CHALMERS

Chalmers Publication Library

A Family of Erasure Correcting Codes with Low Repair Bandwidth and Low Repair Complexity

This document has been downloaded from Chalmers Publication Library (CPL). It is the author´s version of a work that was accepted for publication in:

IEEE Global Communications Conference (GLOBECOM)

Citation for the published paper:
Kumar, S. ; Graell i Amat, A. ; Andriyanova, I. et al. (2015) "A Family of Erasure Correcting Codes with Low Repair Bandwidth and Low Repair Complexity". IEEE Global Communications Conference (GLOBECOM)

Downloaded from: http://publications.lib.chalmers.se/publication/219654

Notice: Changes introduced as a result of publishing processes such as copy-editing and formatting may not be reflected in this document. For a definitive version of this work, please refer to the published source. Please note that access to the published version might require a subscription.

Chalmers Publication Library (CPL) offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all types of publications: articles, dissertations, licentiate theses, masters theses, conference papers, reports etc. Since 2006 it is the official tool for Chalmers official publication statistics. To ensure that Chalmers research results are disseminated as widely as possible, an Open Access Policy has been adopted.
The CPL service is administrated and maintained by Chalmers Library.

(article starts on next page)

A Family of Erasure Correcting Codes with Low Repair Bandwidth and Low Repair Complexity

Siddhartha Kumar†, Alexandre Graell i Amat†, Iryna Andriyanova‡, and Fredrik Brännström†
†Department of Signals and Systems, Chalmers University of Technology, Gothenburg, Sweden
‡ETIS Laboratory, ENSEA/University of Cergy-Pontoise/CNRS, Cergy-Pontoise, France

***Abstract*—We present the construction of a new family of erasure correcting codes for distributed storage that yield low repair bandwidth and low repair complexity. The construction is based on two classes of parity symbols. The primary goal of the first class of symbols is to provide good fault tolerance, while the second class facilitates node repair, reducing the repair bandwidth and the repair complexity. We compare the proposed codes with other codes proposed in the literature.**

I. Introduction

Distributed storage (DS) uses a network of interconnected inexpensive storage devices (referred to as storage nodes or simply nodes) to store data reliably over long periods of time. Reliability against node failures (commonly referred to as *fault tolerance*) is achieved by means of erasure correcting coding. Furthermore, when a node fails, a new node needs to be added to the DS network and populated with data to maintain the initial state of reliability. The problem of *repairing* a failed node is known as the *repair problem*.

Classical maximum distance separable (MDS) codes are optimal in terms of the fault tolerance/storage overhead tradeoff. However, the repair of a failed node requires the retrieval of large amounts of data from a large subset of nodes. Therefore, in the recent years, the design of erasure correcting codes that reduce the cost of repair has attracted a great deal of attention. Pyramid codes [1] were one of the first code constructions that addressed this problem. In particular, they aim at reducing the number of nodes that need to be contacted to repair a failed node, known as the repair access. Other codes that reduce the repair access are the local reconstruction codes (LRCs) [2], and the locally repairable codes [3], [4].

Other code constructions aim at reducing the repair bandwidth, defined as the amount of information that needs to be read from the DS network to repair a failed node. Among them, we can mention minimum disk I/O repairable (MDR) codes [5], Zigzag codes [6] and piggyback codes [7]. Piggybacking consists of adding carefully chosen linear combinations of data symbols (called piggybacks) to the parity symbols of a given erasure correcting code. This results in a lower repair bandwidth at the expense of a lower erasure correcting capability with respect to the original code.

In this paper, we propose a family of erasure correcting codes that achieve low repair bandwidth and low repair complexity. In particular, we propose a systematic code construction based on two classes of parity symbols. Correspondingly, there are two classes of parity nodes. The first class of parity nodes, whose primary goal is to provide erasure correcting capability, is constructed using an MDS code modified by applying specially designed piggybacks to some of its code symbols. The second class of parity nodes is constructed using a block code whose parity symbols are obtained with simple additions. This class of parity nodes does not have the purpose to bring any additional erasure correcting capability, but to facilitate node repair at low repair bandwidth and low repair complexity. In the paper, we compare the proposed codes with MDR codes, Zigzag codes, piggyback codes and LRCs [2], in terms of repair bandwidth and repair complexity.

This work was partially funded by the Swedish Research Council under grant #2011-5961. Siddhartha Kumar is now with the University of Bergen and the Simula Research Lab, Norway.

