Decentralized Erasure Coding for Efficient Data Archival in Distributed Storage Systems

Lluis Pamies-Juarez[1], Frederique Oggier[1], and Anwitaman Datta[2]

[1] School of Mathematical and Physical Sciences
[2] School of Computer Engineering
Nanyang Technological University, Singapore
{lpjuarez,frederique,anwitaman}@ntu.edu.sg

Abstract. Distributed storage systems usually achieve fault tolerance by replicating data across different nodes. However, redundancy schemes based on *erasure codes* can provide a storage-efficient alternative to replication. This is particularly suited for *data archival* since archived data is rarely accessed. Typically, the migration to erasure-encoded storage does not leverage on the existing replication based redundancy, and simply discards (garbage collects) the excessive replicas. In this paper we propose a new *decentralized erasure coding process* that achieves the migration in a *network-efficient* manner in contrast to the traditional coding processes. The proposed approach exploits the presence of data that is already replicated across the system and distributes the redundancy generation among those nodes that store part of this replicated data, which in turn reduces the overall amount of data transferred during the encoding process. By storing additional replicated blocks at nodes executing the distributed encoding tasks, the necessary network traffic for archiving can be further reduced. We analyze the problem using symbolic computation and show that the proposed decentralized encoding process can reduce the traffic by up to 56% for typical system configurations.

Keywords: archival, migration, erasure codes, distributed storage.

1 Introduction

Large data centers such as Google file-system (GFS) [9], Amazon S3 [2] or Hadoop file-system (HDFS) [3] handle extremely big volume of data by scaling-out, i.e., by realizing a distributed storage system comprising of hundreds or even thousands of commodity storage servers. To ensure that the stored data survives failures of some of the storage nodes, all data needs to be redundantly stored. A common and simple form of redundancy is to store multiple copies (replicas) of each data across the system. Storing erasure coded data is a more sophisticated alternative, which achieves significantly better trade-off in terms of storage-overhead and fault-tolerance [14, 17]. Many recent systems such as Microsoft Azure [10], Facebook's HDFS-RAID [4, 16] and the new version of the Google File System [8] among others have thus embraced erasure codes based storage systems. Typical parameter choices of erasure codes used in these deployed systems incur overall overhead of 1.3×–2× the size of the original data [4, 7, 10].

This translates to a reduction of up to 50% in infrastructural costs with respect to a (3-way) replication based storage system.

Even though erasure coding based systems have significantly lower storage overhead, newly inserted data is usually first replicated across different storage nodes. The reasons for initially using replication are twofold. Replication is easy and fast to achieve by pipelining the data through the involved storage nodes,[1] without incurring any computation related costs or latency. This allows high throughput data insertion, achieving immediate fault tolerance. Furthermore, replication allows task schedulers to exploit data locality and achieve load-balancing [9], helping improve performance of applications manipulating the stored data.

Thus the use of erasure code based redundancy is primarily relegated to archival of data which is no longer frequently accessed [7]. Such a pragmatic design choice reduces the storage footprint significantly, and has immediate impact on the infrastructural and operational costs of a data center. However, this dual redundancy based approach, namely, the use of replication for newly inserted data and that of erasure codes for subsequent archival suffers from an important problem: all archived data passes through two independent redundancy generation processes, each of them having their own associated costs.

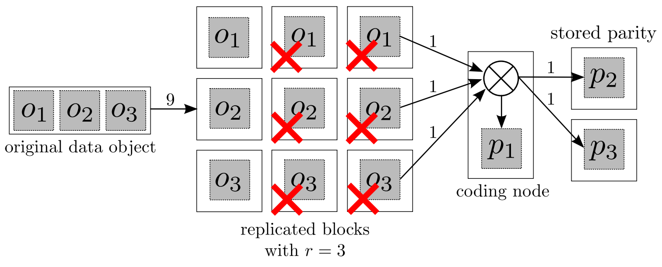
In this work, we explore a new decentralized erasure coding process leveraging on the existence of replicas from the first (data insertion) phase to reduce the network overheads during the second (migration from replication to erasure coded storage) phase. The basic idea is illustrated with a toy example shown in Figure 1. However, in order to amplify the reduction of the network traffic w.r.to the traditional archiving process, the replicas created during the insertion of new data have to be placed in some specific manner. Specifically, if multiple distinct blocks of a replicated object are collocated in a specific manner within the subset of nodes performing the decentralized encoding, then such locality can be exploited to further reduce the network traffic. Note that collocation of random object blocks may not be amenable to such benefits.

