This system of 86 symbols, stored over 15 nodes and encoding 40 original symbols, completes the encoding scheme of the Heptagon-Local Code.

5.2 Decoding

In order to be able to denote the nodes properly we say that the most upper node in Figure 2 is node 1, and we count up counter clockwise until the node on the right of node 1 is called node 7. We will now discuss up to three failures happening what possibilities and combinations we have and how we can solve these optimally.

5.2.1 One failure

First we look at one failure. This is rather simple since all six remaining nodes contain a copy of one of the six missing symbols which thus only have to be copied. For instance when we look at Figure 2 where node 1 failed, we see that the symbol 1 is stored in node 2, 10 is stored in node 3, 20 is stored in node 4, 15 is stored in node 5, p is stored in node 6 and 7 is stored in node 7. We copy these to node 1 and we have repaired the single failure with only 6 of the 40 original symbols (15%). Of course when the global parity fails, we can only repair this by downloading all 40 original symbols (100%). Thus the repair bandwidth of 1 failure thus is $15\% * \frac{14}{15} + 100\% * \frac{1}{15} \approx 20.67\%$.

5.2.2 Two failures

When two nodes fail there are three options: they are in the same heptagon, they are in different heptagons, or one of the two is the global parity node.
1. First, when both failures happen in the same heptagon, say nodes 1 and 2, we note that we now miss 11 symbols, where one edge in the graph represents a symbol that connects both our failed nodes. In our example this is the edge representing symbol 1. This means that this symbol is erased from the system. We however know the other 10 symbols are still copied anywhere else, so we start by downloading these. (Node 3 copies symbol 2 to node 2 and symbol 10 to node 1, and so on.) We also know that during the construction of the heptagon we took the 20 original symbols and made the 21^{st} symbol p which by adding all 20 symbols. To this end we use the 10 symbols we just copied together with the other 10 symbols we have not downloaded yet, and we can calculate the 21^{st} symbol (here symbol 1) and we are done. Note that we have to make sure not to download any duplicates. For instances, we want to download symbol 4 from either node 4 or 5, but not from both. If we do this properly, we can repair this by using only 20 of the 40 original symbols, giving a repair bandwidth of 50%. Note that we thus can repair two failures in the same heptagon by using only symbols that are stored within this single heptagon. We call this *repairing locally*.
2. When the two failures happen in different heptagons, we repair them both locally as in the previous subsection. This gives a repair bandwidth of $2 * 15\% = 30\%$.
3. Again, when one of the two is the global parity, we have to repair the failure in the heptagon first (like we did with 1 failure), then download all 40 original symbols and calculate the global parities again. This makes our repair bandwidth 100%.

To this end we see

$$\text{Repair bandwidth of 2 failures} = 2 * \frac{\binom{7}{2} * 50\% + \binom{7}{1}\binom{7}{1} * 30\% + \binom{14}{1}\binom{1}{1} * 100\%}{\binom{15}{2}} \approx 47.33\%$$

5.2.3 Three failures

Now, when three nodes fail, we again have a couple of options.
1. When three nodes within a single heptagon fail, we see that there are three edges erased. Lets say nodes 1, 2 and 3 fail. We can of course immediately copy the symbols that are still stored anywhere else, but we then still miss three symbols: 1, 2 and 10. We can use our parity symbol p to make 1 equation with three unknowns, but then we still need at least 2 more equations. For this, we use the global parities. As we know, the global parities are the sum and the product respectively of all 40 original symbols, which means we also have to download the 20 original symbols of the connected heptagon. Now we can make a system of three equations with 3 unknowns, which makes the decoding possible. This completes the repair scheme. We thus note that we can repair up to three failures in a single heptagon, as long as this does not happen in the other heptagon as well. The repair bandwidth is $40/40 = 100\%$
2. When the global parity node fails, we know that at most 2 failures happened in one of the heptagons. Since this can always be repaired locally, we do this first. Then we download all 40 original symbols for the repair of the global parity. We can quickly conclude that this is always repairable and the repair bandwidth is 100%
3. When 2 failures happen in one heptagon and 1 failure happens in the other, we can repair them both locally by using the schemes of the previous subsections. This gives a repair bandwidth of $50\% + 15\% = 65\%$.

Together this gives

$$\text{Repair bandwidth 3 failures} = \frac{\binom{7}{3} * 2 * 100\% + \binom{14}{2}\binom{1}{1} * 100\% + \binom{7}{2}\binom{7}{1} * 2 * 65\%}{\binom{15}{3}} \approx 77.38\%$$

5.3 Quality

We calculate that the storage overhead is $86/40 = 2.15$. The repair bandwidths for 1, 2 and 3 failures are already calculated above, deduced from schemes provided in [6]. However, we will now show that the Heptagon-Local Code also has a great chance of being able to repair when 4 or even 5 failures happen. We also see that the repair of 6 failures is impossible.

The repair of 4 failures is possible for some combinations, but not always. When for instance 4 failures happen in 1 heptagon, we see that there are now 6 symbols entirely lost. Since we can make only 3 equations from the 3 parity symbols we have, we cannot restore the 6 missing symbols. We however can repair:
1. When the global parity node fails, we can only at most 2 failures in the same heptagon, leaving one failure in the other. Namely, when three failures happen in a single heptagon, we cannot repair it other than using the global parity node. Since we need to repair the global parity node in these cases, we always have a repair bandwidth of 100%.
2. When the global parity node does not fail we can have 3 failures in one heptagon and 1 in the other, or we can have 2 failures in both heptagons. In the first case we need the global parity node, giving a repair bandwidth of 100%, in the second case we repair both heptagons locally with both a repair bandwidth of 50%, thus 100% in total.
In all cases we have a repair bandwidth of 100%. In conclusion, when 4 failures happen there is a

$$\frac{2 * \binom{1}{1}\binom{7}{2}\binom{7}{1} + 2 * \binom{7}{3}\binom{7}{1} + \binom{7}{2}\binom{7}{2}}{\binom{15}{4}} \approx 89.74\%$$

chance that we can repair with a repair bandwidth of 100%.

When 5 failures happen we again have 2 cases. When the global parity node fails, we can only have 2 failures in both heptagons, and when the global parity node does not fail, we can have 3 failures in the one heptagon and 2 in the other (and vice versa). This is in

$$\frac{\binom{1}{1}\binom{7}{2}\binom{7}{2} + 2 * \binom{7}{3}\binom{7}{2}}{\binom{15}{5}} \approx 63.64\%$$

of the cases. Thus when 5 failures happen we can repair in 63.64% of the cases with a repair bandwidth of 100%.

When 6 failures occur we see that we can never repair this:
1. When the global parity node fails we know that we know that we can only repair up to 2 failures per heptagon. However, when 6 failures occur where 5 happen in the heptagons, there must be at least 3 failures in one of the heptagons, meaning we cannot repair this.
2. When the global parity does not fail we can repair up to three failures in a single heptagon, however we already saw that when three failures happen in both heptagons at the same time we have data loss. When 6 failures occur in the heptagons, either both heptagons have 3 failures, or one of them has 4 or more failures, which in both mean data loss.