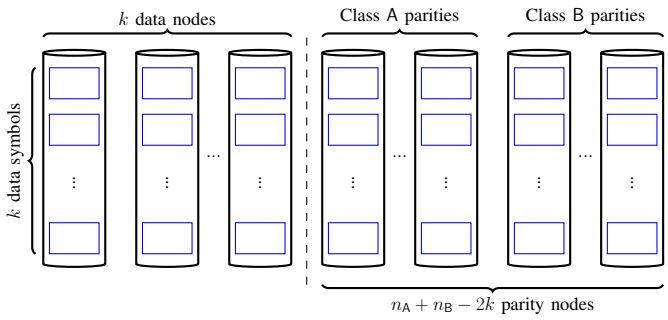


Fig. 1: System model.

Notation: We define the operator $(a+b)_k \triangleq (a+b) \mod k$. The Galois field of order q^p is denoted by $\mathbb{F}_{q^p}$.

II. Code Construction

We consider the distributed storage system depicted in Fig. 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, and 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. 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 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. 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. More precisely, let β be the number of symbols per node[2]. Then,

$$\lambda = \frac{\gamma}{\nu\beta}, \tag{1}$$

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

The main principle behind our 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 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).

III. 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 [7] to some of the parity symbols. The construction of Class A parity nodes is performed in two steps as follows.

[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.

[2]For our code construction, $\beta = k$, but this is not the case in general.

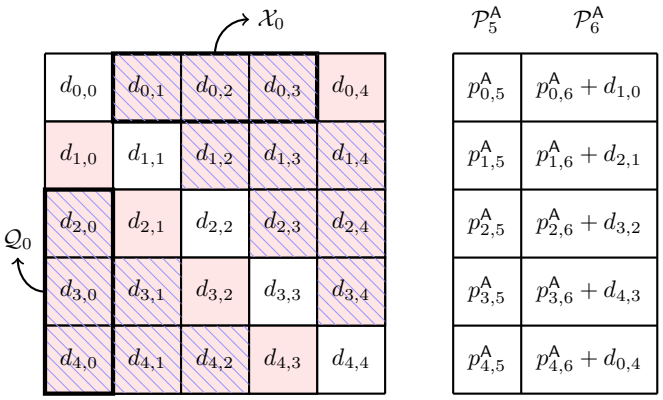


Fig. 2: A $(7, 5)$ Class A code with $\tau = 1$ constructed from a $(7, 5)$ MDS code. $\mathcal{P}_5^{\mathsf{A}}$ and $\mathcal{P}_6^{\mathsf{A}}$ are the parity nodes. For each row j, colored symbols belong to $\mathcal{R}_j$.

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[3]

$$p_{i,j}^{\mathsf{A}} = \sum_{l=0}^{k-1} \alpha_{l,j} d_{i,l}, \ j = k, \ldots, n_{\mathsf{A}} - 1, \tag{2}$$

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

$$p_{i,u}^{\mathsf{A}} = p_{i,u}^{\mathsf{A}} + d_{(i+u-n_{\mathsf{A}}+\tau+1)_k, i}, \tag{3}$$

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

The *fault tolerance* (i.e., the number of node failures that can be tolerated) of Class A codes is given in the following theorem.

Theorem 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 the appendix. ■

When a failure of a data node occurs, Class A parity nodes are used to repair $\tau + 1$ 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 repaired 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 IV-C). For later use, we define the set $\mathcal{R}_j$ as follows.

Definition 2: For $j = 0, \ldots, k-1$, define the set $\mathcal{R}_j$ as $\mathcal{R}_j = \{d_{j,(j+1)_k}, d_{j,(j+2)_k}, \cdots, d_{j,(j+k-1)_k}\}$.

The set $\mathcal{R}_j$ 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.

Example 1: An example of Class A code is shown in Fig. 2. One can verify that the code can correct any 2 node failures.

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

For each row j, the set $\mathcal{R}_j$ is indicated in red color. For instance, $\mathcal{R}_0 = \{d_{0,1}, d_{0,2}, d_{0,3}, d_{0,4}\}$.

The main purpose of Class A parity nodes is to provide good erasure correcting capability. However, the use of piggybacks helps also in reducing the number of symbols that need to be read to repair the $\tau+1$ symbols of a failed node that are repaired using Class A code, as compared to MDS codes. The remaining $k-\tau-1$ data symbols of the failed node 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.

IV. 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, $p^{\mathsf{B}}_{i,u}$, $i = 0, \ldots, k-1$, $u = n_{\mathsf{A}}, \ldots, n-1$.

