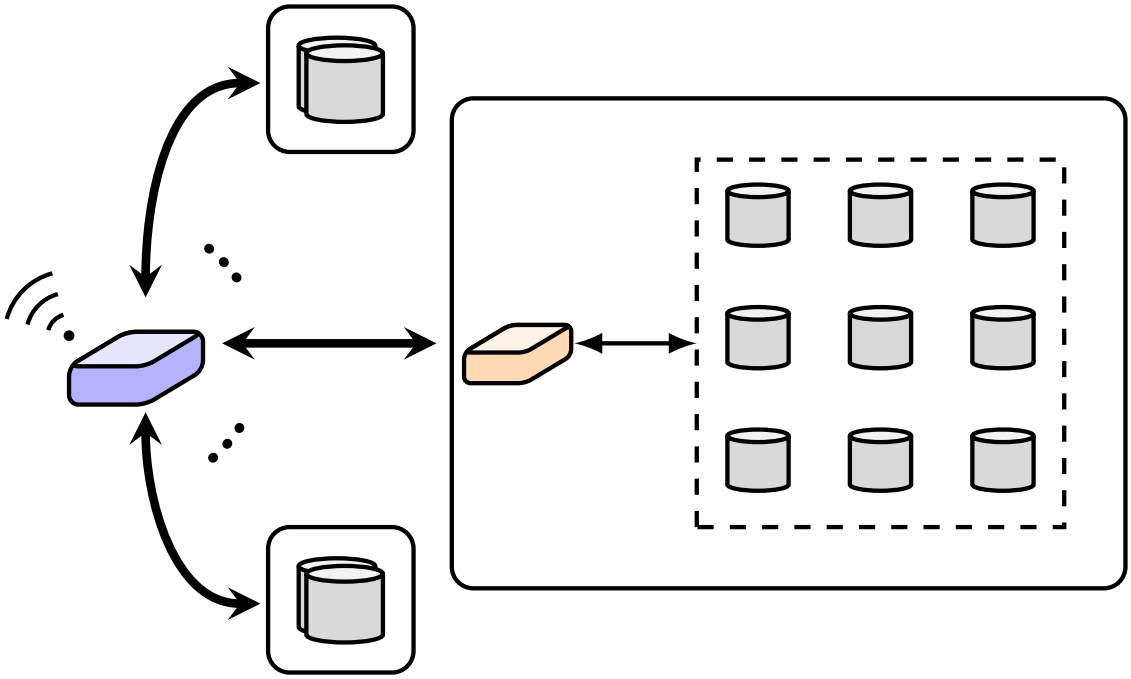
CHALMERS
UNIVERSITY OF TECHNOLOGY



Repair Efficient Erasure Correcting Codes for Distributed Storage Systems

Master's thesis in Communication Engineering

SIDDHARTHA KUMAR

Department of Signals and Systems
Chalmers University of Technology
Gothenburg, Sweden 2015

Master's thesis EX042/2015

Repair Efficient Erasure Correcting Codes for Distributed Storage Systems

SIDDHARTHA KUMAR

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Signals and Systems
Division of Communication Systems
Chalmers University of Technology
Gothenburg, Sweden 2015

Master Thesis for the Master Programme "Communication Engineering"
Repair Efficient Erasure Correcting Codes for Distributed Storage Systems
SIDDHARTHA KUMAR

Master's Thesis EX042/2015
Department of Signals and Systems
Division of Communication Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Architecture of a typical distributed storage system present in data centers. The cylinders denote the storage nodes while the colored boxes denote the nodes that perform file handling.

Typeset in LaTeX
Gothenburg, Sweden 2015

Abstract

The current age of information technology is characterized by an ever increasing presence of digital data in the world. Digital data has become an integral part of our lives in the form of social networking, online streaming and accessing crucial data on the go. The huge amount of data generated needs to be stored in an inexpensive and reliable way.

Distributed storage (DS) is a technology that stores data on a network of inexpensive storage devices, referred to as nodes, thereby lowering the cost of storage. However, such storage nodes are prone to failures, which leads to unavailability of the stored data. The ability to tolerate multiple node failures is defined as fault tolerance. The addition of redundancy in DS systems allows for the recovery of the lost data by guaranteeing fault tolerance. The easiest way to achieve this, is to replicate the data in a system, i.e., the data is copied over several nodes. However, replication schemes store data inefficiently. Hence, the storage industry has started moving towards erasure correcting codes (ECCs) that store data much more efficiently. Essentially, data in DS systems is available as long as the number of node failures does not exceed its limit of fault tolerance. When a node fails, to maintain the initial level of availability, another node needs to be populated with the lost data. This is referred to as repair. By using ECCs, one gains in storage efficiency but looses in repair performance, namely in such parameters as repair bandwidth and repair complexity. Therefore, in recent years, the research has been focused on designing ECCs for DS that perform repair efficiently.

In this thesis, we present the construction of a new family of ECCs for DS that yield low repair bandwidth and low repair complexity for a single failed data node. In particular, we present a systematic construction based on two classes of parity symbols. The primary goal of the first class of symbols is to provide good erasure correcting capability, while the second class facilitates node repair, reducing the repair bandwidth and the repair complexity. Lastly, we compare the proposed codes with Minimum Disk I/O codes, Zigzag codes, piggyback codes and local reconstruction codes that are proposed in the literature.

Keywords: Erasure correcting codes, distributed storage, repair bandwidth, repair complexity

Acknowledgements

First, I would like to express my deepest gratitude towards my supervisor and examiner Alexandre Graell i Amat, who at each step encouraged and motivated me to reach excellence. Without his guidance, I would not have been able to overcome the challenges I faced. I am also grateful towards my co-supervisors Iryna Andriyanova and Fredrik Brännström who where always approachable and did their very best to guide me throughout my thesis.
I would like to thank my parents who provided me an opportunity to study in Sweden. Even though they were not physically present, they always provided me with emotional support.
Last, I would like to thank all my friends who always supported me in my good and bad times.

Siddhartha Kumar
Gothenburg, June 15, 2015

Contents

Acronyms

DS	Distributed Storage
ECC	Erasure Correcting Codes
GFS	Google File System
HDFS	Hadoop Distributed File System
LRC	Local Reconstruction Codes
MDR	Minimum Disk I/O repairable
MDS	Maximum Distance Separable
P2P	peer-to-peer
QFS	Quick File System
RAID	Redundant Array of Independent Disks
WAS	Windows Azure Storage

Notations

x	Scalar
$\boldsymbol{x}$	Vector
$\boldsymbol{x}^\mathsf{T}$	Transpose of $\boldsymbol{x}$
$\mathcal{X}$	Set
$\boldsymbol{X}$	Matrix
$\mathbb{F}_{q^p}$	Galois field of size q^p
$(a+b)_k$	Summation over modulo k

Chapter 1

Introduction

The advent of the digital revolution during the late 20^{th} century, marked the onset of the information age in human history. Central to this are the advances in miniaturization of logic circuits that have lead to popularization of mobile phones, personal computers and other wireless devices. This in-turn has lead to an exponential increase in the amount of digital data. In a study conducted by EMC, it was predicted that by the end of this decade, approximately 40000 exabytes ($1\,\text{EB} = 10^{18}$ bytes) of digital data would be stored yearly [1]. Nowadays, it is common to see Internet companies such as Google or Facebook to handle terabytes ($1\,\text{TB} = 10^{12}$ bytes) of data each day. In fact, Facebook stores petabytes ($1\,\text{PB} = 10^{15}$ bytes) of data from its Internet based social network website [2]. To store such large amounts of data on a single storage device is impractical due to high costs and reliability issues.

This has lead to the emergence of a new concept called distributed storage (DS). DS systems consist of a network of numerous, relatively inexpensive storage devices (or simply storage nodes) where the data is stored in a distributed fashion. Storing data in this fashion introduces scalability, i.e., the means of maintaining stable performance for steadily growing data collection by adding new resources to the system [3]. For instance, to increase the storage capacity of a DS system, one just needs to add new storage nodes to the network without the fear of degrading the performance of the system. However, these storage nodes are subject to electrical and mechanical failures [4]. In some cases, network outage due to sudden influx of traffic in the network can also lead to node failures. Therefore, it is important to prevent the loss of data in such systems. To achieve the ability to recover the lost data from failed nodes requires the addition of redundancy to the system. The easiest way to achieve fault tolerance in DS systems is by introducing n-replication schemes. In particular, DS systems such as the Google File System (GFS) [5] and the Hadoop Distributed File System (HDFS) [6] employ 3-replication scheme, i.e., they replicate the data three times. As a result, the system can tolerate up to two storage node failures. There is of course a price to pay: adding redundancy, increases the amount of data that needs to be stored. As it turns out, replication schemes are inefficient when it comes to the amount of redundancy introduced.

A more efficient alternative is to employ parity schemes such as the Redundant Array of Independent Disks (RAID) [7] as the amount of redundancy added is less. However, RAID schemes are not reliable in the case when there are more than 2 node failures. In this regard, erasure correcting coding schemes are a better alternative as they require less redundancy as compared to replication and can deal with more node failures than RAID.

1.1 Coding for DS Systems

An (n, k) erasure correcting code (ECC) of code rate $R = k/n$, encodes k data symbols to obtain $n > k$ code symbols. The $n-k$ redundancy symbols added serve the purpose of recovering failures. An (n, k) ECC can repair up to $d_{\min} - 1$ simultaneous symbol erasures, where $d_{\min}$ is the code's minimum hamming distance. From a coding perspective, a replication scheme can be seen as an $(n, 1)$ repetition code, where 1 data symbol is replicated n times. These kind of codes, though they are easy to implement, suffer from a high storage overhead, $n/k = n$. In other words, n symbols need to be stored for each data symbol. This directly translates to high cost of storage. An alternative to the replication scheme are the classical (n, k) maximum distance separable (MDS) codes. MDS codes are ECCs that satisfy the singleton bound, $d_{\min} \leq n - k + 1$, with equality. This implies MDS codes have the best fault tolerance-storage overhead tradeoff [8]. For instance, a $(9, 7)$ MDS code and a $(3, 1)$ repetition code have the same reliability (can correct upto 2 simultaneous failures) but the former has storage overhead of 1.29 compared to latter's 3. This has lead the industry to move from replication schemes towards ECC in DS systems. More specifically, storage systems like GFS II (successor of GFS by Google) and Quick File System (QFS) use $(9, 6)$ Reed-Solomon (RS) code (also an MDS code) [9].