Symbolic computation based analysis show that for typical erasure code configurations used in in-production storage systems, the proposed decentralized erasure coding process can reduce the network traffic by up to 24% or 56%, depending on the collocation of the replicated blocks.

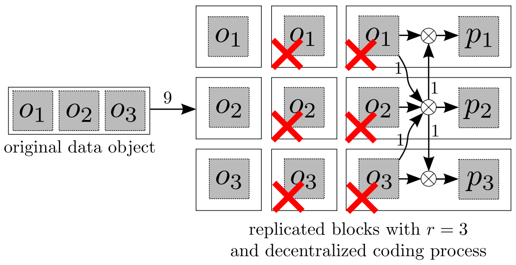
The main contributions of this paper are threefold.

1. We introduce a new *decentralized erasure coding process* to reduce the traffic required to archive data in replicated storage systems.
2. We provide a *generic code construction* that exploits this decentralized coding process.
3. We show how the traffic required during the decentralized coding process can by further reduced by adopting smart *replica placement strategies* during the data insertion phase.

[1] New inserted data can be stored in a first node while it is concurrently forwarded to and stored in a second node, and from this second node to a third, and so on [3,9].



(a) Traditional archiving process.



original data object
replicated blocks with $r = 3$ and decentralized coding process
9
1

(b) Decentralized coding process.

Fig. 1. Example of how to generate erasure code redundancy from a replicated system using (a) a traditional encoding process, and (b) a decentralized coding process. White squares represent storage nodes and arrow labels denote the number of blocks transferred over the network. We can see how (a) requires a total of 5 network transfers while (b) only needs 4 network transfers. The "X" symbol denotes the replicas that are discarded once the archival finishes, and the symbol $\otimes$ denotes an encoding operation.

The rest of the paper is organized as follows. In Section 2 we provide some background on erasure codes for distributed storage systems. In Section 3 we present our new decentralized erasure coding process, and in Section 4 we evaluate its fault tolerance and its encoding traffic savings. In Section 5 we discuss some related works. Finally, in Section 6 we draw our conclusions and outline some planned future work.

2 Background

When a data object is stored for the first time in a distributed storage system such as GFS [9] or HDFS [3], it is initially split into blocks of size B and each of them is replicated over r different storage nodes (usually $r = 3$). The size of these blocks is set to relatively large values of B, e.g., $B = 64$MB in GFS and HDFS, which allow to amortize data access latencies, as well as to exploit local data caching [9].

At a later time, when an object does not need to be frequently accessed, it can be archived using an erasure code, reducing its storage footprint, and hence its associated storage costs. This encoding process takes k blocks of data, each of size B, and computes m parity blocks (or redundancy blocks) of the same size, which are stored in m other different storage nodes. Since in most cases it is unlikely that data objects were split exactly into k blocks during the insertion process, the k blocks used in the encoding process might belong to different data objects. For example, in some systems files from the same directory are jointly encoded [5].

We can formally define the erasure encoding process as follows. Let the vector $\mathbf{o} = (o_1, \dots, o_k)$ denote a data object of $k \times q$ bits. That is, each symbol o_i, $i = 1, \dots, k$ is a string of q bits. Operations are typically performed using finite field arithmetic, that is, the two bits $\{0, 1\}$ are seen as forming the finite field $\mathbb{F}_2$ of two elements, while o_i, $i = 1, \dots, k$ then belong to the binary extension field $\mathbb{F}_{2^q}$ containing 2^q elements. Then, the encoding of the object $\mathbf{o}$ is performed using an $(n \times k)$ generator matrix G such that $G \cdot \mathbf{o}^T = \mathbf{c}^T$, in order to obtain an n-dimensional codeword $\mathbf{c} = (c_1, \dots, c_n)$ of size $n \times q$ bits. When the generator matrix G has the form $G = [I_k, G']^T$ where I_k is the identity matrix and G' is a $k \times m$ matrix, the codeword $\mathbf{c}$ becomes $\mathbf{c} = [\mathbf{o}, \mathbf{p}]$ where $\mathbf{o}$ is the original object, and $\mathbf{p}$ is a parity vector containing $m \times q$ parity bits. The code is then said to be systematic, in which case the k parts of the original object remain unaltered after the coding process. The data can then still be read without requiring a decoding process by accessing these systematic pieces.