5.4 Difference with 2-Replication

Like said before, the Heptagon-Local Code depends on replication of all original symbols. Even 6 extra parity symbols have to be added to the system of 80 symbols. Of course the Heptagon-Local code has an advantage in comparison with just 2-Replication. Therefore we will now compare the two. To make a fair comparison we want to compare the Heptagon-Local Code with a 2-Replication system which stores the same amount of original data. Since 2-Replication always has an even amount of servers we take a 2-Replication system with 14 and one with 16 nodes. We will show that the systems with approximately the same amount of nodes are both less reliable than the Heptagon-Local Code. Like we have seen, the Heptagon-Local Code can always repair up to 3 failures, and like we have also already seen in Example 2.1 with 2-Replication data can be lost with 2 failed nodes. When 1 failure happened in the 2-Replication system of 16 nodes, there is a $1/15 = 6.7\%$ chance that the next failure is fatal to the data. In the same way when 2 not-fatal failures happened, there is a $2/14 = 14.3\%$ chance that the next failure is fatal and so on. In general, when in a 2-Replication system of n nodes there have happened i non-fatal failures, the chance of the next failure being fatal is $\frac{i}{n-i}$ (where we have already lost data when $\frac{i}{n-i} > \frac{1}{2}$). We see this in the next table:

	1^{st} fail	**2^{nd} fail**	**3^{rd} fail**	**4^{th} fail**	**5^{th} fail**	**6^{th} fail**	**7^{th} fail**
2-Rep (n=14)	0%	7.7%	16.7%	27.3%	40%	55.6%	75%
2-Rep (n=16)	0%	6.7%	14.3%	23.1%	33.3%	45.5%	60%
HLC	0%	0%	0%	10.26%	36.36%	100%	-

Table 8: *When an amount of non-fatal failures happened, the chance of the next failure leading to data loss is given for three codes: two 2-Replication systems of 14 and 16 nodes and the Heptagon-Local Code of 15 nodes.*

We can observe that the great advantage of the Heptagon-Local Code is the reliability. Up to 3 erasures can always be repaired, and up to 4 erasures, the chance of that erasure being fatal is smaller than that of the 2-Replication. With 4 failures the chance of the 5th failure being fatal is for the *average* of the two 2-Replication systems approximately the same as for Heptagon-Local Code. Since the chance of 6 out of 15 nodes failing is negligible, these percentages are practically insignificant. Even if the chances of 4 or 5 failures were better, the fact that 2-Replication can't always resolve 2 and 3 failures is already a great disadvantage, since any data loss is something that we absolutely do not want as a company. Even if the chance of that failure being fatal is small.

6 Comparison

In this chapter we will compare the (14,10)-RS, the Piggybacked (14,10)-RS and the Heptagon-Local Code. In order to do this, we will use the information that the literature provided us with, and the information that we derived from it, which is all summed up in the previous chapters.

6.1 Repair bandwidth and storage overhead comparison

First we ask ourselves if we can compare the two codes, since we said in Section 2.1 that both systems need to store the same amount of original data for that. The Piggybacked code stores 100 original symbols and the Heptagon-Local code stores 40 symbols. For that, we do the same as we did when we wanted to piggyback the parity nodes of the (14,10)-RS code in Section 3.2.2: we store multiple systems in the same nodes. We will store 5 systems of the Heptagon-Local Code in the same 15 nodes and we will store 2 systems of the Piggybacked (14,10)-RS code in the same 14 nodes. This way we can make a fair comparison, which does not change the repair bandwidth nor the storage overhead. We will see a more detailed explanation of this way of adjusting a code in Section 6.3.1. The next table summarizes the found repair bandwidths of the discussed codes:

	1 failure	**2 failures**	**3 failures**	**4 failures**	**5 failures**
(14,10)-RS	100%	100%	100%	100%	data loss
PB'd (14,10)-RS	67.21%	91.23%	91.23%$\leq x \leq$100%	91.23%$\leq x \leq$100%	data loss
HLC	20.67%	47.33%	77.38%	100% [1]	100% [2]

Table 9: *The repair bandwidths for the three evaluated codes: the (14,10)-RS Code, the Piggybacked (14,10)-RS Code and the Heptagon-Local Code.*

We see that the Heptagon-Local Code scores more than 3 times better than the Piggybacked RS Code when it comes to repairing 1 failure. Also, when 1 node fails in the Heptagon-Local Code, no calculations need to be made since the symbols only need to be copied to the failed node, where the Piggybacked RS code needs some calculation after the download of three times more data. Also, when 2 failures happen the repair bandwidth of the Heptagon-Local Code is almost 2 times better than that of the Piggybacked RS Code. Still, when it comes to storage overhead, the Heptagon-Local Code costs way more.

	Storage overhead
(14,10)-RS	1.4
PB'd (14,10)-RS	1.4
HLC	2.1

Table 10: *The storage overhead for the three evaluated codes: the (14,10)-RS Code, the Piggybacked (14,10)-RS Code and the Heptagon-Local Code.*

Since we do not know how much these two factors (storage overhead and repair bandwidth) weigh in terms of money in comparison to each other, we will look at if what happens if we change some parameters in both directions.

[1] Like described in Section 5.3 repair is possible in 89.74% of the cases.
[2] Like described in Section 5.3 repair is possible in 63.64% of the cases.

6.2 Changing the Heptagon-Local Code

It is difficult to change the length of the Heptagon-Local Code. The only thing that comes to mind when talking about changing something about it is combining two of these systems via the adding two global parity nodes on both sides. This way we have 4 heptagons and 4 global parity nodes connected alternately until the fourth global parity node is connected to the first heptagon. This adds an extra layer of security for when 4 or 5 failures happen, namely when for instance 3 failures in a heptagon happen, and a connected global parity fails, the heptagon still has another global parity with connected heptagon to try and restore the failed nodes. However, in comparison to the Piggybacked (14,10)-RS this adds even more storage overhead, and the only thing it wins is more security, which the heptagon-Local Code already had. This makes this idea condemnable.

6.3 Changing the Piggybacked (14,10)-RS code

As we know, the Piggybacked (14,10)-RS code performs great in terms of storage overhead, however, we also saw that its repair bandwidth for when 1 failure happens is more than 3 times higher then that of the Heptagon-Local Code. We will try and see what happens when we increase the storage overhead to 2.1, which is nearly as high as that of the Heptagon-Local Code (2.15). We first observe that this makes it a Piggybacked (21,10)-RS code, and we also observe that this is the limit of the length when it comes to Piggybacking, since there are exactly 10 groups S_g, which is equal to the amount of systematic nodes, making $\mathbf{q}$-vector consist of only 1 symbol. When we use the encoding algorithm, we see that we now have 19 instances in 10 systematic nodes, meaning our message is 190 symbols long. We have 10 groups S_g all existing of 1 systematic node, and we are able to define the $\mathbf{v}$-vectors. Next are the $\mathbf{q}$-vectors, which all exist of just 1 non-zero entry of the corresponding $\mathbf{p}$-vector. We can now Piggyback the 11 parity nodes accordingly to the algorithm.

6.3.1 Making a fair comparison

Remember from Section 2.1 that we said that a fair comparison means that both systems store the same amount of original symbols. Now a system of the (14,10)-code stores 50 original symbols, and a system of the (21,10)-code stores 190 original symbols. To this end we choose to store 950 symbols with both codes (which is the smallest common multiple of 50 and 190). This means that the (14,10)-code needs 19 systems of 10 systematic and 4 parity nodes, and the (21,10)-code needs 5 systems of 10 systematic and 11 parity nodes. Since our nodes are big enough in practice to store multiple systems, we can store all 19 systems of 14 nodes in the same 14 nodes. Even so, we can store the 5 systems of 21 nodes in the same 21 nodes. This looks as follows:

	Ins 1		Ins 5	Ins 6		Ins 10		Ins 91		Ins 95
Node 1	$a_{1,1}$	$\dots$	$a_{1,5}$	$a_{1,6}$	$\dots$	$a_{1,10}$	$\dots$	$a_{1,91}$	$\dots$	$a_{1,95}$
	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$		$\vdots$	$\ddots$	$\vdots$
Node 10	$a_{10,1}$	$\dots$	$a_{10,5}$	$a_{10,6}$	$\dots$	$a_{10,10}$	$\dots$	$a_{10,91}$	$\dots$	$a_{10,95}$
Node 11	$\mathbf{p}_1^T\mathbf{a}_1$	$\dots$	$\mathbf{p}_1^T\mathbf{a}_5$	$\mathbf{p}_1^T\mathbf{a}_6$	$\dots$	$\mathbf{p}_1^T\mathbf{a}_{10}$	$\dots$	$\mathbf{p}_1^T\mathbf{a}_{91}$	$\dots$	$\mathbf{p}_1^T\mathbf{a}_{95}$
	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$		$\vdots$	$\ddots$	$\vdots$
Node 14	$\mathbf{p}_4^T\mathbf{a}_1$	$\dots$	$\mathbf{p}_4^T\mathbf{a}_5 + \mathbf{q}_{4,2}^T\mathbf{v}_4$	$\mathbf{p}_4^T\mathbf{a}_6$	$\dots$	$\mathbf{p}_4^T\mathbf{a}_{10} + \mathbf{q}_{4,5}^T\mathbf{v}_4$	$\dots$	$\mathbf{p}_4^T\mathbf{a}_{91}$	$\dots$	$\mathbf{p}_4^T\mathbf{a}_{95}\mathbf{q}_{4,56}^T\mathbf{v}_4$