In a DS system, the n code symbols are typically stored in n nodes. Therefore, the system can tolerate up to $d_{\min} - 1$ node failures. The number of failures that the system can tolerate is referred to as fault tolerance. We also say that an (n, k) ECC introduces a storage overhead of n/k. Essentially, an ECC ensures data availability as long as node failure does not exceed the fault tolerance limit. Therefore, to maintain the availability of data over long periods of time, failed nodes need to be repaired. In general, classical ECCs do not perform efficient repair. Repair efficiency is parameterized as repair access and repair bandwidth. Repair access is defined as the number of nodes that need to be contacted for the repair of a failed node, while repair bandwidth is defined as the amount of information (in bits) that is communicated through the network for the repair of a single node. A classical ECC has a high repair access and repair bandwidth as it entails reading $k' \geq k$ symbols in k' nodes for a repair of a failed node. For

example, an (n, k) MDS code has a repair access of k and a repair bandwidth of $k \times \#\,\mathsf{symbols}\,/\,\mathsf{node} \times \mathsf{size\,of\,symbol}$. In contrast, an $(n,1)$ repetition code has a repair access of 1 and repair bandwidth of $1 \times \#\,\mathsf{symbols}\,/\,\mathsf{node} \times \mathsf{size\,of\,symbol}$.

During the repair process, the nodes involved in the repair are not available for other network processes (e.g., download). Therefore, a high repair access harms the data availability. On the other hand, a high repair bandwidth incurs a high data traffic. As a consequence, the repair process would dominate system's resources. Therefore, in recent years, the research has been focused in designing ECCs that achieve low repair access and low repair bandwidth.

1.2 Erasure Correcting Codes

Pyramid codes introduced in [10] were the first codes to address the problem of high repair access. These codes sacrificed on the fault tolerance to reduce the repair access. In [11], it was shown that these codes were optimal in terms of fault tolerance/repair access. On the downside, these codes needed to be constructed from a large galois field that led to higher complexity. Local reconstruction codes (LRCs) introduced in [12] are constructed from an MDS code. The parity symbols are modified such that they reduce repair access. Unlike the pyramid codes, LRCs needed to be constructed from a smaller galois field. LRCs are currently being used in the Windows cloud storage platform, Windows Azure Storage (WAS). Other known codes that reduce the repair access are the locally repairable codes [2], [13]. As with LRCs, these codes are modified MDS codes.

In [14], the authors discussed the effects of a high repair bandwidth and identified a tradeoff between storage and repair bandwidth. In particular, they provided theoretical bounds on the repair bandwidth for any code. Codes such as minimum disk I/O repairable (MDR) codes [15], zigzag codes [16] and the piggyback framework [17] aim at reducing the repair bandwidth. MDR codes are a family of RAID-6 codes. This means that they have a reliability of 2. These codes provide optimal repair bandwidth for a given code rate and at the same time have low complexity. On the other hand, Zigzag codes are a family of variable rate ECCs that have optimal repair bandwidth for a given rate but have high complexity. Both MDR and Zigzag codes are a class of MDS codes. The piggyback framework is applied to an existing code to reduce the repair bandwidth. The drawback of such a framework is that it reduces the reliability of the existing code.

In Table 1.1, we have summarized the codes that have been discussed[1]. The

[1]In Table 1.1, ν is the size of the data symbol in bits. A [k,l,g] LRC code encodes k symbols to get $n = k + l + g$ code symbols, where l denotes the number of local parity symbols that are encoded by k/l data symbols. g denotes the number of global parity symbols that are encoded by k data symbols. An [n,k,r] locally repairable code encodes k symbols to get n code symbols having repair access r. An $[n, k\ r]$ piggyback code modifies $r-1$ parity nodes of an $[n, k]$ base code

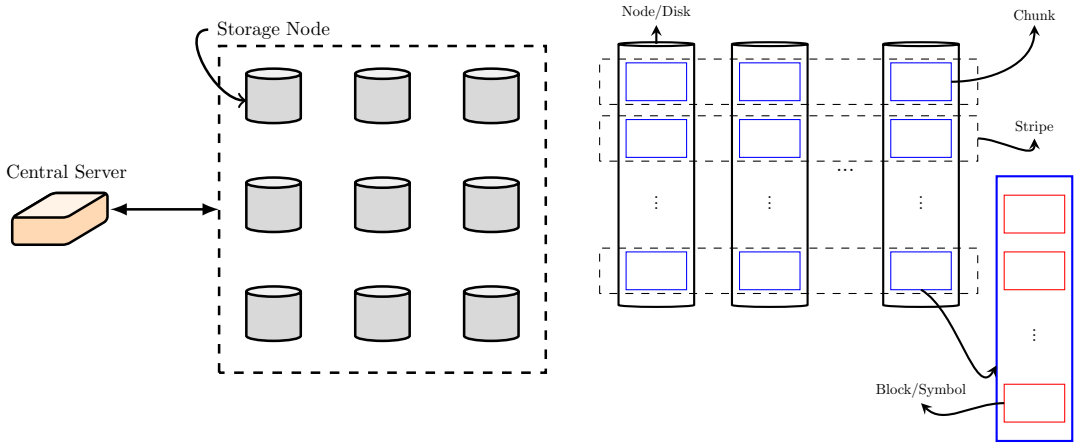
Code Family	Objective	Fault Tolerance	Overhead	RA	RB
MDS (n, k)	Reliability	$n-k$	$\frac{n}{k}$	k	$k\nu$
Pyramid (n, k) [10]	Reduce RA	MDS	$\frac{n}{k}$	–	–
LRC $[k, l, g]$ [12]	Reduce RA	$g+1$	$\frac{k+l+g}{k}$	$\frac{k}{l}$	$\frac{k\nu}{l}$
Locally Repairable Code $[n, k, r]$ [2]	Reduce RA	MDS	$\frac{n}{k}$	r	$r\nu$
Locally Repairable Code $[n, k, r]$ [13]	Reduce RA	MDS	$\frac{(r+1)n}{rk}$	r	$r\nu$
MDR $(k+2, k)$ [15]	Reduce RB	2	$\frac{k+2}{k}$	k	$\frac{(k+1)\nu}{k}$
Zigzag (n, k) [16]	Reduce RB	$n-k$	$\frac{n}{k}$	k	$\frac{n-1}{n-k}\nu$
Piggyback $[n, k, r]$ [17]	Reduce RB	$n-k-r+1$	$\frac{n}{k}$	k	Reduction of 25% to 50%

Table 1.1: Summary of different ECCs present in literature. RA and RB denote repair access and repair bandwidth respectively.

repair access and repair bandwidth reported in the table are normalized per data symbol. Fault tolerance of pyramid and locally repairable codes depend on the underlining smaller MDS codes from which they are constructed. In the case of piggyback codes, they provide a minimum fault tolerance of 1. This fault tolerance depends upon the repair bandwidth. Thus, a better repair bandwidth would mean a lower fault tolerance achieved by the code. Modifying the parity symbols by using piggybacks in the fashion as introduced in [17] leads to a reduction of repair bandwidth by 25% to 50%. It is easily seen from the table that Zigzag codes provide the best repair bandwidth for a given code rate and fault tolerance.

1.3 System Model

In general, DS systems fall into two main categories which are data centers and peer-to-peer (P2P) storage/backup systems [18]. While data centers like GFS [5] and HDFS [6], comprise of a network of clusters, each containing thousands of



(a) A Cluster **(b)** Data Layout

Figure 1.1: Architecture of a DS system

nodes. P2P systems like Wuala[2] consists of a network of nodes that are geographically distributed. In this section, we describe the architecture of a DS system for data centers.

A DS system used in data centers consists of a collection of clusters. Each cluster is a collection of interconnected nodes that are connected to a central server as seen in Fig. 1.1(a). The function of the central server is to manage the data storage. This is to say that it handles the creation, deletion and encoding schemes of the data. A file that needs to be stored is first divided into blocks (called symbols) of a fixed size (For example, 1 MB [9]). These symbols are then stored in groups referred to as chunks in a node. Each chunk is of fixed size, default being 64 MB in GFS [5] and HDFS [9]. Each node contains a number of chunks and a row of chunks across the nodes is referred as stripe, as seen in Fig. 1.1b. An (n, k) ECC is employed over a cluster once the data is stored across the nodes. This means that k data symbols from k nodes across the stripe are encoded to get $n-k$ parity symbols that are stored across $n-k$ nodes.

Consider a DS system that employs an (n, k) erasure code to store data reliably. In what follows, the parameters used in this thesis that characterize the performance of ECC in the context of DS are presented and formally defined.

1. **Fault tolerance:** Fault tolerance of an erasure code is defined as the number of simultaneous node failures that a DS system can tolerate. For a DS system that employs an (n, k) coding scheme, with minimum distance $d_{\min}$, its fault tolerance is $f = d_{\min} - 1$.

[2] `www.wuala.com/en/learn/technology`

2. **Storage overhead:** This parameter characterizes the amount of symbols that need to be stored on the DS system per data symbol to maintain the given fault tolerance of the system [12]. In the case when coding is not employed, the storage overhead is 1. When an (n, k) erasure code is employed, the storage overhead is n/k.

3. **Repair bandwidth:** It is defined as the amount of information in bits that needs to be read from the DS system to repair a failed node [14]. More formally, for a DS system storing β symbols per node, the repair bandwidth, γ is defined as

$$\gamma = \nu\beta\lambda \tag{1.1}$$

   where ν is the number of bits per symbols and λ is the total number of symbols read for he recovery of a failed symbol. DS systems having low repair bandwidth are preferable as they can repair faster.

4. **Complexity:** The term complexity determines the amount of resource required for the completion of a task. This resource may be time, space, operations etc. In this thesis, this resource is the number of operations. Basic tasks such as addition and multiplications of symbols of size ν, require $O(\nu)$ and $O(\nu^2)$ bit operations, respectively [19]. Here, we consider two aspects pertaining to complexity. The first being the encoding complexity, which is defined as the number of operations that needs to be performed to to obtain the parity symbols in each stripe. The second is the repair complexity, which is defined as the number of operations performed to repair a failed symbol. Complexity can be directly translated to hardware costs. Therefore, it is preferable for DS systems to have a low complexity ECC.

1.4 Contributions and Thesis Outline

In this thesis, we present and analyze a new family of ECCs that achieve low repair complexity and repair bandwidth for a single failed data node. In particular, we present a systematic construction based on two classes of parity symbols. Correspondingly, there are two classes of parity nodes.

The first class of parity nodes, referred as Class A nodes, are constructed from an MDS code having a fixed fault tolerance. These codes are then modified using the concept of piggybacking, introduced in [17]. Piggybacking considers adding carefully chosen linear combinations of data symbols (called piggybacks) to the parity symbols of a given erasure correcting code. The primary aim of the parity symbols of Class A is to provide fault tolerance to the DS system.

The second class of parity node, referred to as Class B nodes, are constructed from a block code whose parity symbols are obtained with simple additions. These parity symbols facilitate low complexity repair of a failed data node using low repair bandwidth. In addition to this, Class B nodes offer rate compatibility, i.e., storage overhead can be reduced at the expense of extra repair bandwidth.

We compare the proposed codes with MDR codes, Zigzag codes, piggyback codes and LRCs in terms of repair bandwidth and repair complexity.

We present a couple of notations that would be used throughout the document. We define the operator $(a+b)_k \triangleq (a+b) \mod k$. The Galois field of order q^p is denoted by $\mathbb{F}_{q^p}$.