Definition 3: For $j = 0, \ldots, k-1$, define the set $\mathcal{Q}_j$ as

$$\mathcal{Q}_j = \{d_{(j+\tau+1)_k, j}, d_{(j+\tau+2)_k, j}, \cdots, d_{(j+k-1)_k, j}\}. \quad (4)$$

Assume that data node j fails. It is 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, 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 4: For $j = 0, \ldots, k-1$, define the set $\mathcal{X}_j$ as

$$\mathcal{X}_j = \{d_{j,(j+1)_k}, d_{j,(j+2)_k}, \cdots, d_{j,(j+k-\tau-1)_k}\}. \quad (5)$$

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

Example 3: For the example in Fig. 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. 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 IV-A below) to reduce the normalized repair bandwidth, λ. In particular, we add $k-\tau-1$ Class B parity nodes which allow to reduce the additional read cost of all $k(k-\tau-1)$ data symbols in all $\mathcal{Q}_j$'s to 1. (The addition of a single Class B parity node allows to recover one new data symbol in each $\mathcal{Q}_j$, $j = 0, \ldots, k-1$, at the cost of one additional read).

In order to describe the code construction, we define the function $\mathrm{read}(d, p^{\mathsf{B}})$ as follows.

Definition 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,

$$\mathrm{read}(d, p^{\mathsf{B}}) = |\breve{\mathcal{D}} \backslash \mathcal{X}_j|, \quad (6)$$

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

For a given data symbol d, the function $\mathrm{read}(d, p^{\mathsf{B}})$ 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).

A. Construction Example

In the following, we propose a recursive algorithm for the construction of Class B parity nodes. To ease understanding, we introduce the algorithm through an example.

We construct a $(10, 5)$ code starting from the $(7, 5)$ Class A code in Fig. 2. In particular, we construct $k-\tau-1 = 3$ Class B parity nodes, so that the additional number of reads to repair each of the remaining failed $k-\tau-1 = 3$ symbols (after recovering $\tau+1 = 2$ symbols using Class A parity nodes) is 1. With some abuse of notation, we denote these parity nodes by $\mathcal{P}^{\mathsf{B}}_7$, $\mathcal{P}^{\mathsf{B}}_8$, and $\mathcal{P}^{\mathsf{B}}_9$.

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,

$$a_{i,j} = \begin{cases} \infty & \text{if } d_{i,j} \in \{\cup_t \mathcal{Q}_t\} \\ k & \text{if } i = j \\ 1 & \text{otherwise,} \end{cases} \quad (7)$$

where $t = 0, \ldots, k-1$. For our example, $\boldsymbol{A}$ after Class A decoding is given in Fig. 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, $\mathcal{P}^{\mathsf{B}}_7$.

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

$$p^{\mathsf{B}}_{t,7} = d_{(i+t)_k, t} + d_{t,(i+t)_k} \quad (8)$$

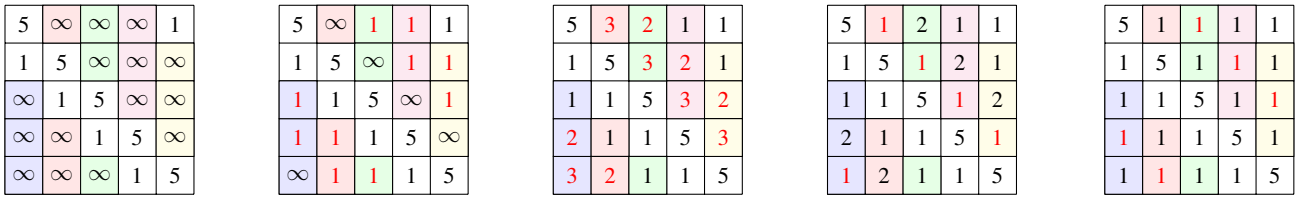
and update the respective $a_{i,0}$ and $a_{0,i}$,

$$a_{(i+t)_k, t} = a_{t,(i+t)_k} = \mathrm{read}(d_{(i+t)_k, t}, p^{\mathsf{B}}_{t,7}). \quad (9)$$

The resulting matrix $\boldsymbol{A}$ is shown in Fig. 3(b). There are still entries $a_{i,j} = \infty$ that need to be handled.

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

$$p^{\mathsf{B}}_{t,7} = p^{\mathsf{B}}_{t,7} + d_{t,(i'+t)_k}, \quad (10)$$



(a) Initial. (b) Step 1c. (c) Step 1d. (d) Step 2b. (e) Step 3b.

Fig. 3: Update of $\boldsymbol{A}$ during the construction of Class B parity nodes for the example in Section IV-A. 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 $\mathcal{X}_i$.