Table 11: *Storing 950 symbols according to the Piggybacked (14,10)-RS scheme.*

	Ins 1		Ins 19		Ins 76		Ins 95
Node 1	$a_{1,1}$	...	$a_{1,19}$	...	$a_{1,76}$	...	$a_{1,95}$
	$\vdots$	$\ddots$	$\vdots$		$\vdots$	$\ddots$	$\vdots$
Node 10	$a_{10,1}$	...	$a_{10,19}$	...	$a_{10,76}$	...	$a_{10,95}$
Node 11	$\mathbf{p}_1^T\mathbf{a}_1$	...	$\mathbf{p}_1^T\mathbf{a}_{19}$	...	$\mathbf{p}_1^T\mathbf{a}_{76}$	...	$\mathbf{p}_1^T\mathbf{a}_{95}$
	$\vdots$	$\ddots$	$\vdots$		$\vdots$	$\ddots$	$\vdots$
Node 21	$\mathbf{p}_{11}^T\mathbf{a}_1$	...	$\mathbf{p}_{11}^T\mathbf{a}_{19} + \mathbf{q}_{11,9}^T\mathbf{v}_{11}$	...	$\mathbf{p}_{11}^T\mathbf{a}_{76}$	...	$\mathbf{p}_{11}^T\mathbf{a}_{95} + \mathbf{q}_{11,49}^T\mathbf{v}_{11}$

Table 12: *Storing 950 symbols according to the Piggybacked (21,10)-RS scheme*

Now both codes store the same amount of original data. This means we are ready for a fair comparison.

6.3.2 Comparing the Piggybacked (21,10)-RS with the Piggybacked (14,10)-RS

We will look at the repair bandwidths for 1, 2 and 3 failures in the Piggybacked (21,10)-RS code. We start with 1. When it comes to restoring 1 failed systematic node, we can follow the decoding algorithm from Section 3.2.1.

Example 6.1. *Say node 1 failed.*
1. *We download all data from the last $r-2=9$ instances in nodes $\{1,...,11\}\backslash\{1\}$. We use these symbols to repair the symbols of these instances. Now there are 10 symbols left to recover.*
2. *Since node 1 is in group S_1, we download the 10 parity symbols that consist of a part that is of the form $q_{i,1}$. We subtract the parts that are of the form $\boldsymbol{p}_r^T\boldsymbol{a}_i$ with $i \in \{10,...,19\}$ (which we already downloaded).*
3. *There are no other nodes in group S_1 so we do not have to download symbols from this group. Therefore we have already recovered the lost data in the previous step. We thus needed to download 10+9 parity symbols and 9*9 systematic symbols, making a total of 100 symbols.*

This is a repair bandwidth of $100/190 = 52.63\%$ which means that some progress is made for the repair of a systematic node. However, for the parity nodes we get a repair bandwidth of 100%.

Example 6.2. *When we look at the parity symbol of node 11 instance 20, we see that this symbol consists of the encoded message vector $\boldsymbol{a}_{20}$, added by the sum of all parity symbols of instance 11, nodes 12 to 21. When for example node 12 has failed, we can use that piggybacked symbol of instance 20 for the repair of the lost symbol in instance 11. We do this by downloading it, together with the 10 systematic symbols of instance 20 and the parity symbols of instance 11, nodes 13 to 21. However, we now see that instead of downloading all 10 systematic symbols in instance 11, we have to download 9 parity symbols in instance 11 together with the piggybacked parity symbol in instance 20. This is also 10 symbols. When we do this for every instance we see that we still need to download 190 symbols for repair, which makes the repair bandwidth 100%.*

In general, the smaller the groups S_g are, the smaller the effect of Piggybacking parity nodes has on the repair bandwidth. This is also said in [4]. Since there are 11 parity nodes and only 10 systematic nodes, we know that the repair bandwidth of 1 failure in total is $52.63\% * \frac{10}{21} + 100\% * \frac{11}{21} = 77.44\%$. Remember that the repair bandwidth in the (14,10)-code was only 67.21%, which means that by increasing the storage overhead, we also increased the repair bandwidth.

We can almost already condemn this (21,10)-code, however, for the sake of completeness we briefly discuss what happens for 2 and 3 failures. When one wants to check these repair bandwidths, one only has to check for 1 option, namely multiple failures happening in different groups of systematic nodes. This is because when a parity node fails, the repair bandwidth is now 100%, and there cannot be two failures in the same group since every group exists of only 1 node. The repair bandwidth of 2 failed systematic nodes is 74.21%, and when 2 failures happen where at least one of them is a parity node, we have a repair bandwidth of 100%. This means

$$\text{Repair bandwidth for two failures} = \frac{\binom{10}{2}}{\binom{21}{2}} * 74.21\% + \frac{\binom{10}{1}\binom{11}{1} + \binom{11}{2}}{\binom{21}{2}} * 100\% = 94.74\%$$

This in comparison to 91.23% with the (14,10)-code. In the same way we calculate the repair bandwidth for 3 failed systematic nodes, which is 84.74%, giving a total repair bandwidth for 3 failures of 98.62%. Even if the repair bandwidth for 3 failures in the (14,10) code is 100% (which is only an upper bound) this will not make up for the costs that are made for increasing the storage overhead as well as for the repair bandwidths of 1 and 2 failures.

Theoretically there is one benefit to the (21,10) code, which is the fact that it can repair up to 11 failed nodes. However, we already know that the chance that we need to repair more than 4 out of 14 nodes at the same time is extremely small. Also, like we briefly saw at the end of Section 2.1, when we increase the amount of nodes (from 14 to 21), the chance of node failures within the system also increases. The repair of a server takes a certain amount of time, and every server has a certain chance of failing within that time, say α. The chance that no server fails in this time is $(1-\alpha)^{14}$ versus $(1-\alpha)^{21}$, so as we expect, this chance is greater for 14 nodes than for 21 nodes. So, not only do we increase the repair bandwidth, we also increase the chance of a node failing. We can state that also the idea of increasing the storage overhead of the Piggybacked (14,10)-RS code is condemnable.

7 Conclusion and future work

In this chapter we will discuss the conclusions we can deduce from this thesis, then we will discuss a list of recommendation for future work together with some remarks on what could have improved the results of this thesis.

7.1 Conclusion

The two tables presented in Section 6.1 provide the best found results in this thesis. We know that Piggybacking the (14,10)-RS code increases the quality of this the RS code itself, however, in comparison to the Heptagon-Local Code one could argue which code works better. One could say that the two codes serve different purposes. For instance when we look at the difference between hot data (data that is being accessed a lot) and cold data (data that is not being accessed a lot and is sometimes not even being accessed for months). A system like the Heptagon-Local Code would work better for the storage of hot data since it can repair 1 failure real quick, where the Piggybacked (14,10)-RS code would work better for cold data, since it matters less that lost data takes longer to be repaired, while at the same time it keeps the storage overhead costs low. However, the answer to the question we asked ourselves in Section 1.2 is yes: it seems to be profitable to switch from the RS code to either Piggybacking or the Heptagon-Local Code.

7.2 Discussion and future work

We have discussed two recently found codes in terms of reliability, where we said that there are certain chances of nodes failing and certain chances of that many failures being fatal to the data. However, this comparison exists of a lot of numbers which can all be summarized in a beautiful parameter called the Mean Time To Data Loss (MTTDL), which is a rather self-explaining parameter. It is calculated for 2-Replication in [7] by using Markov chains, and the authors of [6] used this method to calculate the MTTDL for the Heptagon-Local Code. If we would do the same for the Piggybacked (14,10)-RS code, we would be able to specifically compare the two codes in reliability. During the writing of this thesis, it was in fact planned to include this parameter in the comparison. However, a week before the deadline we found a mistake, which left no time for the inclusion of new work. Another interesting matter is that we saw that increasing the storage overhead in the Piggybacked (14,10)-RS code also increased the repair bandwidth. This arises the question of what would happen if we decrease the storage overhead and for instance look at an (13,10)-code. Would this also decrease the repair bandwidth, and how would this affect its MTTDL? Then, it would be easier to compare codes if we would know how often a node fails, how long it takes to repair a node and how expensive storage overhead is. Different practical researches have been done on this subject but results differ quite a lot from each other. At last, in [2] 8 recently found codes are named. We only discussed two, so there are 6 more codes left to cover.