The remainder of the thesis is organized as follows. In Chapter 2, the code construction for both Class A and Class B parity symbols is presented. We then provide a decoding schedule for the repair of a failed node. In Chapter 3, we analyze the fault tolerance, complexity and repair bandwidth of the proposed codes. We conclude this report by summarizing the work done and presenting some open problems that need to be considered for future in Chapter 4.

Chapter 2

Code Construction

In the previous chapter, the concept of DS was introduced. To ensure the reliability of these storage systems, the concept of ECC is applied. In this chapter, a new family of ECC that has low repair bandwidth and low repair complexity is presented.

2.1 Preliminaries

We consider the distributed storage system depicted in Fig. 2.1. There are k data nodes, each containing a very large number of data symbols over $\mathbb{F}_{q^p}$. As we shall see in the sequel, the proposed code construction works with blocks of k data symbols per node. Thus, without loss of generality, we assume that each node contains k data symbols. We denote by $d_{i,j}$, i,$j = 0,\ldots,k-1$, the ith data symbol in the jth data node. We say that the data symbols form a $k \times k$ *data array* $\boldsymbol{D}$, where $d_{i,j} = [\boldsymbol{D}]_{i,j}$. For later use, we also define the set of data symbols $\mathcal{D} \triangleq \{d_{i,j}\}$. Further, there are $n-k$ parity nodes each storing k parity symbols. We denote by $p_{i,j}$, $i = 0,\ldots,k-1$, $j = k,\ldots,n-1$, the ith parity symbol in the jth parity node. We further define the set $\mathcal{P}_j$ as the set of parity symbols in the jth parity node. The set of all parity symbols is denoted by $\mathcal{P} \triangleq \cup_j\{\mathcal{P}_j\}$. We say that the data and parity symbols form a $k \times n$ *code array* $\boldsymbol{C}$, where $c_{i,j} = [\boldsymbol{C}]_{i,j}$. Note that $c_{i,j} = d_{i,j}$ for $i,j = 0,\ldots,k-1$ and $c_{i,j} = p_{i,j}$ for $i = 0,\ldots,k-1$, $j = k,\ldots,n-1$.

Our main goal is to construct codes that yield low repair bandwidth and low repair complexity of a single failed systematic node. To this purpose, we construct a family of systematic (n,k) codes consisting of two different classes of parity symbols. Correspondingly, there are two classes of parity nodes, referred to as Class A and Class B parity nodes, as shown in Fig. 2.1. Class A and Class B parity nodes are built using an (n_{A},k) code and an (n_{B},k) code, respectively, such that $n = n_{\mathsf{A}} + n_{\mathsf{B}} - k$. In other words, the parity nodes of the (n,k) code[1] correspond to the parity nodes of Class A and Class B codes. The primary goal of Class A parity

[1]With some abuse of language we refer to the nodes storing the parity symbols of a code as the parity nodes of the code.

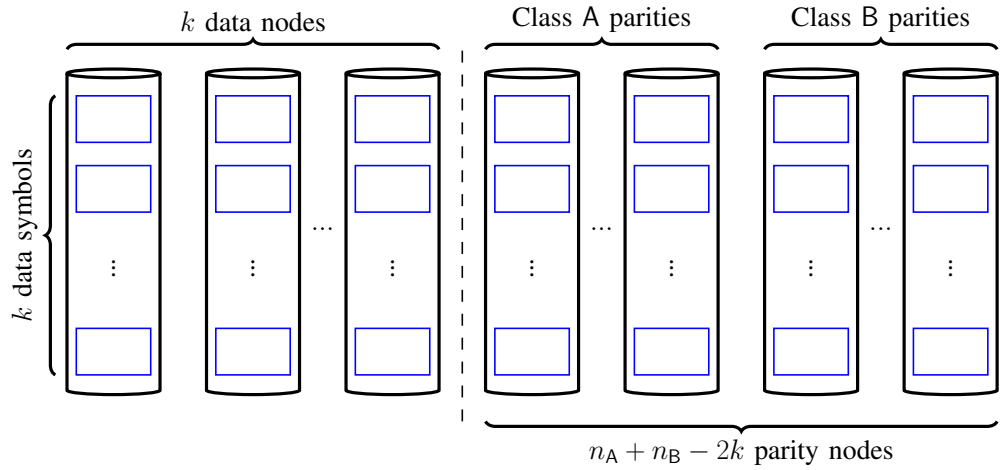


Figure 2.1: System model.

nodes is to achieve good erasure correcting capability, while the purpose of Class B nodes is to yield low repair bandwidth and low repair complexity. In particular, we focus on the repair of data nodes. Note that the repair bandwidth (in bits) per node, denoted by γ, is proportional to the average number of symbols (data and parity) that need to be read to repair a data symbol, denoted by λ. Then,

$$\lambda = \frac{\gamma}{\nu\beta} = \frac{\gamma}{\nu k}, \tag{2.1}$$

where $\nu = \lceil \log_2 q^p \rceil$ is the size (in bits) of a symbol. Note that λ can also be interpreted as the repair bandwidth normalized by the size (in bits) of a node. Therefore, in the rest of the paper λ is used to refer to the normalized repair bandwidth.

The main principle behind the code construction is the following. The repair is performed one symbol at a time. After the repair of a data symbol is accomplished, the symbols read to repair that symbol are cached in the memory. Therefore, they can be used to repair the remaining data symbols at no additional read cost. The proposed codes are constructed in such a way that the repair of a new data symbol requires a low additional read cost (defined as the number of additional symbols that need to be read to repair the data symbol), so that λ (hence γ) is reduced. Since we will often use the concepts of read cost and additional read cost in the remainder of the paper, we define them in the following definition.

Definition 2.1. The *read cost* of a symbol is the number of symbols that need to be read to repair the symbol. The *additional read cost* of a symbol is the additional number of symbols that need to be read to repair the symbol, considering that other symbols are already cached in the memory (i.e., have been read to recover some other data symbols previously).

2.2 Class A Parity Nodes

Class A parity nodes are constructed using a modified (n_{A}, k) MDS code, $k+2 \leq n_{\mathsf{A}} < 2k$, over $\mathbb{F}_{q^p}$. In particular, we start from an (n_{A}, k) MDS code and apply piggybacks [17] to some of the parity symbols. The construction of Class A parity nodes is performed in two steps as follows.

1) Encode each row of the data array using an (n_{A},k) MDS code (the same for each row). The parity symbol $p_{i,j}^{\mathsf{A}}$ is[2]

$$p_{i,j}^{\mathsf{A}} = \sum_{l=0}^{k-1} \alpha_{l,j} d_{i,l},\ j = k, \ldots n_{\mathsf{A}} - 1, \tag{2.2}$$

where $\alpha_{l,j}$ denotes a coefficient in $\mathbb{F}_{q^p}$. Store the parity symbols in the corresponding row of the code array. Overall, $k(n_{\mathsf{A}} - k)$ parity symbols are generated.

2) Modify some of the parity symbols by adding piggybacks. Let τ, $1 \leq \tau \leq n_{\mathsf{A}} - k - 1$, be the number of piggybacks introduced per row. The parity symbol $p_{i,u}$ is obtained as

$$p_{i,u}^{\mathsf{A}} = p_{i,u}^{\mathsf{A}} + d_{(i+u-n_{\mathsf{A}}+\tau+1)_k, i}, \tag{2.3}$$

where $u = n_{\mathsf{A}} - \tau, \ldots, n_{\mathsf{A}} - 1$ and the second term in the summation is the piggyback.

Codes constructed in this way have erasure correcting capability at least $n_{\mathsf{A}} - k - \tau + 1$. We prove this in the following theorem.

Theorem 2.1. *An (n_{A},k) Class A code with τ piggybacks per row can correct a minimum of $n_{\mathsf{A}} - k - \tau + 1$ node failures.*

Proof. The proof is given in Appendix A. □

When a failure of a data node occurs, Class A parity nodes are used to recover $\tau+1$ of the k failed symbols. The Class A parity symbols are constructed in such a way that, when node j is erased, $\tau+1$ data symbols in this node can be recovered reading the (non-failed) $k-1$ data symbols in the jth row of the data array and $\tau+1$ parity symbols in the jth row of Class A nodes (see also Section 2.3.4). For later use, we define the set $\mathcal{R}_j$ as follows.

Definition 2.2. $\mathcal{R}_j$ is the set of $k-1$ data symbols that are read from row j to recover $\tau+1$ data symbols of node j using Class A parity nodes.

[2] We use the superscript A to indicate that the parity symbol is stored in a Class A parity node.

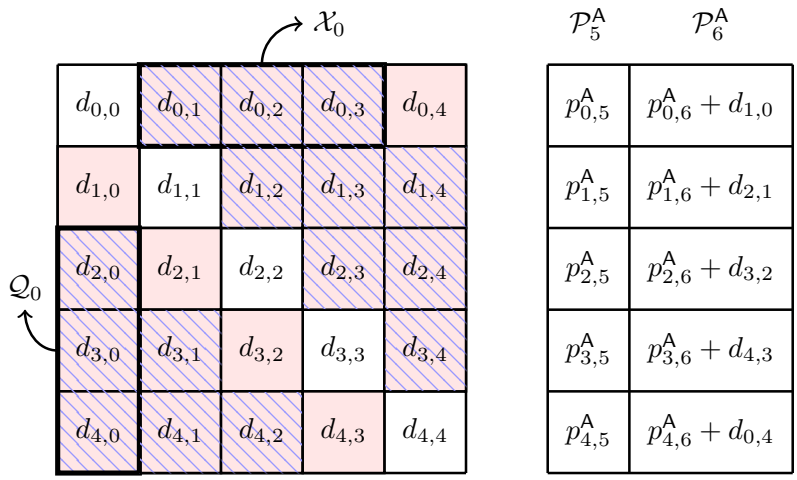


Figure 2.2: A $(7,5)$ Class A code with $\tau = 1$ constructed from a $(7,5)$ MDS code. $\mathcal{P}_5^{\mathsf{A}}$ and $\mathcal{P}_6^{\mathsf{A}}$ are the parity nodes. For each row i, colored symbols belong to $\mathcal{R}_i$.

Example 1. An example of Class A code is shown in Fig. 2.2. One can verify that the code can correct any 2 node failures. For each row i, the set $\mathcal{R}_i$ is indicated in red color. For instance, $\mathcal{R}_0 = \{d_{0,1},d_{0,2},d_{0,3},d_{0,4}\}$.

Although the main purpose of Class A parity nodes is to provide good erasure correcting capability, the use of piggybacks helps also in reducing the number of symbols that need to be read to recover the $\tau+1$ symbols. The remaining $k-\tau-1$ data symbols can also be recovered from Class A parity nodes, but at a high symbol read cost. Hence, the idea is to add another class of parity nodes, namely Class B parity nodes, in such a way that these symbols can be recovered with lower read cost.