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

Fig. 4: Class B parity nodes for the data nodes in Fig. 2.

where $d_{0,i'} \in \mathcal{X}_0$ and $a_{0,i'} = \infty$ after step 1b. Update $\boldsymbol{A}$ accordingly (see Fig. 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 $d_{0,2}$. Note that 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. 3(d).

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

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. 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. 4.

A general version of the algorithm to construct Class B parity nodes can be found in the extended version of this paper [8].

B. 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 (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 $d_{j,t} \in \mathcal{X}_j \backslash \tilde{\mathcal{D}}_{i,j}$ (details are given in [8]). 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 $\boldsymbol{A}$.

Adding $k - \tau - 1$ Class B nodes allows to reduce the additional read cost for all data symbols in all $\mathcal{Q}_j$ to 1 (see Fig. 3(e)). However, this comes at the expense of a reduction in the code rate, i.e., the storage overhead is increased. In the example, $k - \tau - 1 = 3$ Class B parity nodes need to be introduced, which reduces the code rate from $R = 5/7$ to $R = 5/10 = 1/2$. 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 increased repair bandwidth. If all Class B parity nodes are punctured, 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 which trades off between repair bandwidth and storage overhead: adding more Class B nodes reduces the repair bandwidth but increases the storage overhead.

C. 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.

- **Decoding of Class A code**. To reconstruct the failed data symbol in the jth row of the code array, k symbols ($k-1$

data symbols and $p^{\mathsf{A}}_{j,k}$) in the jth row are read. These symbols are now cached in the memory. We then read the τ piggybacked symbols in the jth row. By construction (see (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 5) contains $d_{i,j}$. In particular, if several Class B parity symbols $p^{\mathsf{B}}_{i',j'}$ contain $d_{i,j}$, we read the parity symbol with largest index j'. This yields the lowest additional read cost.

V. Code Characteristics and Comparison

In this section we characterize some different properties of the codes presented in Sections III and IV.

A. 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 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.

B. Normalized Repair Bandwidth

According to Section IV-C, 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 IV, 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 $\lambda < (k+\tau+(k-\tau-1)^2)/k$. It is observed that when τ increases, the fault tolerance reduces while λ improves.

C. Repair Complexity of a Failed Node

We first consider the complexity of elementary arithmetic operations of elements of size $\nu = \lceil \log_2 q^p \rceil$ in $\mathbb{F}_{q^p}$. An addition requires $O(\nu)$ and multiplication requires $O(\nu^2)$. The term inside $O(\cdot)$ denotes the number of elementary binary additions. 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

$$C_{\mathrm{R}} = O((k-1)\nu + k\nu^2) + O(\tau k(\nu+\nu^2)) + O((k-\tau-2)^2\nu). \tag{11}$$

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

D. Encoding Complexity

The encoding complexity of the (n,k) code, C_{E}, 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, $p^{\mathsf{A}}_{i,j}$ in (2), requires k multiplications and $k-1$ additions. Adding data symbols to τ of these parity symbols according to (3) requires an additional τ additions. The encoding complexity of the Class A code is therefore

$$C_{\mathsf{A}} = O((n_{\mathsf{A}} - k)(k\nu^2 + (k-1)\nu)) + O(\tau\nu). \tag{12}$$

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

$$C_{\mathsf{B}} = \sum_{i=1}^{n-n_{\mathsf{A}}} O((k-\tau-1-i)\nu). \tag{13}$$

Finally, $C_{\mathrm{E}} = C_{\mathsf{A}} + C_{\mathsf{B}}$.

E. Code Comparison

Table I provides a summary of the characteristics of different codes proposed in the literature as well as the codes constructed in this paper.[4] In the table, column 2 reports the value of β (see (1)) for each code construction. For our code, $\beta = k$, unlike for MDR and Zigzag codes, for which β grows exponentially with k. 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 in columns 3, 4, 5, and 6, respectively.

In Fig. 5, we compare our codes with other codes in the literature. In particular, the figure plots the normalized repair complexity of (n,k,f) codes over $\mathbb{F}_{2^8}$ ($\nu = 8$) versus their normalized repair bandwidth λ. In contrast to the bounds for the repair bandwidth and complexity reported in Table I, Fig. 5 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. 5, 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 yield the lowest repair complexity but a higher repair bandwidth and worse fault tolerance than Zigzag codes. Piggyback codes have a repair bandwidth between that of Zigzag and MDS codes, but with a higher repair complexity and worse fault tolerance. For a given storage overhead, our proposed codes have better repair bandwidth than MDS codes, Piggyback codes and LRCs, and equal or similar repair bandwidth than Zigzag codes. Furthermore, they

[4]The variables t, t_r and r in Table I are defined in [7] and [2] respectively. The definition of ℓ comes directly from r that is defined in [7].