A From basic erasure coding to Reed Solomon Codes

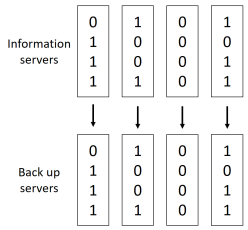
This appendix is made as an alternative for Chapter 2. One could read it in full and should then be able to proceed reading at Chapter 3.

In this thesis we will discuss two recently found codes and we will compare them with Reed Solomon, but before we can do so, we must first understand what erasure coding is and work our way to the definition of a Reed Solomon Code. We start off with an introduction to code design. We continue with three ways of modifying a code. Then we proceed by looking at two classes of codes: cyclic codes and BCH codes. This because Reed Solomon codes are a special case of BCH codes which is a special case of cyclic codes. Most results found in this chapter are derived from [3].

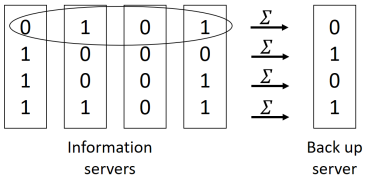
A.1 An introduction to code design

In this section we will discuss the way erasure correcting codes are constructed and how they work. We will first look at this intuitively and than generalize it to the basic mathematical fundamentals for erasure correcting codes. The data we discussed in Chapter 1 is stored in the form of bits; zeros and ones. The servers in which the data like photos etc. is stored are called *information servers*. To these servers, *parity servers* are added and connected in a way that certain groups of nodes should be able to repair a failed node when necessary. Each method has its advantages and its disadvantages. We will now look at an example of this, based on the following two codes.

Example A.1.



(a) A direct copy of every information server.



(b) One parity server based on the sum of each row of bits of the information servers.

Figure 4: Two examples of relatively simple codes.

We can see that there is a trade-off in these two codes when looking at their structure. The first code stores 2 times as much data as the original message size, where the second code only stores 25% extra data. We call this concept *storage overhead*, which is defined as the total amount of data divided by the size of the original data (in this case respectively 2 and 1.25). However, when in the first code a server fails we only have to read&download 25% of the message size (namely the 4 symbols in the copied server) to restore the lost information, where in the second code we need to read&download the data from all four remaining servers, which stores as much as the whole message. This is called the *repair bandwidth* (in this case respectively 25% and 100%). What is also important to notice is that when two nodes fail in the first code, there is still a chance that the message can be recovered, for instance when node 1 and 2 fail, but when two nodes in the second code fail, the message is forever lost. However, there is also a counter argument to this concept. The repair of a server takes a certain amount of time, say 1 day, and

every server has a certain chance of failing within the next 24 hours, say α. Then with a system of 8 servers where 1 is down, the chance that at least one other server failing within 24 hours is $1-(1-\alpha)^7$, where with a system of 5 servers that chance is only $1-(1-\alpha)^4$. These are all factors that we have to consider when comparing codes.

We will now look at the concept of backing up information servers more mathematically based on Example 2.1. We take the first bit of all the information servers, and put them into a *message word*, defined as $\mathbf{m} = m_1m_2m_3...m_k$ with k the length of the message word(= the amount of information servers). We map this message into a *code word* $\mathbf{c} = c_1c_2c_3...c_n$ with $n > k$ the length of the code word(= the total amount of servers), via multiplication with an $k \times n$ matrix called the *generator matrix*. Note that in general the first k bits of $\mathbf{c}$ don't necessarily need to be the same as the bits of $\mathbf{m}$. We will now extend Example 2.1 by looking at their generator matrices.

Example A.2. *On the left we see generator matrix G_a that maps any message word of length 4 into a code word of length 8 consisting of exactly two times the message word. On the right is generator matrix G_b that maps a message word of 4 bits into a code word of 5 bits consisting of the message word followed by a fifth bit that represents the sum of the four previous bits.*

$$G_a = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad G_b = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

We note that both matrices are of the form $[I_k|P]$, with I_k the $k \times k$ identity matrix, which results in the message word being contained in the code word it has been mapped to. These codes are called *systematic*. Any generator matrix can by Gaussian elimination be transformed into a systematical matrix, which results into all the parameters of the code staying the same. Only which message word maps to which code word changes. (Note that not all parameters of a code have yet been discussed.) To this end for simplicity reasons we will sometimes assume a code to be systematic. This can be done without loss of generality.
In order to determine a codes quality we will look at some definitions and put these together into an important parameter called *distance*. We note that these codes are made of vectors over $\mathbb{F}_q^n$, a finite field with q elements and n the length of vectors in C. In Example A.2 these were $\mathbb{F}_2^8$ and $\mathbb{F}_2^5$ resp.

Definition A.3. *A code C is said to be linear if its vectors form a linear subspace of vector space V of rank k over $\mathbb{F}_q^n$.*

Remark A.4. *In the case of binary codes, one only has to check whether addition of two code words gives another code word to see if a code is linear.*

Definition A.5. *The weight of a binary code word $\boldsymbol{c}$ is defined as $\sum_{i\in[1,n]} \boldsymbol{c}_i$.*
Notation: wt($\boldsymbol{c}$)

Definition A.6. *Let C be a binary code. The distance d between two code words $\boldsymbol{x}$ and $\boldsymbol{y}$ in C is defined as wt($\boldsymbol{x}$-$\boldsymbol{y}$). The distance d of C is defined as* $\min\{wt(\boldsymbol{x}-\boldsymbol{y})|\boldsymbol{x},\boldsymbol{y} \in C \wedge \boldsymbol{x} \neq \boldsymbol{y}\}$.
Notation: d($\boldsymbol{x}$,$\boldsymbol{y}$) and d(C) resp.

As we will see shortly after, the distance of a code is a significant parameter. However, to make a list of all possible code words and compare each pair in order to get to know the distance is a lot of work, especially with codes that are being used in practice. To this end we will now show a very helpful lemma for finding the distance of a linear code.

Lemma A.7. *Let C be a linear code. Then d(C)* $= \min\{wt(\boldsymbol{x}) | \boldsymbol{x} \in C \wedge \boldsymbol{x} \neq 0\}$

Since in this thesis we will only be looking at linear codes, Lemma A.7 will be saving us a lot of work.
To know the distance of a code is of great importance in the field of erasure correcting code, because this shows us directly how many erased nodes the code is able to correct by the following relation.

Lemma A.8. *Let C be a code with distance d. Then the code can always correct at least d-1 erasures.*

Proof. Let $\mathbf{c} \in C$ be arbitrary. Since the distance is d we know that, in relation to any arbitrary $\mathbf{c}^* \in C$, $\mathbf{c}$ is unique in at least d bits. Now say that an erasure appears in $d-1$ random bits. Since $\mathbf{c}$ was different in at least d bits, there now is at least 1 bit left that distinguishes $\mathbf{c}$ from $\mathbf{c}^*$. Since $\mathbf{c}$ and $\mathbf{c}^*$ were randomly chosen, this holds for any pair of code words in C. □

We will now look at a new code called the *Hamming code* and use this example to summarize what we have seen so far, and to introduce three ways of modifying the standard form of a code. A Hamming code is a $(n = 2^m - 1, k = 2^m - 1 - m, d = 3)$-code for any $m \in \mathbb{N}$ (e.g., a (7,4,3)-code, a (15,11,3)-code, etc.). However, most of the times when a certain code is used in practice, for multiple possible reasons these standard parameters do not meet the requirements. Say a company would want to use a code with 13 servers (n=13), then the Hamming code would not be usable. In this case, one could use one of these three ways to modify a code: extending, shortening and puncturing. First we will summarize this chapter based on the (7,4,3)-Hamming code and then we will discuss these methods.