2.3 Class B Parity Nodes

Class B parity nodes are obtained using an (n_{B},k) linear block code over $\mathbb{F}_{q^p}$ to encode the $k \times k$ data symbols of the data array, i.e., we use the (n_{B},k) code k times. This generates $(n_{\mathsf{B}} - k) \times k$ Class B parity symbols, $p_{i,u}^{\mathsf{B}}$, $i = 0,\ldots,k-1$, $u = n_{\mathsf{A}},\ldots,n-1$.

Definition 2.3. For $j = 0,\ldots,k-1$, define the set $\mathcal{Q}_j$ as

$$\mathcal{Q}_j = \{d_{(j+\tau+1)_k,j},d_{(j+\tau+2)_k,j},\cdots,d_{(j+k-1)_k,j}\}. \tag{2.4}$$

Assume that data node j fails. Is it easy to see that the set $\mathcal{Q}_j$ is the set of $k-\tau-1$ data symbols that are not recovered using Class A parity nodes.

Example 2. For the example in Fig. 2.2, the set $\mathcal{Q}_j$ is indicated by hatched symbols for each column j, $j = 0,\ldots,k-1$. For instance, $\mathcal{Q}_0 = \{d_{2,0},d_{3,0},d_{4,0}\}$.

For later use, we also define the following set.

Definition 2.4. For $j = 0, \ldots, k-1$, define the set $\mathcal{X}_j$ as

$$\mathcal{X}_j = \{d_{j,(j+1)_k}, d_{j,(j+2)_k}, \cdots, d_{j,(j+k-\tau-1)_k}\}. \tag{2.5}$$

Note that $\mathcal{X}_j = \mathcal{R}_j \cap \{\cup_l \mathcal{Q}_l\}$.

Example 3. For the example in Fig. 2.2, the set $\mathcal{X}_i$ is indicated by hatched symbols for each row i. For instance, $\mathcal{X}_0 = \mathcal{R}_0 \cap \{\mathcal{Q}_0 \cup \mathcal{Q}_1 \cup \mathcal{Q}_2 \cup \mathcal{Q}_3 \cup \mathcal{Q}_4\} = \{d_{0,1}, d_{0,2}, d_{0,3}\}$.

The purpose of Class B parity nodes is to allow recovering of the data symbols in $\mathcal{Q}_j$, $j = 0, \ldots, k-1$, at a low additional read cost. Note that after recovering $\tau + 1$ symbols using Class A parity nodes, the data symbols in $\mathcal{R}_j$ are already stored in the decoder memory, therefore they are accessible for the recovery of the remaining $k - \tau - 1$ data symbols using Class B parity nodes without the need of reading them again. In particular, the addition of a new Class B parity node allows to recover one new data symbol in $\mathcal{Q}_j$ (for all j) at the cost of one additional read. Furthermore, other data symbols in $\mathcal{Q}_j$ can be recovered at a lower additional read cost than using only Class A parity nodes. The main idea is based on the following proposition.

Proposition 1. *If a Class* B *parity symbol* p^{B} *is the sum of one data symbol* $d \in \mathcal{Q}_j$ *and a number of data symbols in* $\mathcal{X}_j$*, then the recovery of* d *comes at the cost of one additional read (one should read parity symbol* p^{B}*).*

This observation is used in the construction of Class B parity nodes (see Section 2.3.1 below) to reduce the normalized repair bandwidth, λ.

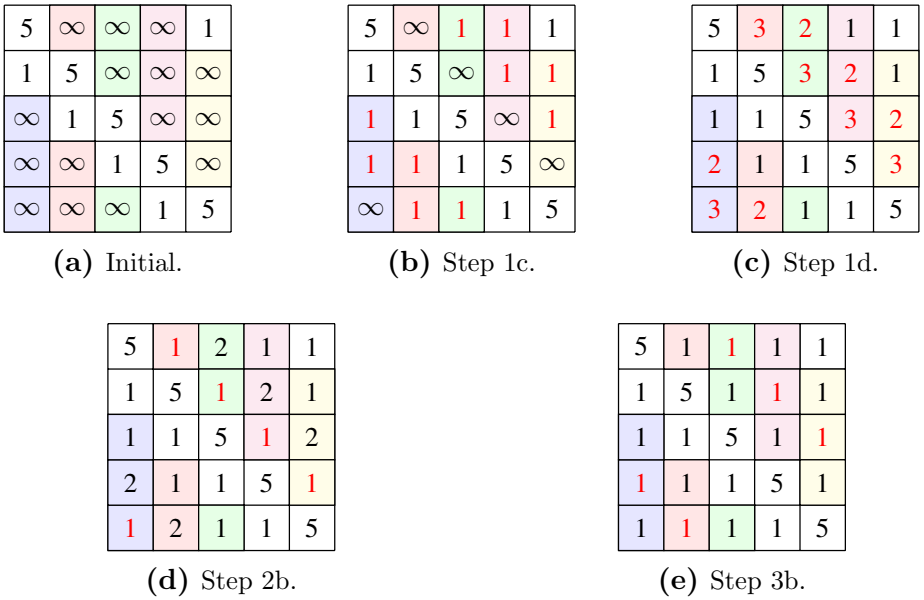
We remark that adding $k - \tau - 1$ Class B nodes would allow to reduce the additional cost read for all data symbols in $\mathcal{Q}_j$ (for all j) to 1. However, this comes at the expense of a reduction in the code rate. Thus, our code construction tradeoffs between repair bandwidth and code rate: the more Class B nodes are added, the lower the repair bandwidth is, but the lower the code rate is.

In the following, we propose a recursive algorithm for the construction of Class B parity nodes. In order to describe this algorithm, we define the function read(d,$\mathcal{P}$) as follows.

Definition 2.5. Consider a Class B parity node and let $\mathcal{P}^{\mathsf{B}}$ denote the set of parity symbols in this node. Also, let $d \in \mathcal{Q}_j$ for some j and $p^{\mathsf{B}} \in \mathcal{P}^{\mathsf{B}}$ be $p^{\mathsf{B}} = d + \sum_{d' \in \mathcal{D}'} d'$, where $\mathcal{D}' \subset \mathcal{D}$, i.e., the parity symbol p^{B} is the sum of d and a subset of other data symbols. Then,

$$\text{read}(d, p^{\mathsf{B}}) = |\breve{\mathcal{D}} \backslash \mathcal{X}_j|, \tag{2.6}$$

where $\breve{\mathcal{D}} = \{\mathcal{D}' \cup d\}$.



(a) Initial. **(b)** Step 1c. **(c)** Step 1d.

(d) Step 2b. **(e)** Step 3b.

Figure 2.3: Update of $\boldsymbol{A}$ during the construction of Class B parity nodes for the example in Section 2.3.1. The updates of $a_{i,j}$ after each step are highlighted in red color. The shaded symbols in column j denote the set $\mathcal{Q}_j$, while the shaded symbols in row i denote the set $\mathcal{X}_i$.

For a given data symbol d, the function read(d,p^{B}) gives the additional number of symbols that need to be read to recover d (considering the fact that some symbols are already cached in the memory).

To ease understanding, we introduce the algorithm through an example.

2.3.1 Construction Example

We construct a (10,5) code starting from the (7,5) Class A code in Fig. 2.2. In particular, we construct three Class B parity nodes, so that the additional number of reads to repair each of the remaining failed $k-\tau-1$ symbols (after recovering $\tau+1$ symbols using Class A parity nodes) is 1. With some abuse of notation, we denote these parity nodes by $\mathcal{P}_7^{\mathsf{B}}$, $\mathcal{P}_8^{\mathsf{B}}$, and $\mathcal{P}_9^{\mathsf{B}}$.

Denote by $\boldsymbol{A}$, $a_{i,j} = [\boldsymbol{A}]_{i,j}$, a temporary matrix of read values for the respective data symbols $d_{i,j}$. After Class A decoding,

$$a_{i,j} = \begin{cases} \infty & \text{if } d_{i,j} \in \{\cup_t \mathcal{Q}_t\} \\ k & \text{if } i = j \\ 1 & \text{otherwise,} \end{cases} \tag{2.7}$$

where $t = 0, \ldots, k-1$. For our example, $\boldsymbol{A}$ after Class A decoding is given in Fig. 2.3(a). Our algorithm operates on the $a_{i,j}$s whose initial value is ∞ and aims

to obtain the lowest possible values for these $a_{i,j}$s under the given number of Class B parity nodes. This is done in a recursive manner as follows.

1. Construct the first parity node, $\mathcal{P}_7^{\mathsf{B}}$.

1a. For each symbol $d_{i,j}$ define the set $\tilde{\mathcal{D}}_{i,j} \triangleq \{d_{(i+s)_k,(j+s)_k}\}_{s=0}^{k-1}$.

1b. Start with the elements in $\mathcal{Q}_0$. Pick an element $d_{i,0} \in \mathcal{Q}_0$ such that $a_{i,0} = \infty$, and $d_{0,i} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$. For instance, we take $d_{2,0}$.

1c. For $t = 0, \ldots, k-1$ compute

$$p_{t,7}^{\mathsf{B}} = d_{(i+t)_k,t} + d_{t,(i+t)_k} \tag{2.8}$$

and update the respective $a_{i,0}$ and $a_{0,i}$,

$$a_{(i+t)_k,t} = a_{t,(i+t)_k} = \text{read}(d_{(i+t)_k,t}, p_{t,7}^{\mathsf{B}}). \tag{2.9}$$

The resulting matrix $\boldsymbol{A}$ is shown in Fig. 2.3(b). There are still entries $a_{i,j} = \infty$ that need to be handled.

1d. For $t = 0, \ldots, k-1$ update

$$p_{t,7}^{\mathsf{B}} = p_{t,7}^{\mathsf{B}} + d_{t,(i'+t)_k}, \tag{2.10}$$

where $d_{0,i'} \in \mathcal{X}_0$ and $a_{0,i'} = \infty$ after step 1b. Update $\boldsymbol{A}$ accordingly (see Fig. 2.3(c)). Note that the read values $a_{(i+t)_k,(j+t)_k}$ have not worsened. This comes from the fact that the new added data symbol belongs to the corresponding set $\mathcal{X}$ and is already cached in the memory. Thus, the additional read cost is 0. On the other hand, the values $a_{(j+t)_k,(i+t)_k}$ increase.

2. Construct the second parity node, $\mathcal{P}_8^{\mathsf{B}}$.

2a. Pick an element $d_{i,0} \in \mathcal{Q}_0$ such that the corresponding $a_{i,j}$ is maximal. In our example, this is $d_{4,0}$ because $a_{4,0} = 3$.

2b. For $t = 0, \ldots, k-1$, do the following. Pick an element $d_{t,(u+t)_k} \in \mathcal{X}_t \backslash \tilde{\mathcal{D}}_{i,j}$ such that for all $d_{i',j'} \in \breve{\mathcal{D}}$, $\text{read}(d_{i',j'}, p_{t,8}) \leq a_{i',j'}$, where p^{B} is set to $p_{t,8}^{\mathsf{B}} = d_{(i+t)_k,t} + d_{t,(u+t)_k}$. For our example, we choose $d_{0,2}$. Notice the only other option $d_{0,3}$ is not a good choice as the new additional read cost would increase from 1 to 2. If such $d_{t,(u+t)_k}$ does not exist, set $p_{t,8}^{\mathsf{B}} = d_{(i+t)_k,t}$.