Finally, an optimal erasure code in terms of the trade-off between storage overhead and fault tolerance is called a maximum distance separable (MDS) code, and has the property that the original object can be reconstructed from any k out of the total $n = k+m$ stored blocks, tolerating the loss of any arbitrary $m = n-k$ blocks. The notation "(n, k) code" is often used to emphasize the code parameters. Examples of the most widely used MDS codes are the Reed-Solomon codes [13].

While the efficacy of the use of erasure codes for fault tolerant storage has long been understood and leveraged, the migration process from a replication based storage to erasure coding based storage has been identified as a significant challenge relatively recently [5], given the tremendous growth in the volume of data that is continuously being generated and needs to be processed and archived. Next, we explain how the network traffic can be significantly reduced during the migration by embracing a decentralized coding process leveraging on the replicas.

3 Decentralizing the Data Archival Process

In this section we first introduce the decentralized erasure coding problem statement in 3.1. In 3.2 we provide a motivating example of how the decentralized archiving process works, and in 3.3 we provide a general code construction using a decentralized erasure coding process.

3.1 Problem Statement

When a data object $\mathbf{o} = (o_1, \dots, o_k)$ is newly inserted in the system, each data block o_i is replicated across r different storage nodes. Once the object $\mathbf{o}$ is no longer in frequent use, it is archived using a systematic (n, k) erasure code. The migration from replicas to encoding goes as follows: a node obtains an entire copy of $\mathbf{o}$ by downloading k different blocks, encodes them to generate a parity vector $\mathbf{p} = (p_1, \ldots, p_m)$, and finally uploads each parity block p_i to a different storage node. The encoding process thus requires the transfer of $n = k + m$ blocks, which can be reduced to $n - 1$ if the coding node keeps one of the parity blocks and uploads $m - 1$ blocks. Once the m parity blocks are stored, the number of replicas of each stored block o_i can be safely reduced to $r = 1$, discarding the remaining $r - 1$ block replicas. After this, a whole replica of the original object $\mathbf{o}$ remains unaltered and stored over k different storage nodes, becoming then the systematic part of the final codeword, which is formed of the blocks $c_1 = o_1, c_2 = o_2, \ldots, c_k = o_k$. An example of this process is depicted in Figure 1a for a simple (6,3) code.

The problem with this traditional coding process is that redundant data is transferred within the network twice (once to obtain and store kr replicated blocks, and once to compute and upload the m parity blocks) which might seem a waste of resources. Indeed, out of the kr replicated blocks generated during the first storage phase, only k of them are used to compute the coded parity blocks. This observation motivates the design of a new decentralized erasure coding process that reuses the original replicated data to reduce the total number of blocks transferred during the data archival process. In Figure 1b we showed a simple toy example of how to perform a decentralized encoding and save a block transfer as compared to the classical encoding process.

Iterative Encoding: Unfortunately, the simple decentralized encoding depicted in Figure 1b cannot be easily adopted by codes with large n and k values. To solve this problem we propose an iterative encoding process that splits the coding in ν different steps and involves up to m nodes that store some of the rk block replicas. The coding can be described in two logical phases: (i) at each step, a node generates a temporary redundant block and forwards it to the next node, and (ii) after ν steps each of the m nodes locally combines the stored replicas with the temporary blocks it received to generate and store one of the m parity blocks $p_1, \ldots, p_m$.

Replica Collocation: Traditionally, distributed storage systems allocate the rk replicated blocks of each data object among different nodes at random, which guarantees with high probability that the different replicated blocks are stored in rk different nodes. Random replica placement balances the amount of data stored per node and guarantees high resiliency in face of correlated node failures. However, in the case of the previous iterative coding process, having only one block replica per node increases the number of steps ν required to obtain an MDS erasure code. To minimize the number of steps required to achieve MDS codes, and thus minimize encoding traffic, we propose to collocate ℓ out of the total $(r-1)k$ unused block replicas within the m coding nodes. By doing so, the coding nodes will have more information about the original data and would be able to reduce the number of encoding steps.