Example A.9. *The generator matrix of the (7,4,3)-Hamming code is defined as follows:*

$$G_{Hamm} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

One can check that this code is linear. We notice that G_{Hamm} *is of the form* $[I_4|P]$ *which means this code is systematically. Let us now use Lemma A.7 in order to determine the distance of the code. We multiply the 16 possible 4-bit vectors by the* G_{Hamm} *and find a list of all possible code words:*

0000.000	1000.111	0100.110	1100.001
0001.011	1001.110	0101.101	1101.010
0010.101	1010.010	0110.011	1110.100
0011.110	1011.001	0111.000	1111.111

We observe that the minimum weight of the code words (that is not the zero word) is 3, therefore by Lemma A.7 we confirm $d = 3$*. Lemma A.8 directly gives us that this code can repair any combination of* $3 - 1 = 2$ *erasures.*

Corollary A.9.1. *We can look at G_{Hamm} in a different way:*

$$\begin{aligned} c_1 &= m_1 \\ c_2 &= m_2 \\ c_3 &= m_3 \\ c_4 &= m_4 \\ c_5 &= c_1 + c_2 + c_3 \\ c_6 &= c_1 + c_2 + c_4 \\ c_7 &= c_1 + c_3 + c_4 \end{aligned}$$

Corollary A.9.1 visualizes how an erasure can be repaired. For instance when c_2 fails, we can easily see that we either need to access nodes c_5, c_1, c_3 or nodes c_6, c_1, c_4. This idea is also used when *hot data* is stored on a server, which means that a lot of computational power is asked from the server. When hot data is stored in c_2, a client asking for this data can simply be redirected by combining the data stored in c_5, c_1, c_3.

We will now move on to the three ways to modify a code by using Example A.9.

A.2 Three ways to modify a code

We start off with *extending*. This is a technique that extends the length of the code by 1, in order to increase the distance of the code by 1. By Lemma A.8 this provides the code with the ability to repair an extra erasure. Extending works by adding a 0 to each code word with an even weight, and a 1 to the words that have an odd weight. Thus this extra bit is determined by taking the sum over the bits of the code word. In a generator matrix this means that a column consisting of all ones is added to the end. For the Hamming code this means the generator matrix would now look like:

$$G_{HammExt} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

The result is a (8,4,4)-Hamming code. Note that k did not change. Extending a code seems like an endless solution, however:

Theorem A.10. *Let C be a code. Extending C increases d(C) by 1 $\iff$ d(C) is odd.*

Proof. First let d(C) be even. We take a pair **x,y** $\in C$ with d(C) = wt(**x**-**y**) arbitrary. Now either wt(**x**) and wt(**y**) must be even, or wt(**x**) and wt(**y**) are odd. In the first case, by the definition of extending, a zero is added, while in the second case a one is added to both **x** and **y**. This means that either way wt(**x**-**y**) has not changed and therefore d(C) has not changed. Now let d(C) be odd. We take a pair **x,y** $\in C$ with d(C) = wt(**x**-**y**) arbitrary. Now it must be either that wt(**x**) is even and wt(**y**) is odd or the other way around. In this first case **x** is extended with a 0 and **y** is extended with a 1 which means wt(**x**-**y**) has increased by 1. The other way around works similarly. Since this hold for any pair in C, we conclude that d(C) has increased by 1. □

We will now look at two methods that decrease the value of n. The first one, *shortening*, affects the value of k whilst keeping d constant most of the times. The second method, *puncturing*, affects the value of d whilst keeping k constant.

Intuitively shortening consists of the following steps:
1. We choose a position between 1 and n
2. We keep only the code words that have a 0 on this position
3. In these remaining words we delete the zero's on this position (since they are all zero this doesn't affect the code)
This means that n has decreased by 1, and for linear codes we can say we now end up with half of the amount of code words we started with. Theoretically it could have happened that in all the pairs of code words $\mathbf{x},\mathbf{y} \in C$ for which $d(C) = wt(\mathbf{x} - \mathbf{y})$ holds, at least one of them was deleted during the shortening. In this case $d(C)$ would have increased by at least one. However, for two reasons in this thesis we will assume that this is not the case. The first reason is that this almost never happens, and the second reason is that we are interested in the 'worst case scenario'. More generally, shortening corresponds with deleting a server node, ergo, deleting a column and its corresponding row from the generator matrix. Thus, after shortening a (n, k, d)-code one time we end up with an $(n-1, k-1, d)$-code. Shortening can be repeated as many times as desired.

The last method, puncturing, means deleting a column of the generator matrix. Whenever this happens with an (n, k, d)-code, we end up with a $(n-1, k, d-1)$-code. Puncturing is a useful tool for example when a company has decided that the amount of information servers is fixed. Since we will not be using puncturing in this thesis, we will not further discuss this.

A.3 Code words as polynomials

Reed Solomon is a class of codes that is being used a lot in practice. This, because Reed Solomon codes have a lot of benefits in comparison to other well known codes. For instance, a user of Reed Solomon codes can choose the amount of information servers and parity servers very easily, and Reed Solomon codes are reliable meaning that they can be repaired still after relatively a lot of erasures. In the remaining part of this chapter we will discuss what Reed Solomon codes are exactly. To do so, (since Reed Solomon is a special case of BCH codes, which is a special case of cyclic codes) we will look at cyclic codes and BHC codes before we move on to Reed Solomon codes. Still, since our goal is to define Reed Solomon codes, we will not discuss all properties concerning cyclic codes and BCH codes. We start off with this section about introducing a new way of notation of code words which we find adequate when discussing any kind of cyclic code in general.

A word $\mathbf{v}$ of length n we can notate as the polynomial $v(x) = a_0 + a_1x + a_2x^2 + ... + a_{n-1}x^{n-1}$ of degree $n-1$, where $a_0, ..., a_{n-1}$ are elements of $\mathbb{F}_q$. We find this adequate because we can calculate more easily with polynomials in comparison to just a list of bits. To this end, let's define the way we calculate with these polynomials. Adding and subtracting works exactly the way we expect it to work. When for instance $f(x)$ and $g(x)$ are polynomials over $\mathbb{F}_2$ (which means that $1 + 1 = 0$) we know that $f(x) - g(x) = f(x) + g(x) = g(x) - f(x)$. We can also observe that if $f(x)$ and $g(x)$ both are of degree k, then since $x^k + x^k = 0$ we see that $(f+g)(x)$ is of degree $< k$. We will look at multiplication in the following example.

Example A.11. *Let* $f(x) = 1 + x^2 + x^4$ *and* $g(x) = 1 + x + x^3$. *Then*
$f(x)g(x) = (1 + x^2 + x^4)(1 + x + x^3) = 1(1 + x + x^3) + x^2(1 + x + x^3) + x^4(1 + x + x^3)$
$= 1 + x + x^3 + x^2 + x^3 + x^5 + x^4 + x^5 + x^7 = 1 + x + x^2 + x^4 + x^7$.

To close it off, we discuss dividing two polynomials by long division.

Example A.12. *Let* $f(x) = 1 + x^3 + x^4 + x^6 + x^7 + x^8$ *and let* $g(x) = 1 + x^2 + x^3$. *By long division we find*

$$
\begin{array}{l|l|l}
1 + x^2 + x^3 & 1 + x^3 + x^4 + x^6 + x^7 + x^8 & x^5 + x^3 + x + 1 \\
\hline
 & \underline{x^5 + x^7 + x^8} - & \\
 & 1 + x^3 + x^4 + x^5 + x^6 & \\
 & \underline{x^3 + x^5 + x^6} - & \\
 & 1 + x^4 & \\
 & \underline{x + x^3 + x^4} - & \\
 & 1 + x + x^3 & \\
 & \underline{1 + x^2 + x^3} - & \\
 & x + x^2 &
\end{array}
$$

Thus the remainder of this division is $x + x^2$. We can look at this the following two ways:
1. $f(x) = (1 + x + x^3 + x^5)g(x) + (x + x^2)$
2. $f(x) = (x + x^2) \bmod g(x)$

Especially the second way of looking at this division, modulo arithmetic, will be of great significance. We will from now on say that if two polynomials, say $f(x)$ and $h(x)$, that are not necessarily the equal to each other, but after division by some polynomial $g(x)$ their remainders are both equal to $r(x)$, then $f(x)$ and $h(x)$ are equivalent (modulo $g(x)$). That is if $f(x) \bmod g(x) = r(x) = h(x) \bmod g(x)$. We denote this by $f(x) \equiv h(x) \pmod{g(x)}$.

A.4 Cyclic codes

As said before, Reed Solomon codes are cyclic codes, therefore in this chapter we will define cyclic codes followed by some useful properties of cyclic codes. First we will define what a cyclic shift of a code word is.