Update $\boldsymbol{A}$. The updated matrix is shown in Fig. 2.3(d).

3. Construct $\mathcal{P}_9^{\mathsf{B}}$.

$\mathcal{P}_7^{\mathsf{B}}$	$\mathcal{P}_8^{\mathsf{B}}$	$\mathcal{P}_9^{\mathsf{B}}$
$d_{2,0} + d_{0,2} + d_{0,1}$	$d_{4,0} + d_{0,2}$	$d_{3,0}$
$d_{3,1} + d_{1,3} + d_{1,2}$	$d_{0,1} + d_{1,3}$	$d_{4,1}$
$d_{4,2} + d_{2,4} + d_{2,3}$	$d_{1,2} + d_{2,4}$	$d_{0,2}$
$d_{0,3} + d_{3,0} + d_{3,4}$	$d_{2,3} + d_{3,0}$	$d_{1,3}$
$d_{1,4} + d_{4,1} + d_{4,0}$	$d_{3,4} + d_{4,1}$	$d_{2,4}$

Figure 2.4: Class B parity nodes for the data nodes in Fig. 2.2.

3a. Pick an element $d_{i,0} \in \mathcal{Q}_0$ such that the corresponding $a_{i,0}$ is maximal. In our example, this is $d_{3,0}$.

3b. For $t = 0, \ldots, k-1$, do the following. $p_{t,9}^{\mathsf{B}} = d_{(i+t)_k,t}$. Update $\boldsymbol{A}$. The resulting $\boldsymbol{A}$ has value k for all diagonal elements and 1 elsewhere (Fig. 2.3(e)).

The Class B parity nodes $\mathcal{P}_7^{\mathsf{B}}$, $\mathcal{P}_8^{\mathsf{B}}$, and $\mathcal{P}_9^{\mathsf{B}}$ are shown in Fig. 2.4.

A general version of the algorithm to construct Class B parity nodes is given in Appendix B.

2.3.2 Discussion of the Construction Example

The construction of Class B parity nodes starts by selecting an element $d_{i,j}$ of a given $\mathcal{Q}_j$ such that $a_{i,j} = \infty$ and $d_{j,i} \in \mathcal{X}_j \backslash \tilde{\mathcal{D}}_{i,j}$ (for simplicity, as in the example, we can start with $j = 0$). The first parity symbol of $\mathcal{P}_7$ after step 1c is therefore $p_{0,7} = d_{i,0} + d_{0,i}$, and the remaining parity symbols are obtained as in (2.8). By Proposition 1 the additional read cost of $d_{i,j}$ (after step 1c) is 1. The reason for selecting $d_{j,i} \in \mathcal{X}_j \backslash \tilde{\mathcal{D}}_{i,j}$ is due to the fact that, again by Proposition 1, its additional read cost is also 1. We remark that for each $d_{i,j} \in \mathcal{Q}_j$ it is not always possible to select $d_{j,i} \in \mathcal{X}_j \backslash \tilde{\mathcal{D}}_{i,j}$ and set $p_{j,7} = d_{i,j} + d_{j,i}$. This is the case when $k < 2(\tau + 1)$. If $d_{j,i} \in \mathcal{X}_j \backslash \tilde{\mathcal{D}}_{i,j}$ does not exist, then we select $d_{j,t} \in \mathcal{X}_j \backslash \tilde{\mathcal{D}}_{i,j}$ (see Appendix B). In this case, the additional read cost of $d_{j,t}$ (after step 1c) is > 1.

In general, step 1d has to be performed $|\mathcal{Q}_j| - 2$ times, corresponding to the number of entries $a_{i,j} = \infty$ per column of $\boldsymbol{A}$.

To reduce the additional read cost for the $k-\tau-1$ data symbols to 1, three Class B parity nodes need to be introduced. This reduces the code rate from $R = 5/7$ to

$R = 5/10 = 1/2$, i.e., it increases the storage overhead. If a lower storage overhead is required, Class B parity nodes can be *punctured*, starting from the last parity node (for the example, nodes $\mathcal{P}_9^{\mathsf{B}}$, $\mathcal{P}_8^{\mathsf{B}}$, and $\mathcal{P}_7^{\mathsf{B}}$ are punctured in this order), at the expense of an increasing repair bandwidth. At the limit, we would remain only with Class A parity nodes and the repair bandwidth corresponds to that of the Class A code. Thus, our code construction gives a family of rate-compatible codes and allows to tradeoff between repair bandwidth and storage overhead.

2.3.3 Matrix Approach

In the previous section, we have seen the construction of Class B parity nodes $\mathcal{P}_{n_{\mathsf{A}}}^{\mathsf{B}}, \ldots, \mathcal{P}_{n-1}^{\mathsf{B}}$ using the algorithm provided in Appendix B. In this section, we approach the construction of Class B nodes using matrices.

Let $\mathbf{d}$ be a row vector of size $1 \times k^2$ consisting of data blocks $d_{i,j}$, where $i,j = 0,1,\ldots,k-1$, row wise. This is to say that $\boldsymbol{d} = (d_{0,0}, d_{0,1}, \ldots, d_{0,k-1}, d_{1,0}, \ldots, d_{1,k-1}, \ldots, d_{k-1,k-1})$. We define column vector $\boldsymbol{p}^{\mathsf{B}} = (p_{0,n_{\mathsf{A}}}^{\mathsf{B}}, p_{1,n_{\mathsf{A}}}^{\mathsf{B}}, \ldots, p_{k-1,n_{\mathsf{A}}}^{\mathsf{B}}, p_{0,n_{\mathsf{A}}+1}^{\mathsf{B}}, p_{1,n_{\mathsf{A}}+1}^{\mathsf{B}}, \ldots, p_{k-1,n_{\mathsf{A}}+1}^{\mathsf{B}}, \ldots, p_{k-1,n-1}^{\mathsf{B}})$ of size $k(n_{\mathsf{B}} - k) \times 1$ that consists of a column of Class B parity symbols. Then these symbols can be constructed by a system of linear equations given as

$$\boldsymbol{G}\boldsymbol{d}^{\mathsf{T}} = \boldsymbol{p}^{\mathsf{B}} \tag{2.11}$$

where $\boldsymbol{G}$ is a $k(n_{\mathsf{B}} - k) \times k^2$ matrix of the form

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{G}_{n_{\mathsf{A}}}(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1}) \\ \boldsymbol{G}_{n_{\mathsf{A}}+1}(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1}) \\ \vdots \\ \boldsymbol{G}_{n-1}(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1}) \end{pmatrix} \tag{2.12}$$

Each submatrix $\mathbf{G}_l(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$, where $l = n_{\mathsf{A}}, n_{\mathsf{A}}+1, \ldots, n-1$, is of the order $k \times k^2$ is defined as

$$\boldsymbol{G}_l(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1}) = \begin{pmatrix} \mathbf{r}_0^0 & \mathbf{r}_1^0 & \cdots & \mathbf{r}_{k-1}^0 \\ \mathbf{r}_{k-1}^1 & \mathbf{r}_0^1 & \cdots & \mathbf{r}_{k-2}^1 \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{r}_1^{k-1} & \mathbf{r}_2^{k-1} & \cdots & \mathbf{r}_0^{k-1} \end{pmatrix} \tag{2.13}$$

Each $\mathbf{r}_m^s$, where $m = 0, \ldots, k-1$ is a vector of size $1 \times k$, where $\mathbf{r}_m = (r_{mk}, r_{mk+1}, \ldots, r_{mk+k-1})$ is $s-$shifted to the right cyclically. Fig. 2.5 illustrates the system of linear equations as shown in (2.11).

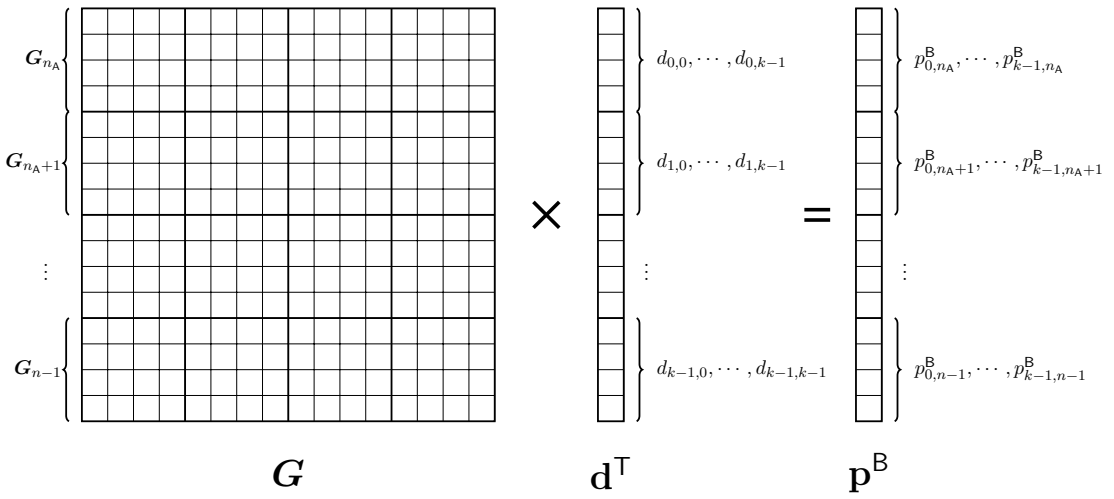


Figure 2.5: Illustration for the construction of Class B parity symbols using matrix approach

Let us revisit the following definitions that would be used in determining $\boldsymbol{G}$. Let $\boldsymbol{A}$ be a temporary matrix that holds read values defined in Definition 2.5. We initialize this matrix according to (2.7). Subsequently, for each symbol $d_{i,j}$ we redefine the set $\tilde{\mathcal{D}}_{i,j} \triangleq \{d_{(i+s)_k,(j+s)_k}\}_{s=0}^{k-1}$.

To determine $\boldsymbol{G}$, we recursively construct $\boldsymbol{G}_{n_\mathsf{A}}(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$, $\boldsymbol{G}_{n_\mathsf{A}+1}(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$, …, $\boldsymbol{G}_{n-1}(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$. The matrix $\boldsymbol{G}_l(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$ is completely determined by the vectors $\mathbf{r}_m$, where $m = 0, \ldots k-1$. These vectors are then determined by selecting a symbol $d_{i,0} \in \mathcal{Q}_0$ and $max\_itr = k - \tau - l + n_\mathsf{A} - 2$ symbols $d_{0,t} \in \mathcal{X}_0$. In the following, we provide the conditions on the choice of these symbols.