For this decentralized erasure coding process to be relevant, the benefits that they provide in terms of network resources should not be at the expense of fault tolerance. We will show that the fault tolerance of the proposed decentralized erasure codes depends on the values of ℓ and ν: the larger these values are, the more likely it is to achieve the MDS property. However, large values of ℓ impose strict placement policies which might complicate load balancing, while large values of ν increase the number of transferred blocks and thus reduce the benefit in terms of communication costs. It is then important to understand the trade-off between these two parameters to find high fault tolerance codes (preferably MDS codes) requiring low communication costs during the archiving process and flexible initial replica placement policies.

The drawback of collocation of ℓ replicas within m storage nodes is that a high collocation might reduce the tolerance of the storage system to correlated node failures. For this reason it is very important to keep low collocation rates (small value of ℓ). We also note that a traditional erasure coding process can also exploit the presence of nodes storing multiple blocks from a single object, and thus reduce the number of blocks downloaded in order to obtain the whole data object $\mathbf{o}$. Thus, in Section 4.3 we compare the required traffic of traditional erasure encoding process and that of the decentralized coding process, and find that for the same amount of collocation, decentralized coding always achieve the same or less traffic than the traditional coding, demonstrating its efficacy. Furthermore, in the cases where both coding schemes require the same traffic, a decentralized coding is preferred over a centralized one since it avoids the network and computing bottlenecks of having a single coding node.

3.2 A Motivating Example

To understand the proposed decentralized erasure coding process, we first provide as example the encoding of a (10,6) erasure code, which provides $m = 10 - 6 = 4$ blocks of redundancy (parity blocks). We have a data object $\mathbf{o} = (o_1, o_2, \ldots, o_6)$ to be stored with a replica placement policy that stores $r = 3$ replicas of $\mathbf{o}$, that is three replicas of every o_i, $i = 1, \ldots, 6$ (for a total of 18 data blocks). We assume that one of the replicas of $\mathbf{o}$ is stored in $k = 6$ different nodes, which will finally constitute the systematic part of the codeword, $c_1 = o_1, \ldots, c_k = o_k$. From the $(r-1)k = 12$ replicas left, we select a subset of ℓ of them to be stored in the $m = 4$ coding nodes that will carry out the decentralized encoding process. The assignment of these ℓ replicas is as follows:

$$\begin{aligned}\mathcal{N}_1 &= \{o_1, o_2, o_3\}\\ \mathcal{N}_2 &= \{o_4, o_5, o_6\}\\ \mathcal{N}_3 &= \{o_1, o_2\}\\ \mathcal{N}_4 &= \{o_3, o_4\}\end{aligned}$$

where $\mathcal{N}_j$ denotes the set of blocks stored in node j. Note that only $\ell = 10$ out of the available $(r-1)k = 12$ blocks are replicated in the m coding nodes, while the remaining two can be flexibly stored in other nodes, e.g., to balance the amount of data stored per node. Note also that no node stores any repeated block, since this would reduce fault tolerance.

To describe the decentralized encoding process we use an iterative encoding process of $\nu = 7$ steps, in which every $\psi_i, \xi_j \in \mathbb{F}_{2^q}$ are predetermined values that define the actual code instance. During step 1, node 1 which has $\mathcal{N}_1$ generates

$$x_1 = o_1\psi_1 + o_2\psi_2 + o_3\psi_3$$

and sends it to node 2, which uses $\mathcal{N}_2$ and x_1 to compute

$$x_2 = o_4\psi_4 + o_5\psi_5 + o_6\psi_6 + x_1\psi_7$$

during step 2. After two more steps, we get:

$$x_3 = o_1\psi_8 + o_2\psi_9 + x_2\psi_{10}$$
$$x_4 = o_3\psi_{11} + o_4\psi_{12} + x_3\psi_{13},$$

and node 4 forwards x_4 to node 1, since $\nu = 7 > m = 4$, which creates

$$x_5 = o_1\psi_{14} + o_2\psi_{15} + o_3\psi_{16} + x_4\psi_{17}$$

before sending x_5 to node 2. For the last two iterations, both node 2 and node 3 use respectively $\mathcal{N}_2$, x_1 and x_5, and $\mathcal{N}_3$, x_2 and x_3 together, to compute

$$x_6 = o_4\psi_{18} + o_5\psi_{19} + o_6\psi_{20} + x_1\psi_{21} + x_5\psi_{22}$$
$$x_7 = o_1\psi_{23} + o_2\psi_{24} + x_2\psi_{25} + x_6\psi_{26}.$$