Definition A.13. *Let C be a code with* $\boldsymbol{v} = v_1v_2v_3...v_n$ *in C. We define the cyclic shift of* $\boldsymbol{v}$ *as* $\pi(\boldsymbol{v}) = v_nv_1v_2...v_{n-1}$.

Definition A.14. *Let C be a code. C is said to be a cyclic code, if for any arbitrary* $\boldsymbol{v} \in C$ *it holds that* $\pi(\boldsymbol{v}) \in C$.

Note that by using induction on Definition A.14 we see that if C is a cyclic code and $\mathbf{v} \in C$ it holds that $\pi^i(\mathbf{v}) \in C$ for $1 \leq i \leq n$. When we want to combine the polynomial notation we introduced in section A.3 seen and the concept of cyclic codes, we notice that a cyclic shift in a polynomial is nothing more than multiplying by x and looking at it mod $1 + x^n$. Note that our polynomials were (and in this way will stay) of degree n-1 and that $1 \equiv x^n \pmod{1 + x^n}$.

Example A.15. *Let C be a code and let $\boldsymbol{v} \in C$ than the polynomial corresponding to $\boldsymbol{v}$ is $v(x) = v_0 + v_1x + v_2x^2 + ... + v_{n-1}x^{n-1}$. Now we multiply $v(x)$ by x and find $xv(x) = v_0x + v_1x^2 + v_2x^3 + ... + v_{n-1}x^n$*
$= v_{n-1} + v_0x + v_1x^2 + ... + v_{n-2}x^{n-1} (\text{mod } 1 + x^n)$.

The remaining part of this chapter we focus on the minimal (non-zero) polynomial of a code we call the generator polynomial. We will proof that there is in fact a unique polynomial of minimal degree, and we will show how it 'generates' the code. The generator polynomial will later on be used to construct the parameters of a Reed Solomon code.
Let C be a cyclic linear code. We claim that there is one unique non-zero polynomial $g(x)$ of minimal degree. This is true due to the fact that if there was another polynomial $g'(x)$ with the same degree as $g(x)$ then the polynomial $g'(x) + g(x)$ (which is in C because of linearity) would have a degree of at least one less than themselves. However, since $g(x)$ and $g'(x)$ both had minimal degree, this can only be the case if the obtained polynomial is 0, but that would mean that $g'(x) = g(x)$. Thus, there is a unique polynomial of minimum degree in C.

Definition A.16. *Let C be a cyclic linear code. The generator polynomial $g(x)$ is the non-zero polynomial in C of minimal degree.*

We call it a generator polynomial because it generates a code C in a way that every code word $v(x) \in C$ has the property that $g(x)$ is a divisor of $v(x)$. That is, for $v(x) \in C$ there exists an $a(x)$ so that $v(x) = a(x)g(x)$. Also any multiplication of $g(x)$ with a random polynomial $a(x)$ is again a code word. For the next observation we take $v(x) \in C$ where words in C have length n. We say that $g(x)$ is of degree $n-k$ and we write $v(x) = a(x)g(x) = a_0g(x)+a_1xg(x)+...+a_{k-1}x^{k-1}g(x)$ (where $a_0, ..., a_{k-1} \in 0, 1$). From this notation we see that our arbitrary chosen $v(x) \in C$, and therefore any code word in C, is in the span $\langle g(x), xg(x), \ldots, x^{k-1}g(x)\rangle$. This shows two things. First, it shows that we can make a generator matrix for this cyclic linear code quite simply. Remember that there are more than one possible generator matrices for a code, however, the easiest one to find is most definitely notated as:

$$G = \begin{bmatrix} g(x) \\ xg(x) \\ \vdots \\ x^{k-1}g(x) \end{bmatrix}$$

Observe that from the generator polynomial we can construct the generator polynomial, and we can see the dimension k of the code without any work.
Second, it shows that any code words polynomial, multiplied by a random polynomial, is again a code words polynomial. Let C be a cyclic linear code and take $v(x) \in C$. We show that $c(x) = a(x)v(x)$ is again in C. Since generator polynomial $g(x)$ is a divisor of $v(x)$, say $v(x) = b(x)g(x)$ we write $c(x) = a(x)b(x)g(x)$ and since any multiplication of a random polynomial by $g(x)$ is again a code words polynomial we conclude that this not only holds for the generator polynomial, but for any code words polynomial.

A.5 BCH codes

BCH codes are cyclic polynomial codes over the finite field $GF(q^r)$ with q prime, in our case $q = 2$, and r a positive integer. This means words have length r. We move on to the concept of *constructing* $GF(2^r)$ using a *primitive polynomial*. We first look at an example and use this to discuss the concepts.

Example A.17. *We construct $GF(2^4)$ using the primitive polynomial $h(x) = 1 + x^3 + x^4$. Remember that in this case $1 + x^3 \equiv x^4 (\text{mod } 1 + x^3 + x^4)$. We take primitive element $\beta = x$ and calculate all the powers of β*

power of β	polynomial in x (mod h(x))	word
-	0	0000
β^0	1	1000
β	x	0100
β^2	x^2	0010
β^3	x^3	0001
β^4	$x^4 \equiv 1 + x^3$	1001
β^5	$x^5 \equiv x + x^4 \equiv 1 + x + x^3$	1101
β^6	$x^6 \equiv x + x^2 + x^4 \equiv 1 + x + x^2 + x^3$	1111
β^7	$x^7 \equiv 1 + x + x^2$	1110
β^8	$x^8 \equiv x + x^2 + x^3$	0111
β^9	$x^9 \equiv 1 + x^2$	1010
β^{10}	$x^{10} \equiv x + x^3$	0101
β^{11}	$x^{11} \equiv 1 + x^2 + x^3$	1011
β^{12}	$x^{12} \equiv 1 + x^2$	1010
β^{13}	$x^{13} \equiv x + x^2$	0110
β^{14}	$x^{14} \equiv x^2 + x^3$	0011
β^{15}	$x^{15} \equiv x^3 + x^4 \equiv 1$	1000

We see that every non-zero word of length 4 is expressed as a power of β where $\beta^{15} = 1 = \beta^0$ and the cycle starts over again (so $\beta^{16} = \beta$ and so on). In general, when $\beta^{2^r-1} = 1$ and $\beta^m \neq 1$ for $1 < m < 2^r - 1$ we call β is a primitive element. However, to be able to construct $GF(2^r)$ in such a way at all, we need a primitive polynomial as well. When $\beta = x$ is a primitive element of the polynomial, the polynomial is primitive. Such a construction is essential to both BCH and Reed Solomon codes.

Another concept that plays a great role in the classes of BCH and Reed Solomon codes is that of the *minimal polynomial*. One of the great advantages of Reed Solomon codes is that one can design the distance of the code (and therefore the amount of nodes that can be repaired) very easily. To do so, one needs a generator polynomial and a minimal polynomial, consequently we will discuss the matter of minimal polynomials next.

We first define an element α is a root of a polynomial $p(x)$ if and only if $p(\alpha) = 0$, and we notice that $K[x]$ is the set of all polynomials over K (so with all coefficients being either 0 or 1).