1. Choose a symbol $d_{i,0} \in \mathcal{Q}_0$: Select a symbol $d_{i,0} \in \mathcal{Q}_0$ such that the corresponding $a_{i,0}$ is maximum and there exists $d_{0,i} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$. If the latter does not exists, then select $d_{i,0}$ such that $a_{i,0}$ is maximum. Such a $d_{i,0}$ always exist.

2. Choose max_itr symbols $d_{0,t} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$: Construct a parity symbol $p^\mathsf{B} = d_{i,0}$. The choice of the symbols depends upon p^B. Choose $d_{0,i} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ such that $a_{0,i} > 1$ and $\text{read}(d_{0,i}, p^\mathsf{B} + d_{0,i}) \leq a_{0,i}$, then update $p^\mathsf{B} = p^\mathsf{B} + d_{0,i}$. If $d_{0,i}$ does not satisfy the conditions, choose $d_{0,t} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ such that $a_{0,t} > 1$ and $\text{read}(d_{0,t}, p^\mathsf{B} + d_{0,t}) \leq a_{0,t}$, then update $p^\mathsf{B} = p^\mathsf{B} + d_{0,t}$. For the remaining max_itr selections, recursively choose and update the following. Choose $d_{0,t} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ such that $a_{0,t} > 1$ and $\text{read}(d_{0,t}, p^\mathsf{B} + d_{0,t}) \leq a_{0,t}$, then update $p^\mathsf{B} = p^\mathsf{B} + d_{0,t}$.

It can be easily seen that the above conditions on the choice of symbols in $\mathcal{Q}_0$ and $\mathcal{X}_0$ is the same as that in Algorithm 1. In particular, the conditions for $d_{i,0} \in \mathcal{Q}_0$

coincides with lines 2 and 3 in the algorithm. Similarly, the conditions for $d_{0,t} \in \mathcal{X}_0$ coincide with line 20 in the same algorithm.

Depending upon the symbols $d_{i,j}$ selected, assign $r_{ik+j} = 1$. The remaining elements in each vector $\mathbf{r}_m$ is the zero element.

Once $\boldsymbol{G}_l(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$ is determined, $\boldsymbol{A}$ should be updated. Let us denote an element in row u and column v of $\boldsymbol{G}_l(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$ as $g^l_{u,v}$. From Fig. 2.5, it can be seen that an element $g^l_{u,v}$ denotes the coefficient of data symbol $d_{\lfloor v/k \rfloor, (v)_k}$ present in the parity check equation of $p^{\mathsf{B}}_{u,l}$. The non-zero elements in $\boldsymbol{G}_l(\mathbf{r}_0, \mathbf{r}_1, \ldots, \mathbf{r}_{k-1})$ signify the data symbols that are used in the construction of node $\mathcal{P}^{\mathsf{B}}_l$. Accordingly, their additional read costs need to be updated. Therefore, for all $u = 0, \ldots, k-1$ and $v = 0, \ldots, k^2 - 1$, update

$$a_{\lfloor v/k \rfloor, (v)_k} = \text{read}(g^l_{u,\,v}, p^{\mathsf{B}}_{u,l}) \text{ s.t. } g^l_{u,v} = 1 \tag{2.14}$$

Example 4. We construct a $(8, 4)$ code starting from a $(6, 4)$ Class A code having $\tau = 1$. Next, we construct a (6,4) Class B code. To do this, we have to determine $\boldsymbol{G}$. We initialize $a_{i,j}$ according to (2.7) as seen in Fig. 2.6(a). We set $max\_itr = 1$.

1. Construct $\boldsymbol{G}_6(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3)$.

1a. Select $d_{i,0} \in \mathcal{Q}_0$ such that $a_{i,0}$ is maximum. For instance, we choose $d_{2,0}$ as $a_{2,0} = \infty$. We then update $r_8 = 1$.

1b. Select $d_{0,t} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ such that its additional read cost is reduced. We choose $d_{0,1}$ as the additional read cost now reduces to 2. We then update $r_1 = 1$.

1c. We now construct $\boldsymbol{G}_6(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3)$ as follows

$$\mathbf{r}_0 = (0,1,0,0) \tag{2.15}$$
$$\mathbf{r}_1 = (0,0,0,0) \tag{2.16}$$
$$\mathbf{r}_2 = (1,0,0,0) \tag{2.17}$$
$$\mathbf{r}_3 = (0,0,0,0) \tag{2.18}$$

4	∞	∞	1
1	4	∞	∞
∞	1	4	∞
∞	∞	1	4

(a) Initial.

4	2	1	1
1	4	2	1
1	1	4	2
2	1	1	4

(b) Step 1c.

4	1	1	1
1	4	1	1
1	1	4	1
1	1	1	4

(c) Step 2d.

Figure 2.6: Update of $\boldsymbol{A}$ during the construction of $\boldsymbol{G}$ for the Example 4. The updates of $a_{i,j}$ after each step is highlighted in red color. The shaded symbols denote the set $\mathcal{Q}_j$, while the shaded symbols in rows i denote the set $\mathcal{X}_i$

and

$$\boldsymbol{G}_6(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3) = \begin{pmatrix} \mathbf{r}_0^0 & \mathbf{r}_1^0 & \mathbf{r}_2^0 & \mathbf{r}_3^0 \\ \mathbf{r}_3^1 & \mathbf{r}_0^1 & \mathbf{r}_1^1 & \mathbf{r}_2^1 \\ \mathbf{r}_2^2 & \mathbf{r}_3^2 & \mathbf{r}_0^2 & \mathbf{r}_1^2 \\ \mathbf{r}_1^3 & \mathbf{r}_2^3 & \mathbf{r}_3^3 & \mathbf{r}_0^3 \end{pmatrix}$$

$$= \left(\begin{array}{cccc|cccc|cccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) \tag{2.19}$$

1d. Update $\boldsymbol{A}$ according to (2.14). The resulting matrix is seen in Fig. 2.6(b).

2. Construct $\boldsymbol{G}_7(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3)$.

1a. Select $d_{i,0} \in \mathcal{Q}_0$ such that $a_{i,0}$ is maximum. For instance, we choose $d_{3,0}$ as $a_{3,0} = 2$. We then update $r_{12} = 1$.

1b. Since $max\_itr = 0$, we do not choose any element in $\mathcal{X}_0$.

1c. We now construct $\boldsymbol{G}_7(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3)$

$$\mathbf{r}_0 = (0,0,0,0) \tag{2.20}$$
$$\mathbf{r}_1 = (0,0,0,0) \tag{2.21}$$
$$\mathbf{r}_2 = (0,0,0,0) \tag{2.22}$$
$$\mathbf{r}_3 = (1,0,0,0) \tag{2.23}$$

and

$$\boldsymbol{G}_7(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3) = \begin{pmatrix} \mathbf{r}_0^0 & \mathbf{r}_1^0 & \mathbf{r}_2^0 & \mathbf{r}_3^0 \\ \mathbf{r}_3^1 & \mathbf{r}_0^1 & \mathbf{r}_1^1 & \mathbf{r}_2^1 \\ \mathbf{r}_2^2 & \mathbf{r}_3^2 & \mathbf{r}_0^2 & \mathbf{r}_1^2 \\ \mathbf{r}_1^3 & \mathbf{r}_2^3 & \mathbf{r}_3^3 & \mathbf{r}_0^3 \end{pmatrix}$$

$$= \left(\begin{array}{cccc|cccc|cccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right) \tag{2.24}$$

1d. Update $\boldsymbol{A}$ according to (2.14). The resulting matrix is seen in Fig. 2.6(c).

The matrix is then given as

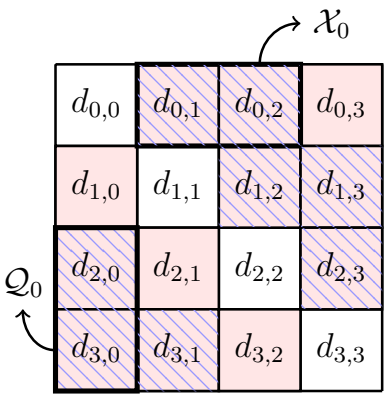
$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{G}_7(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3) \\ \boldsymbol{G}_8(\mathbf{r}_0,\mathbf{r}_1,\mathbf{r}_2,\mathbf{r}_3) \end{pmatrix}$$

$$= \left(\begin{array}{cccc|cccc|cccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right) \tag{2.25}$$

The resulting Class B parity symbols are shown in Fig. 2.7, which are computed from (2.11).

2.3.4 Repair of a Single Node Failure: Decoding Schedule

The repair of a failed systematic node, proceeds as follows. First, $\tau + 1$ symbols are repaired using Class A parity nodes. Then, the remaining symbols are repaired using Class B parity nodes. With a slight abuse of language, we will refer to the repair of symbols using Class A and Class B parity nodes as the decoding of Class A and Class B codes, respectively. Suppose that node j fails. Decoding is as follows.



$\mathcal{P}_4^{\mathsf{A}}$	$\mathcal{P}_5^{\mathsf{A}}$	$\mathcal{P}_6^{\mathsf{B}}$	$\mathcal{P}_7^{\mathsf{B}}$
$p_{0,4}^{\mathsf{A}}$	$p_{0,5}^{\mathsf{A}} + d_{1,0}$	$d_{2,0} + d_{0,1}$	$d_{3,0}$
$p_{1,4}^{\mathsf{A}}$	$p_{1,5}^{\mathsf{A}} + d_{2,1}$	$d_{3,1} + d_{1,2}$	$d_{0,1}$
$p_{2,4}^{\mathsf{A}}$	$p_{2,5}^{\mathsf{A}} + d_{3,2}$	$d_{0,2} + d_{2,3}$	$d_{1,2}$
$p_{3,4}^{\mathsf{A}}$	$p_{3,5}^{\mathsf{A}} + d_{0,3}$	$d_{1,3} + d_{3,0}$	$d_{2,3}$

Figure 2.7: A $(8, 4)$ code constructed from a $(6, 4)$ Class A with $\tau = 1$ and $(6, 4)$ Class B codes.

- Decoding of Class A code. k symbols ($k - 1$ data symbols in the jth row of the code array and $p_{j,k}^{\mathsf{A}}$) are read to reconstruct the failed data symbol in that row. These symbols are now cached in the memory. We then read the τ piggybacked symbols in the jth row. By construction (see (2.3)), this allows to repair τ failed symbols, at the cost of an additional read each.

- Decoding of Class B code. Each remaining failed data symbol $d_{i,j} \in \mathcal{Q}_j$ is obtained by reading a Class B parity symbol whose corresponding set $\breve{\mathcal{D}}$ (see Definition 2.5) contains $d_{i,j}$. In particular, if several Class B parity symbols $p_{i',j'}^{\mathsf{B}}$ contain $d_{i,j}$, we read the parity symbol with largest index j'. This yields the lowest additional read cost.

Chapter 3

Code Characteristics and Comparison

In this chapter, we characterize the different properties of the codes presented in Chapter 2. In particular, we present the fault tolerance, repair bandwidth, encoding and decoding complexities of the codes. We go on to show that one can improve the fault tolerance but at the cost of a lower repair bandwidth. Later, we compare these codes with the codes present in literature.