After this phase, node 1 to 4 are locally storing:

$$\mathcal{N}_1 = \{o_1, o_2, o_3, x_4\}$$
$$\mathcal{N}_2 = \{o_4, o_5, o_6, x_1, x_5\}$$
$$\mathcal{N}_3 = \{o_1, o_2, x_2, x_6\}$$
$$\mathcal{N}_4 = \{o_3, o_4, x_3, x_7\}$$

from which they compute the final m parity blocks:

$$p_1 = o_1\xi_1 + o_2\xi_2 + o_3\xi_3 + x_4\xi_4$$
$$p_2 = o_4\xi_5 + o_5\xi_6 + o_6\xi_7 + x_1\xi_8 + x_5\xi_9$$
$$p_3 = o_1\xi_{10} + o_2\xi_{11} + x_2\xi_{12} + x_6\xi_{13}$$
$$p_4 = o_3\xi_{14} + o_4\xi_{15} + x_3\xi_{16} + x_7\xi_{17}.$$

The final codeword is $\mathbf{c} = [\mathbf{o}, \mathbf{p}] = (o_1, \dots, o_6, p_1, \dots, p_4)$. There is a total of ν blocks transmitted during the encoding process (those forwarded during the iterative phase). In this example, $\nu = 7$, and the encoding process requires two block transmissions less than the classical encoding process, which requires $n - 1 = 9$ blocks, thus achieving a 22% reduction of the traffic.

We will in fact analytically show (see Section 4) that this decentralized encoding obtains a (10,6) MDS code. It means that there is a set of values for the coefficients ψ_i and ξ_i, which can be determined (for example, by exhaustive search), guaranteeing maximum fault tolerance of the code. In fact, the larger the field $\mathbb{F}_{2^q}$ is, the more likely it is to obtain MDS codes, allowing to use random ψ_i and ξ_i values in practical settings [1].

3.3 General Code Construction

We now provide a general technique to generate the parity vector $\mathbf{p} = (p_1, \dots, p_m)$ in a decentralized manner by using m storage nodes that altogether store ℓ out of the total $(r-1)k$ block replicas.

We first define how the ℓ replicated blocks are allocated among the m coding nodes, i.e., the content of the set $\mathcal{N}_j$ for each node j. For the sake of simplicity, we assume that the ℓ replicas are deterministically assigned in a sequential manner as illustrated in the example used in 3.2, trying to even out the number of blocks assigned to each node. A formal description of this allocation is provided in Algorithm 1. Note that for practical small values of ℓ Algorithm 1 avoids the replication of the same block in a single node.

Algorithm 1. Replica placement policy

```
1: $i \leftarrow 1$
2: **for** $j = 1, \dots, m$ **do**
3:     $\alpha \leftarrow \lfloor \ell/m \rfloor$
4:     **if** $j \leq (\ell \mod m)$ **then**
5:         $\alpha \leftarrow \alpha + 1$
6:     **end if**
7:     $\mathcal{N}_j = \{o_l : l = (j \bmod k),\ j = i, \dots, i+\alpha\}$
8:     $i \leftarrow i + \alpha$
9: **end for**
```

This assignment policy imposes some restrictions on the location of the different replicated blocks. These restrictions might become a drawback in systems trying to uniformly distribute the storage load among all nodes in the system. The problem can be specially important in the extreme case when $\ell = (r-1)k$, where all replicas need to be specifically stored in the m coding blocks. However, smaller values of ℓ provide some flexibility on where to assign the $(r-1)k - \ell$ remaining replicas. We will explore the effects that different ℓ values have in the fault tolerance of the erasure code in Section 4.

Remark 1. In the case of $\ell = k$, there is no replica assignment policy and a random placement can be used.