TABLE I: Comparison of (n, k) 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. Note that for MDR codes $n = k + 2$.

	β	**Fault Tolerance**	**Norm. Repair Band.**	**Norm. Repair Compl.**	**Enc. Complexity**
MDS	1	$n-k$	k	$O((k-1)\nu + k\nu^2)$	$O((n-k)((k-1)\nu) + k\nu^2)$
LRC [2]	1	$r+1$	$\frac{k}{n-k-r}$	$O((\lceil \frac{k}{n-k-r} \rceil - 1)\nu)$	$rO((k-1)\nu + k\nu^2) + (n-k-r)O((\lceil \frac{k}{n-k-r} \rceil - 1)\nu)$
MDR [5]	2^k	2	$\frac{k+1}{2}$	$O((k-1)\nu)$	$O((k-1)\nu)$
Zigzag [6]	$(n-k)^{k-1}$	$n-k$	$\frac{n-1}{n-k}$	$O((k-1)\nu + k\nu^2)$	$O((n-k)((k-1)\nu) + k\nu^2)$
Piggyback [7]	2	1	$\frac{(k-t_r)(k+t)+t_r(k+t_r+\ell-2)}{2k}$	–	–
Proposed Codes	k	$\geq n-n_{\mathsf{B}}-\tau+1$	$< \frac{k+\tau+(k-\tau-1)^2}{k}$	C_{R}/k	C_{E}

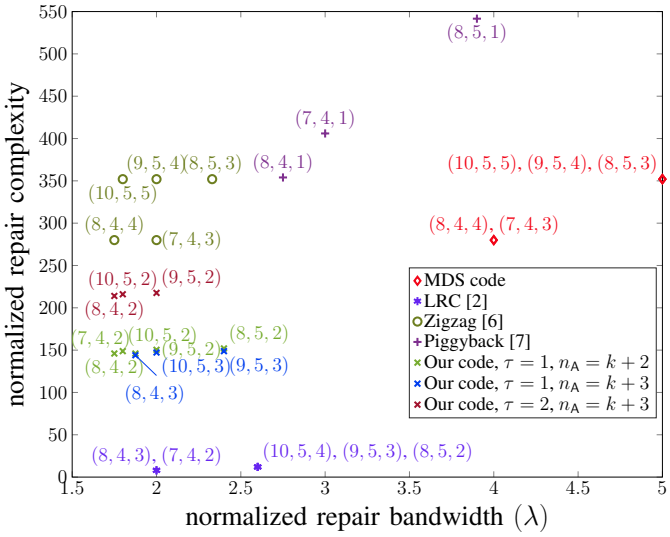


Fig. 5: Comparisons of different codes (n, k, f) with $\nu = 8$.

yield lower repair complexity as compared to MDS, Piggyback and Zigzag codes. However, the benefits in terms of repair bandwidth and/or repair complexity with respect to MDS and Zigzag codes come at a price of a lower fault tolerance.

VI. Conclusion

In this paper, we constructed a new class of codes that achieve low repair bandwidth and low repair complexity for a single node failure. The codes are constructed from two smaller codes, Class A and B, where the former focuses on the fault tolerance of the code, and the latter focuses on reducing the repair bandwidth and complexity. 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.

Appendix A
Proof of Theorem 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 node 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

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{a} & \boldsymbol{u}_0 & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{a} & \boldsymbol{u}_1 & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \boldsymbol{a} & \boldsymbol{u}_2 & \ldots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{u}_{k-1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \boldsymbol{a} \end{pmatrix} \tag{14}$$

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: a) its diagonal elements are all $\alpha_{r,u}$; b) it has 1 at row r and column $(r+1)_k$; c) all other entries are 0. Note that $\boldsymbol{G}'$ is full rank. Therefore, one arbitrary data symbol can be corrected and, hence, the erasure correcting capability of Class A code is at least $n_{\mathsf{A}} - k - \tau + 1$, which completes the proof.

References

[1] 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.

[2] 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.

[3] M. Sathiamoorthy *et al.*, "XORing elephants: Novel erasure codes for big data," *Proc. Very Large Data Bases Endowment*, vol. 6, no. 5, Mar. 2013.

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

[5] 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.

[6] 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.

[7] 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.

[8] S. Kumar, A. Graell i Amat, I. Andriyanova, and F. Brännström, "A family of erasure correcting codes with low repair bandwidth and low repair complexity," 2015. [Online]. Available: arXiv:1505.03491