3.1 Fault Tolerance

The fault tolerance of the Class A code depends on the MDS code used in its construction and τ, as stated in Theorem 2.1. Hence, our proposed code has also fault tolerance $f \geq n_{\mathsf{A}} - k - \tau + 1$. Since $1 \leq \tau \leq n_{\mathsf{A}} - k - 1$, our codes have a fault tolerance of at least 2.

3.2 Normalized Repair Bandwidth

According to Section 2.3.4, to repair the first $\tau+1$ symbols in a failed node requires that $k-1$ data symbols plus $\tau+1$ Class A parity symbols are read. The remaining $k-\tau-1$ data symbols in the failed node are repaired by reading the Class B parity symbols. As seen in Section 2.3, the parity symbols in the first Class B parity node are constructed from sets of data symbols of cardinality $|\mathcal{Q}_j| = k-\tau-1$. Therefore, to repair each of the $k-\tau-1$ data symbols in this set requires to read at most $k-\tau-1$ symbols. The remaining Class B parity nodes are constructed from fewer symbols than $k-\tau-1$. An upper bound on the normalized repair bandwidth is therefore $\lambda \leq (k+\tau+(k-\tau-1)^2)/k$. In the closed interval $[1, k-2]$, it is observed that when τ increases, the fault tolerance reduces while the λ improves.

3.3 Repair Complexity of a Failed Node

To repair the first symbol requires k multiplications and $k-1$ additions. To repair the following τ symbols require an additional τk multiplications and additions. The final $k-\tau-1$ symbols require at most $k-\tau-2$ additions, since Class B parity symbols are constructed as the sum of at most $k-\tau-1$ data symbols. The repair complexity of one failed node is therefore

$$C_{\mathrm{R}} = O((k-1)\nu + k\nu^2) + O(\tau k(\nu+\nu^2)) + O((k-\tau-2)^2\nu). \tag{3.1}$$

The first two terms correspond to the Class A code while the last term corresponds to the Class B code.

3.4 Encoding Complexity

The encoding complexity of the (n,k) code is the sum of the encoding complexities of the two codes. The generation of each of the $n_{\mathsf{A}} - k$ Class A parity symbols in one row of the code array, $p_{i,j}^{\mathsf{A}}$ in (2.2), requires k multiplications and $k-1$ additions. Adding data symbols to τ of these parity symbols according to (2.3) requires an additional τ additions. The encoding complexity of the Class A code is therefore

$$C_{\mathsf{A}} = O((n_{\mathsf{A}} - k)(k\nu^2 + (k-1)\nu)) + O(\tau\nu). \tag{3.2}$$

According to Section 2.3, the first Class B parity symbol is constructed as the sum of $k-\tau-1$ data symbols, and each subsequent parity symbol uses less data symbols. Therefore, the encoding complexity of the Class B code is

$$C_{\mathsf{B}} = \sum_{i=1}^{n-n_{\mathsf{A}}} O((k-\tau-1-i)\nu). \tag{3.3}$$

Hence the total encoding complexity per row in our proposed code array is

$$C_{\mathrm{E}} = C_{\mathsf{A}} + C_{\mathsf{B}}. \tag{3.4}$$

3.5 Code Comparison

Table 3.1 provides a summary of the characteristics of different codes present in the literature as well as the codes constructed in this paper. Here, # Rows reported in column 3 refers to the number of rows in the code array and, without loss of

	Norm. Repair Band.	**# Rows**	**Fault Tolerance**	**Enc. Complexity**	**Norm. Repair Compl.**
MDS (n,k)	k	1	$n-k$	$O((n-k)((k-1)\nu+k\nu^2))$	$O((k-1)\nu+k\nu^2)$
LRC $[k,l,g]$ [12]	$\frac{k}{l}$	1	$g+1$	$gO((k-1)\nu+k\nu^2)+lO((\lceil\frac{k}{l}\rceil-1)\nu)$	$O((\lceil\frac{k}{l}\rceil-1)\nu)$
MDR $(k+2,k)$ [15]	$\frac{k+1}{2}$	2^k	2	$O((k-1)\nu)$	$O((k-1)\nu)$
Zigzag (n,k) [16]	$\frac{n-1}{n-k}$	$(n-k)^{k-1}$	$n-k$	$(n-k)O((k-1)\nu+k\nu^2)$	$O((k-1)\nu+k\nu^2)$
Piggyback $[n,k,r]$ [17]	$\frac{(k-t_r)(k+t)+t_r(k+t_r+r-2)}{2k}$	2	$n-k-r+1$	–	–
Proposed Codes $(n_\mathsf{A}+n_\mathsf{B}-k,k)$	$<\frac{k+\tau+(k-\tau-1)^2}{k}$	k	$\geq n_\mathsf{A}-k-\tau+1$	C_E	C_R/k

Table 3.1: Comparison of codes that aim at reducing repair bandwidth. The repair bandwidth and the repair complexity are normalized per symbol, while the encoding complexity is given per row in the code array.

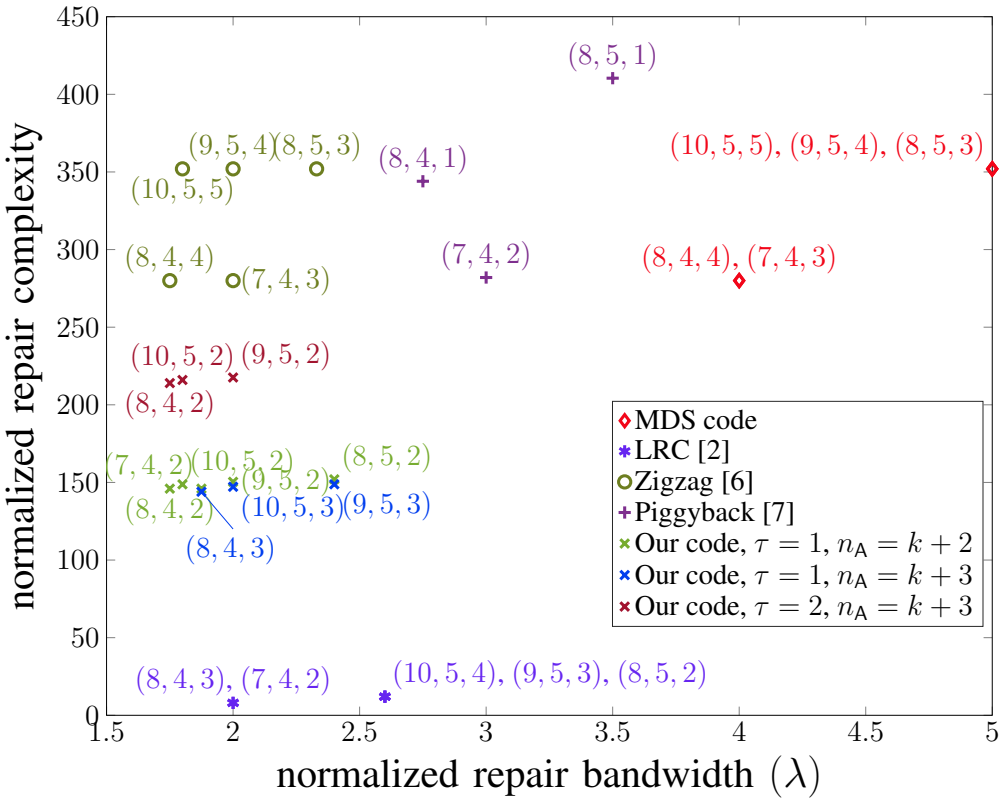


Figure 3.1: Comparisons of different codes (n,k,f) with $\nu = 8$.

generality, it can be assumed to be equal to β (the number of symbols per node) in (2.1). For our code, β grows linearly with k unlike MDR and Zigzag codes, where the growth is exponential. This implies that our codes require less memory to cache data symbols during repair. The fault tolerance f, the normalized repair bandwidth λ, the normalized repair complexity, and the encoding complexity, discussed in the previous subsections, are reported for all codes in Table 3.1, columns 4, 2, 6, and 5, respectively.

We have compared our codes with other codes in the literature. In Fig. 3.1, the normalized repair complexity of (n,k,f) codes over $\mathbb{F}_{2^8}$ ($\nu = 8$) is plotted versus its normalized repair bandwidth λ. In contrast to the bounds for the repair bandwidth and complexity reported in Table 3.1, Fig. 3.1 contains the exact number of integer additions.

The best codes for a DS system should be the ones that achieve the lowest repair bandwidth and have the lowest repair complexity. As seen in Fig. 3.1, MDS codes have both high repair complexity and repair bandwidth, but they are optimal in terms of fault tolerance for a given n and k. Zigzag codes achieve the same fault tolerance and high repair complexity as MDS codes, but at the lowest

repair bandwidth. At the other end, LRCs have the lowest repair complexity but a higher repair bandwidth and worse fault tolerance than Zigzag codes. Piggyback codes have a repair bandwidth between Zigzag and MDS codes, but with a higher repair complexity and worse fault tolerance. Our proposed codes have better repair complexity than Zigzag, MDS, and Piggyback codes, and also lower repair bandwidth compared to LRCs and Piggyback codes, at the same low repair bandwidth as Zigzag codes, but with the price of a lower fault tolerance.

Chapter 4
Conclusions

In this thesis, we constructed a new class of codes that achieve low repair bandwidth and low repair complexity for a single data node failure at the cost of fault tolerance. The codes are constructed from two smaller codes, Class A and B, where the former focuses on the reliability of the code, and latter focuses on reducing the repair bandwidth and complexity. Class A codes are constructed from an MDS code, which is then modified by using the piggybacking framework while Class B codes are low complexity linear block codes whose construction depends upon the Class A codes.

Our proposed codes achieve better repair complexity than Zigzag codes and Piggyback codes and better repair bandwidth than LRCs, but at the cost of slightly lower fault tolerance. A side effect of such a construction is that the number of symbols per node that needs to be encoded grows linearly with the code dimension. This implies that our codes are suitable for memory constrained DS systems as compared to Zigzag and MDR codes for which the number of symbols per node increases exponentially with the code dimension.

4.1 Future Work

While this thesis has presented a new family of repair efficient ECCs for DS systems, there are still opportunities to extend the work done here. Possible future research includes:

1. Modifying the construction of Class A and B codes in such a way that it provides optimal reduction in repair bandwidth.

2. Constructing codes that are able to reduce the repair bandwidth when there are multiple node failures.

3. Applying the idea behind the proposed codes to protograph based sparse graph codes since it is a known fact that these codes are low complexity. Intuitively, such codes would have much lower complexity than the proposed codes.