Given the previous replica assignment policy, the decentralized encoding process is split in two different phases: the *iterative encoding* and the *local encoding.*

The iterative encoding consists of ν sequential encoding steps, where at each step, each node generates and forwards a temporary redundant block. For each step i, where $i = 1, \dots, \nu$, node $j = (i \bmod m)$ which stores the set of blocks $\mathcal{N}_j = \{z_1, z_2, \dots\}$ locally computes a temporary block $x_i \in \mathbb{F}_{2^q}$ as follows:

$$x_i = z_1\psi_1 + z_2\psi_2 + \cdots + z_{|\mathcal{N}_j|}\psi_{|\mathcal{N}_j|}, \tag{1}$$

where $\psi_i \in \mathbb{F}_{2^q}$ are predetermined values. Once x_i is computed, node j sends x_i to the next node $l = (i+1 \bmod m)$, who stores locally the new temporary block:

$\mathcal{N}_l = \mathcal{N}_l \cup \{x_i\}$. After that, node l computes x_{i+1} as defined in (1) and forwards it the next node. The iterative process is similarly repeated a total of ν times.

After this iterative encoding phase, each node $i = 1, \ldots, m$ executes a local encoding process where the stored blocks $\mathcal{N}_i$ (including the temporary blocks generated during the iterative encoding phase) are combined to generate the final parity block p_i (for predetermined values of $\xi_i \in \mathbb{F}_{2^q}$) as follows:

$$p_i = z_1\xi_1 + z_2\xi_2 + \cdots + z_{|\mathcal{N}_i|}\xi_{|\mathcal{N}_i|}. \tag{2}$$

Finally, we describe the overall distributed encoding algorithm (including the iterative encoding and the local encoding) in Algorithm 2. Note that values ψ_l and ξ_l (lines 7 and 17) are picked at random. In a sufficiently large field (e.g., when $q = 16$) this random choice will not introduce additional dependencies other than the ones introduced by the iterative encoding process itself [1].

Algorithm 2. Decentralized redundancy generation

```
 1: $l \leftarrow 1$
 2: $j \leftarrow 1$
 3: $x \leftarrow 0$
 4: for $i = 1, \ldots, \nu$ do                ▶ Generation of the $\nu$ temporary blocks.
 5:     $x \leftarrow 0$
 6:     for $z \in \mathcal{N}_j$ do           ▶ Coding operation as described in (1).
 7:         $x \leftarrow x + \psi_l \cdot z$
 8:         $l \leftarrow l + 1$
 9:     end for
10:     $j \leftarrow (i + 1) \bmod m$
11:     $\mathcal{N}_j \leftarrow \mathcal{N}_j \cup \{x\}$   ▶ Each union ($\cup$) represents a block transfer.
12: end for
13: $l \leftarrow 1$
14: for $i = 1, \ldots, m$ do                  ▶ Generation of the final $m$ parity blocks.
15:     $p_i \leftarrow 0$
16:     for $x \in \mathcal{N}_i$ do           ▶ Coding operation as described in (2).
17:         $p_i \leftarrow p_i + \xi_l \cdot x$
18:         $l \leftarrow l + 1$
19:     end for
20: end for
```

4 Evaluation

In this section we evaluate the effects that the number of collocated replicas, ℓ, and the number of steps, ν, have in the fault tolerance of the code obtained by the decentralized coding strategy. To do it, we symbolically compute different iterated codings as it is specified in Algorithm 2. We do it for different values of n, k, ℓ and ν, however, the block values o_i and the value of coefficients ψ_i and ξ_i are not specified, which means that after the iterative coding phase we obtain a symbolic codeword $\mathbf{c} = (c_1, \ldots, c_n)$. We use this codeword then to enumerate all the possible $\binom{n}{k}$ k-subsets of blocks in $\mathbf{c}$ and measuring how many of them contain k linearly independent blocks. We refer to the fault tolerance of the code, π, as the percentage of k-subsets that do not contain linear dependencies between its blocks. When all the $\binom{n}{k}$ k-subsets are free of linear dependencies we say that the code is MDS, and has maximum fault tolerance, i.e., $\pi = 1$.

Since we aim to evaluate the effects of the parameters ℓ and ν, we select three different (n,k) instances commonly used by in-production distributed storage systems: (i) a (6,3) code, suggested in the new Google FS [8], (ii) a (10,6) code used in the Microsoft Azure Storage service [10], and (iii) a (14,10) code used in Facebook's HDFS-RAID implementation [4, 15]. Respectively, these codes have storage footprints of $2\times$, $1.\dot{6}\times$ and $1.4\times$ the size of the original stored data, which represents a diverse spectrum of code parameters. For each of them we evaluate the fault tolerance of a code generated with a decentralized erasure code that uses ℓ collocated blocks and ν coding steps, for different values of ℓ and ν.