Definition A.18. *Let $\alpha \in GF(2^r)$. The minimal polynomial of α, $m_\alpha(x)$, is the polynomial in $K[x]$ of smallest degree having α as a root.*

Theorem A.19. *Let $\alpha \in GF(2^r)$. Then $\alpha, \alpha^2, \alpha^4, ..., \alpha^{2^{r-1}}$ are exactly all the roots of $m_\alpha(x)$.*

Example A.20. *When using the construction of Example A.17 we find that $\alpha = \beta^5$ gives the minimal polynomial $m_5 = (x - \alpha)(x - \alpha^2) = (x - \beta^5)(x - \beta^{10}) = \beta^{15} + \beta^5 x \beta^{10} x + x^2 = 1 + (1101)x + (0101)x + x^2 = 1 + x + x^2 + x^4 + x^2 + x^4 + x^2 = 1 + x + x^2$. Notice that since $\alpha^3 = \beta^{20} = \beta^5$ we stop at α^2. Also we see that $m_{10} = (x + \alpha)(x - \alpha^2) = (x - \beta^{10})(x - \beta^5)$ so $m_5 = m_{10}$. The rest of the powers of β can be used to calculate the other minimal polynomials*

in the same way. This gives:

$$\begin{aligned} m_1 = m_2 = m_4 = m_8 &= 1 + x^3 + x^4 \\ m_3 = m_6 = m_{12} = m_9 &= 1 + x + x^2 + x^3 + x^4 \\ m_5 = m_{10} &= 1 + x + x^2 \\ m_7 = m_{14} = m_{13} = m_{11} &= 1 + x + x^4 \end{aligned}$$

A.6 Reed Solomon codes

Reed Solomon (RS) codes and BCH codes are very much alike, but still RS codes are being used in practice a lot more than other BCH codes. The significant difference to the codes we have discussed before, is that the symbols of a code word were singular bits, where in Reed Solomon codes this is not the case anymore. In Reed Solomon codes when for instance we work over the field $GF(2^3)$, a symbol consists of 3 bits (which makes 8 possible symbols instead of 2) and all code words contain $2^3 - 1 = 7$ symbols. When for example we use the $GF(2^4)$ of Example A.17, a word can look like this $\beta^8\beta^5 1000000000000$ which makes it the word 0111.1101.1000.0000.0000.0000.0000.0000.0000.0000.0000.0000.0000.0000.0000 in terms of bits. We first formally define RS codes and then we will look at the parameters of the code and give an example. We close the chapter off with a theorem about shortening RS codes. Something which is done a lot in practice.

Definition A.21. *A binary Reed Solomon code $RS(2^r, d)$ is a cyclic linear code over $GF(2^r)$ with generator polynomial $g(x) = (\beta^{m+1} + x)(\beta^{m+2} + x)\dots(\beta^{m+d-1} + x)$ for $m \in \mathbb{N}$ and β a primitive element of $GF(2^r)$.*

We see that we can design the distance d of an RS code by choosing at what power of β we stop during building the generator polynomial. To summarize the parameters of an $RS(2^r, d)$ code:
$n = 2^r - 1$,
$k = 2^r - d$,
The amount of possible words $= 2^{rk}$
So we see that by choosing r and d, the other parameters immediately follow. Also a possible generator matrix is the same as we discussed in BCH codes, namely the rows represent the $k-1$ cyclic shifts of $g(x)$ and $g(x)$ itself. We will now look at an example of an RS code, where almost all concepts we have seen will come across.

Example A.22. *Let C be the $RS(8, 5)$ with generator polynomial $g(x) = (1 + x)(\beta + x)(\beta^2 + x)(\beta^3 + x)$ using $GF(2^3)$ constructed by using primitive polynomial $1 + x + x^3$. Then $r = 3, d = 5$ which gives:*
$n = 2^3 - 1 = 7$,
$k = 2^3 - 5 = 3$,
Amount of code words $= 2^{3*3} = 512$.
From the construction of $GF(2^3)$ by using $1 + x + x^3$, given in Appendix, we find that $g(x) = \beta^6 + x(\beta^6 + \beta^5 + \beta^4 + \beta^3) + x^2(\beta^5 + \beta^4 + \beta^3 + \beta^3 + \beta^2 + \beta) + x^3(1 + \beta + \beta^2 + \beta^3) + x^4 = \beta^6 + \beta^5 x + \beta^5 x^2 + \beta^2 x^3 + x^4$ which gives a generator matrix:

$$G = \begin{bmatrix} \beta^6 & \beta^5 & \beta^5 & \beta^2 & 1 & 0 & 0 \\ 0 & \beta^6 & \beta^5 & \beta^5 & \beta^2 & 1 & 0 \\ 0 & 0 & \beta^6 & \beta^5 & \beta^5 & \beta^2 & 1 \end{bmatrix}$$

Encoding for instance the message word $\boldsymbol{m} = \beta\beta^4 1$ gives the code word $\boldsymbol{c} = \boldsymbol{m}G = \beta^7\beta^4\beta^2 00\beta 1$

In practice RS codes are shortened a lot. We end this chapter by discussing how this affects the code. When we shorten the $RS(8,5)$ of the example from above we know from section A.2 that $n = 6, k = 2, d = 5$ and since there are 8 possible symbols in each of the three positions, and we now only have 2 positions left, we see that the amount of code words is still 2^{rk} which is now $2^{3*2} = 16$. We generalize shortening an RS code in the next theorem.

Theorem A.23. *Let C be an $RS(2^r, d)$ code with generator polynomial $g(x)$. The s-times shortened $RS(2^r, d)$ code has:*
$n = 2^r - 1 - s$
$k = 2^r - d - s$
Amount of code words $= 2^{rk}$
The generator matrix is

$$G = \begin{bmatrix} g(x) \\ xg(x) \\ \vdots \\ x^{k-1-s}g(x) \end{bmatrix}$$

B Repair of two failed nodes in the same group after Piggybacking

In this appendix we look at the the optimal repair scheme of two failed systematic nodes in the same group after Piggybacking the (14,10)-RS code of Section 4.2. We look at the situation where nodes 6 and 7 fail, which are both of group S_2. When we look at Table 3 we first note that the symbols in parity nodes consist of parts with a **p**-vector and parts with a **q**-vector. To refresh our knowledge about these vectors: the part with a **p**-vector is made out of a message vector $\mathbf{a}_j$ consisting of 10 systematic symbols in instance j, which is then multiplied by a Reed Solomon encoding vector **p**. A **q**-vector is also a vector of length 10 but with only a few entries matching to its corresponding **p**-vector and the rest zero's, depending on which group S_g it corresponds to. Now lets look at what we can do with this information. If we for example want to restore the first symbol of node 1, we could download the first parity symbol $\mathbf{p}_1^T\mathbf{a}_1$ together with the remaining 9 systematic symbols of this instance, and then subtract the downloaded 9 systematic symbols from this parity symbol. Now we are left with the first symbol of node 1. The lesson we learn from this is

Remark B.1. *Since a **p**-vector is of length 10, we always have to subtract 9 systematic symbols in order to be left with the 10^{th}.*

When we would do the same with a **q**-vector instead of a **p**-vector, we would only have to subtract 2 or 3 symbols, in order to be left with the 1 missing symbol. Without the piggybacking algorithm we would have to download 5 parity symbols with a **p**-vector (1 in each instance), together with 9 systematic symbols in each instance to be able to repair 1 failed node, leaving the repair with a repair bandwidth of 100%. When 2 systematic nodes fail we would need to download 2 parity symbols in each instance, together with the 8 remaining systematic symbols, again with a repair bandwidth of 100%, and so on. We deduce from this that the only option for reducing our repair bandwidth is to make use of the **q**-vectors as much as possible, and avoid using **p**-vectors. To this end, we will (same as in the repair of one failed node) download the parity symbols from instance 4 from node 12, instance 3 from node 13, and instance 5 from node 14 because they are the only symbols that contain $\mathbf{q}_{i,2}$. These are

$$\mathbf{p}_2^T\mathbf{a}_4 + \mathbf{q}_{2,2}^T\mathbf{v}_2 \qquad \mathbf{q}_{3,2}^T\mathbf{a}_3 - (\mathbf{p}_3^T\mathbf{a}_4 + \mathbf{p}_3^T\mathbf{a}_5) \qquad \mathbf{p}_4^T\mathbf{a}_5 + \mathbf{q}_{4,2}^T\mathbf{v}_4$$

We see that these also contain **p**-vectors, but only from instances 4 and 5. This leads us to the unavoidable downloading of the 8 remaining symbols from both instances and subtract these from the terms. Now we are left with only the parts with a **q**-vector. Remember that $\mathbf{q}_{i,2}$ means that these terms consist of only symbols in group S_2. This gives us only 1 option for efficient repair. Since node 5 is still available, we download the three symbols in this node from instance 1, 2 and 3 and we subtract these from the parts with the **q**-vectors which leaves us with the following:

$$\begin{cases} p_{2,6}(a_{6,4} + a_{6,3} + 2a_{6,2} + 4a_{6,1}) + p_{2,7}(a_{7,4} + a_{7,3} + 2a_{7,2} + 4a_{7,1}) \\ p_{3,6}(a_{6,3} + a_{6,4} + a_{6,5}) + p_{3,7}(a_{7,3} + a_{7,4} + a_{7,5}) \\ p_{4,6}(a_{6,4} + a_{6,3} + 4a_{6,2} + 16a_{6,1}) + p_{4,7}(a_{7,4} + a_{7,3} + 4a_{7,2} + 16a_{7,1}) \end{cases}$$

Here $p_{i,j}$ is the j^{th} entry of the vector $\mathbf{p}_i$. What we see is that we have 3 equations, with 10 symbols unknown. In other words, we will need to restore 7 symbols in another way, or we need to add 7 more equations in order to be able to use this system of equations. Let us review our possibilities. In the next table we see the result of the Piggybacking encoding scheme like in

Table 3, where an empty entry represents a symbol we already downloaded and an x represents a missing symbol that we want to restore. In other words, here we see the remaining options we have for downloadable symbols.