Appendix A

Proof of Theorem 2.1

Each row in the code array contains $n_{\mathsf{A}} - k - \tau$ parity symbols based on the MDS construction (i.e., parity symbols without piggybacks). Using these symbols, one can recover $n_{\mathsf{A}} - k - \tau$ data symbols in that row and, thus, $n_{\mathsf{A}} - k - \tau$ failures of systematic nodes. In order to prove the theorem, we need to show that by using piggybacked parity symbols $p_{i,u}$, $i = 0, \ldots, k-1$, in some parity node, u, it is possible to correct one arbitrary systematic failure. To do this, let us consider the system of linear equations $\boldsymbol{G}\boldsymbol{d}^{\mathsf{T}} = \boldsymbol{p}^{\mathsf{T}}$, representing the set of parity equations to compute $p_{i,u}$s, where $u = n_{\mathsf{A}} - \tau$.

In other words, $\boldsymbol{d} = (d_{0,0}, \ldots, d_{0,k-1}, d_{1,0}, \ldots, d_{k-1,k-1})$, $\boldsymbol{p} = (p_{0,u}, \ldots, p_{k-1,u})$, and $\boldsymbol{G}$ is given by

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{a} & \boldsymbol{u}_0 & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{a} & \boldsymbol{u}_1 & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{a} & \boldsymbol{u}_2 & \ldots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{u}_{k-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \boldsymbol{a} \end{pmatrix} \tag{A.1}$$

where $\boldsymbol{a} = (\alpha_{0,u}, \ldots, \alpha_{k-1,u})$, $\boldsymbol{u}_i$ is a vector of length k with one at position i and zeros elsewhere, and $\mathbf{0}$ is the all-zero vector of size k. Now, assume a systematic node r has failed. In order to repair it, we need to solve the following subsystem of linear equations $\boldsymbol{G}'\boldsymbol{w}^{\mathsf{T}} = \boldsymbol{p}^{\mathsf{T}}$, in which $\boldsymbol{w} = (d_{0,r}, \ldots, d_{k-1,r})$ and $\boldsymbol{G}'$ is a $k \times k$ submatrix of $\boldsymbol{G}$ such that:

1. Its diagonal elements are all $\alpha_{r,u}$.

2. It has 1 at row r and column $(r+1)_k$

3. All other entries are 0.

$$\boldsymbol{G}' = \begin{pmatrix} \alpha_{r,u} & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \alpha_{r,u} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & \alpha_{r,u} & 1 & \cdots & 0 \\ \vdots & \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & \alpha_{r,u} \end{pmatrix} \tag{A.2}$$

Note that $\boldsymbol{G}'$ is full rank. Therefore, one arbitrary data symbol can be corrected and, hence, the erasure correcting capabilities of the Class A code is $n_{\mathsf{A}} - k - \tau + 1$, which completes the proof.

Appendix B

Algorithm to Construct Class B Parity Nodes

We give an algorithm to construct $k-\tau-1$ Class B parity nodes in the order $\mathcal{P}^{\mathsf{B}}_{n_{\mathsf{A}}}, \mathcal{P}^{\mathsf{B}}_{n_{\mathsf{A}}+1}, \ldots, \mathcal{P}^{\mathsf{B}}_{n_{\mathsf{A}}+k-\tau-2}$. This results in the construction of $(k-\tau-1)k$ parity symbols $p^{\mathsf{B}}_{t,j}$. The algorithm is given in Algorithm 1. Consider the construction of the parity symbols of parity node $\mathcal{P}_{n_{\mathsf{A}}}$. The algorithm constructs first the parity symbol $p^{\mathsf{B}}_{0,n_{\mathsf{A}}}$ as the sum of an element $d_{i,0} \in \mathcal{Q}_0$ and max_itr elements in $\mathcal{X}_0$. Then, the other parity symbols $p^{\mathsf{B}}_{t,n_{\mathsf{A}}}$, $t > 0$, are constructed as the sum of an element $d_{(i+t)_k,t} \in \mathcal{Q}_t$ and max_itr elements in $\mathcal{X}_t$, i.e., following a specific pattern. The remaining parity nodes are constructed in a similar way, with the only difference that the number of elements added from the sets $\mathcal{X}_t$, max_itr, varies for each parity node. The construction of the parity symbols $p^{\mathsf{B}}_{t,j}$ depends on the choice of the symbols in the sets $\mathcal{Q}_t$ and $\mathcal{X}_t$. Assume that a parity symbol $p^{\mathsf{B}}_{0,j}$ is constructed. The data symbols involved in $p^{\mathsf{B}}_{0,j}$ are picked as follows.

- Choice of a data symbol in $\mathcal{Q}_0$: Select a symbol $d_{i,0} \in \mathcal{Q}_0$ such that the corresponding $a_{i,0}$ is maximum and there exists $d_{0,i} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ (lines 2 and 3 in the algorithm). If the latter does not exists, then select $d_{i,0}$ such that $a_{i,0}$ is maximum. Such a $d_{i,0}$ always exist.
- Choice of max_itr data symbols in $\mathcal{X}_0$: Select max_itr symbols $d_{0,i'} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ such that $a_{0,i'} > 1$ and its additional read cost does not increase (line 20 in the algorithm). If such a condition is not met, then the symbol $d_{0,i'}$ is not used in the construction of the parity symbol.

After the construction of each parity symbol, the corresponding entry of matrix $\boldsymbol{A}$ is updated.

Algorithm 1: Construction of Class B parity nodes

Initialization:

$\forall i,j = 0,\ldots,k-1$

$a_{i,j}$ as defined in (2.7)

$\tilde{\mathcal{D}}_{i,j} \triangleq \{d_{(i+s)_k,(j+s)_k}\}_{s=0}^{k-1}$

$max\_itr = k - \tau - 2$

for $\omega \leftarrow n_{\mathsf{A}}$ **to** $n_{\mathsf{A}} + k - \tau - 2$ **do**

// construct $k - \tau - 1$ nodes

  choose $d_{i,0} \in \mathcal{Q}_0$ s.t. $a_{i,0}$ is max && $d_{0,i} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$

  if $d_{0,i} \notin \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ **then** choose $d_{i,0} \in \mathcal{Q}_0$ s.t. $a_{i,0}$ is max

  $p^{\mathsf{B}}_{0,\omega} = d_{i,0}$

  for $t \leftarrow 1$ **to** $k-1$ **do**

    $p^{\mathsf{B}}_{t,\omega} = d_{(i+t)_k,t}$

  end

  for $itr \leftarrow 1$ **to** max_itr **do**

    $temp = p^{\mathsf{B}}_{0,\omega} + d_{0,i}$

    if $itr = 1$ && $d_{0,i} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ && $\mathrm{read}(d_{0,i},temp) < a_{0,i}$ **then**

      $i' \leftarrow i$

      $p^{\mathsf{B}}_{0,\omega} = temp$

      $a_{0,i'} = a_{i',0} = \mathrm{read}(d_{0,i'},p^{\mathsf{B}}_{0,\omega}) = 1$

      for $t \leftarrow 1$ **to** $k-1$ **do**

        $p^{\mathsf{B}}_{t,\omega} = p^{\mathsf{B}}_{t,\omega} + d_{t,(i'+t)_k}$

        $a_{t,(i'+t)_k} = \mathrm{read}(d_{t,(i'+t)_k},p^{\mathsf{B}}_{t,\omega})$

        $a_{(i+t)_k,t} = \mathrm{read}(d_{(i+t)_k,t},p^{\mathsf{B}}_{t,\omega})$

      end

    else

      if $\exists d_{0,i'} \in \mathcal{X}_0 \backslash \tilde{\mathcal{D}}_{i,0}$ && $\mathrm{read}(d_{0,i'},p^{\mathsf{B}}_{0,w}) \leq a_{0,i'}$ && $a_{0,i'} > 1$ **then**

        $p^{\mathsf{B}}_{0,\omega} = p^{\mathsf{B}}_{0,\omega} + d_{0,i'}$

        $a_{0,i'} = \mathrm{read}(d_{0,i'},p^{\mathsf{B}}_{0,\omega})$

        for $t \leftarrow 1$ **to** $k-1$ **do**

          $p^{\mathsf{B}}_{t,\omega} = p^{\mathsf{B}}_{t,\omega} + d_{t,(i'+t)_k}$

          $a_{t,(i'+t)_k} = \mathrm{read}(d_{t,(i'+t)_k},p^{\mathsf{B}}_{t,\omega})$

          $a_{(i+t)_k,t} = \mathrm{read}(d_{(i+t)_k,t},p^{\mathsf{B}}_{t,\omega})$

        end

      end

    end

  end

  $max\_itr \leftarrow max\_itr - 1$

end

Bibliography

[1] J. Gantz and D. Reinsel, "THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East," EMC Corporation, Tech. Rep., 2012.

[2] M. Sathiamoorthy, M. Asteris, D. S. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endow.*, vol. 6, no. 5, pp. 325–336, Mar. 2013.

[3] S. Abiteboul, I. Manolescu, P. Rigaux, M. Rousset, and P. Senellart, *Web Data Managment*. Cambridge University Press, 2011.

[4] S. Sankar, M. Shaw, K. Vaid, and S. Gurumurthi, "Datacenter scale evaluation of the impact of temperature on hard disk drive failures," *Trans. Storage*, vol. 9, no. 2, pp. 6:1–6:24, Jul. 2013.

[5] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003.

[6] J. J. Hanson, "An Introduction to the Hadoop Distributed File System," IBM, Tech. Rep., Feb. 2011.

[7] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '88. New York, NY, USA: ACM, 1988, pp. 109–116.

[8] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*. North-Holland Publishing Company, 1977.

[9] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie, "Scale-RS: An efficient scaling scheme for rs-coded storage clusters," *IEEE Trans. Parallel and Distributed Systems*, vol. PP, no. 99, p. 1, May 2014.

[10] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade

space for access efficiency in reliable data storage systems," in *Proc. IEEE Int. Symp. Network Computing and Applications*, Jul. 2007.

[11] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Trans. Inf. Theory*, vol. 58, no. 11, pp. 6925–6934, Nov. 2012.

[12] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. USENIX Annual Technical Conference*, Jun. 2012.

[13] D. Papailiopoulos and A. Dimakis, "Locally repairable codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Jul. 2012.

[14] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sept. 2010.

[15] Y. Wang, X. Yin, and X. Wang, "MDR codes: A new class of raid-6 codes with optimal rebuilding and encoding," *IEEE J. Sel. Areas in Commun.*, vol. 32, no. 5, pp. 1008–1018, May 2014.

[16] I. Tamo, Z. Wang, and J. Bruck, "Zigzag codes: MDS array codes with optimal rebuilding," *IEEE Trans. Inf. Theory*, vol. 59, no. 3, pp. 1597–1616, Mar. 2013.

[17] K. Rashmi, N. Shah, and K. Ramchandran, "A piggybacking design framework for read-and download-efficient distributed storage codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Jul. 2013.

[18] A. Datta and F. Oggier, "An overview of codes tailor-made for better repairability in networked distributed storage systems," *SIGACT News*, vol. 44, no. 1, pp. 89–105, Mar. 2013.

[19] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, 5th ed. CRC Press, Aug. 2001.