Node 1	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$		
Node 2	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$		
Node 3	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$		
Node 4	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$		
Node 5					
Node 6	x	x	x	x	x
Node 7	x	x	x	x	x
Node 8	$a_{8,1}$	$a_{8,2}$	$a_{8,3}$		
Node 9	$a_{9,1}$	$a_{9,2}$	$a_{9,3}$		
Node 10	$a_{10,1}$	$a_{10,2}$	$a_{10,3}$		
Node 11	$\mathbf{p}_1^T\mathbf{a}_1$	$\mathbf{p}_1^T\mathbf{a}_2$	$\mathbf{p}_1^T\ \mathbf{a}_3$	$\mathbf{p}_1^T\mathbf{a}_4$	$\mathbf{p}_1^T\mathbf{a}_5$
Node 12	$\mathbf{p}_2^T\mathbf{a}_1$	$\mathbf{p}_2^T\mathbf{a}_2$	$\mathbf{q}_{2,1}^T\mathbf{a}_3$ - $(\mathbf{p}_2^T\mathbf{a}_4 + \mathbf{p}_2^T\mathbf{a}_5)$		$\mathbf{p}_2^T\mathbf{a}_5 + \mathbf{q}_{2,3}^T\mathbf{v}_2$
Node 13	$\mathbf{p}_3^T\mathbf{a}_1$	$\mathbf{p}_3^T\mathbf{a}_2$		$\mathbf{p}_3^T\mathbf{a}_4 + \mathbf{q}_{3,1}^T\mathbf{v}_3$	$\mathbf{p}_3^T\mathbf{a}_5 + \mathbf{q}_{3,3}^T\mathbf{v}_3$
Node 14	$\mathbf{p}_4^T\mathbf{a}_1$	$\mathbf{p}_4^T\mathbf{a}_2$	$\mathbf{q}_{4,3}^T\mathbf{a}_3$ - $(\mathbf{p}_4^T\mathbf{a}_4 + \mathbf{p}_4^T\mathbf{a}_5)$	$\mathbf{p}_4^T\mathbf{a}_4 + \mathbf{q}_{4,1}^T\mathbf{v}_4$	

Table 13: *An overview of downloadable symbols. An empty entry represents a symbol we have already downloaded and an x marks a missing symbol that we are trying to restore.*

What we are trying is to do is restore the missing symbols without downloading all $a_{i,j}$'s. We have 3 types of options for downloading a parity symbol in order to add an equation to our system:
1. If we download node 11, instance 1, in order to add an equation, we immediately need to download all 7 remaining $a_{i,1}$ for subtraction. However, when we downloaded node 11 instance 1, we can download node 12 instance 1 without having to download any extra symbols (of course, proceeding by downloading also node 13 instance 1 is not an option since there are only 2 symbols missing from instance 1). The same holds for instance 2.
2. The second type is downloading node 11, instances 4 and 5. These are what we can call 'free' equations because the systematic symbols of these instances are already downloaded.
3. The last type is to download one of the parity symbols in from nodes 12, 13, 14 in instances 3, 4, 5. These require the download of a part of the remaining downloadable systematic symbols, since we need to subtract the **q**-vector parts to be left with another of our so called free equations (e.g., when we choose to download $\mathbf{p}_3^T\mathbf{a}_4 + \mathbf{q}_{3,1}^T\mathbf{v}_3$, we also need to download the systematic symbols of nodes 1, 2, 3 and 4, to subtract the $\mathbf{q}_{3,1}^T\mathbf{v}_3$ part). However, if we choose this option, we must avoid downloading two symbols where one requires the download of systematic symbols of group 1, and the other requires the download of the symbols of group 3, because then our repair bandwidth automatically is above 100%.
Since the parity symbols of node 11, instance 4 and 5 are the only symbols that do not require any extra download, we choose these two in any case first. We add these two to our system of equations, which leaves us at having to add 5 more equations. Now we proceed by noticing that the download of instance 1 node 11 and 12 requires 7 systematic symbols, the download of instance 2 node 11 and 12 requires 7 systematic symbols, and the download of instance 3 node 12 and 14 require 4 and 3 systematic symbols respectively. However, one can see that when we choose the last of these three options, we can add a free equation in node 11 instance 3. Shortly: if we choose the three parity symbols from instance 3, which requires only the download of the systematic symbols of instance 3, and then we choose to download the first two parity symbols

from instance 1, which also requires downloading 7 symbols, thus saving the download of 7 systematic symbols in instance 2.

We have now found an upper bound for the repair bandwidth, which is 86%. We will now show that this is also the lower bound. For that, we go back one step where we are in the situation of Table 13, just after downloading the parity symbols of node 11, instance 4 and 5, because up to that point we did not make any choices ourselves. Now, we remember that there is no point in downloading 3 parity symbols from instance 1 and 2, we can download a maximum of 2 parity symbols in these instances. We are for the same reasons limited in downloading parity symbols concerning the other three instances, and we already downloaded a parity symbol concerning instances 4 and 5. We know that, in order to save 8 or more symbols, we must save downloading systematic symbols in at least two different instances. From this point in the downloading process, we need to download 5 more parity symbols to add to the system of equations. We will review the options:
- We cannot download a symbol of instance 4, because then we are left with 9 systematic symbols, and we have 4 more equations to add, which is not possible.
- If we download a parity symbol from instance 5, we are left with 3 times 4 systematic symbols, meaning that we have to add 4 more equations with downloading only from 1 instance, which is not possible.
- If we download 2 parity symbols from instances 1, we cannot download from instance 2 and vice versa.
This means that our found repairing method was the optimal method. We might have one more concern, which is that we Piggybacked the parity nodes with $m = 2$, which means that the structure of the second systems differs from the first. One can however check that this does not affect the repair bandwidth with our repair method.

References

[1] K. V. Rashmi, N. B. Shah, H. K. D. Gu, D. Borthakur, and K. Ramchandran, *A Solution to the Network Challenges of Data Recovery in Erasure-coded Dis tributed Storage Systems: A Study on the Facebook Warehouse Cluster.* Proc. 5th USENIX Workshop on Hot Topics in Storage and File Systems, 2013.

[2] B. S.B., *Erasure Codes for Distributed Storage: Tight Bounds and Matching Constructions.* Electrical Communication Engineering Indian Institute of Science Bangalore, 2018.

[3] D. Hankerson, D. Hoffman, D. Leonard, C.C.Lindner, K.T.Phelps, C. Rodger, and J. Wall, *Coding Theory and Cryptography, the Essentials.* CRC Press, 2000.

[4] K. V. Rashmi, N. B. Shah, and K. Ramchandran, *A Piggybacking Design Framework for Read-and Download-efficient Distributed Storage Codes.* Fellow, IEEE Department of Electrical Engineering and Computer Sciences University of California, Berkeley, 2013.

[5] H. Dau, I. Duursma, M. Kiah, and O. Milenkovic, *Optimal repair schemes for Reed-Solomon codes with single and multiple erasures.* Information Theory and Applications Workshop, San Diego, 2017.

[6] M. N. Krishnan, N. Prakash, V. Lalitha, B. Sasidharan, P. V. Kumar, S. Narayanamurthy, R. Kumar, and S. Nandi, *Evaluation of Codes with Inherent Double Replication for Hadoop.* Indian Institute of Science, Bangalore and NetApp Inc., 2014.

[7] Q. Xin, E. L. Miller, T. Schwarz, D. D. Long, S. A. Brandt, and W. Litwin, *Reliability Mechanisms for Very Large Storage Systems.* IEEE MSST, 2013.