4.1 Fault Tolerance Analysis

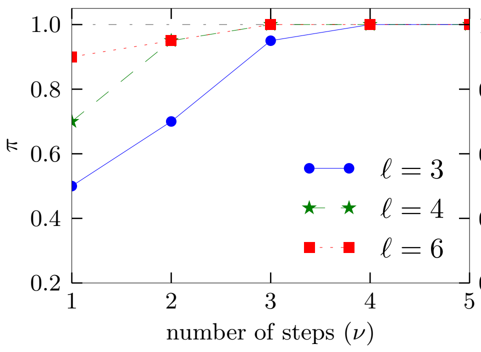
We divide the fault tolerance analysis in two experiments, one aiming to evaluate the effects of ν, and another one to evaluate the effects of ℓ.

In figures 2a, 2c, and 2e we show the fault tolerance of the code π as a function of the number of steps, ν. For each of the three different codes we depict the effects of ν for three different values of ℓ: $\ell = k$, $\ell = 1.5$ and $\ell = 2k$. We can see how the proportion of linearly independent k-subsets increases as more encoding iterations are executed. Achieving the maximum fault tolerance (when the fraction of linearly independent k-subsets is one) requires less iterations for high replica collocation values ℓ.

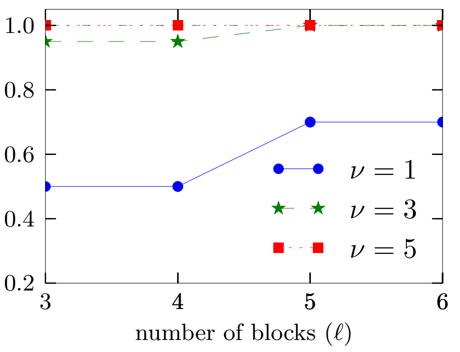
Similarly, in figures 2b, 2d, and 2f we show the fault tolerance as a function of the number of blocks stored within the m coding nodes, ℓ. For each code we also show the results for three different values of ν, which aim to show the fault tolerance when all (i) only a few coding nodes execute the iterative encoding process, (ii) when all coding nodes execute it exactly once, and (iii) when some coding nodes execute it more than once. In general we can see how increasing the number of initially collocated replicas ℓ increases the fault tolerance of the code. However, for small values of ν there are cases where increasing ℓ might slightly reduce the fault tolerance. Finally, we want to note that in those cases where $\nu \leq m$ (only a few coding nodes execute the iterative encoding), the code produced by the decentralized coding can never achieve maximum fault tolerance. To achieve maximum fault tolerance all the m coding nodes need to execute at least one coding step.

4.2 Obtaining an (6,3) MDS Code

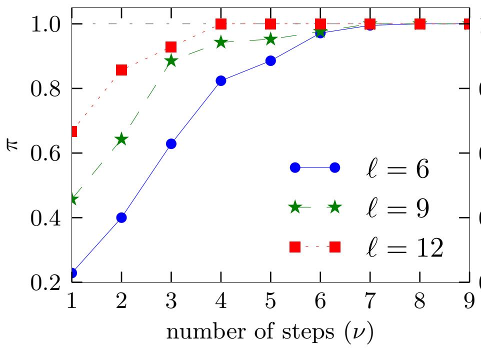
We propose a simple example to understand why the iterative encoding process allows to obtain MDS codes. Suppose we have a (6,3) erasure code, whose codewords are of the general form



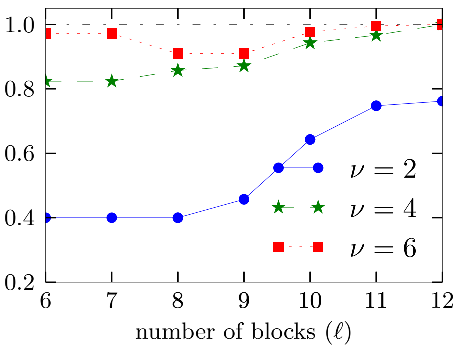
(a) Results for (6,3) code



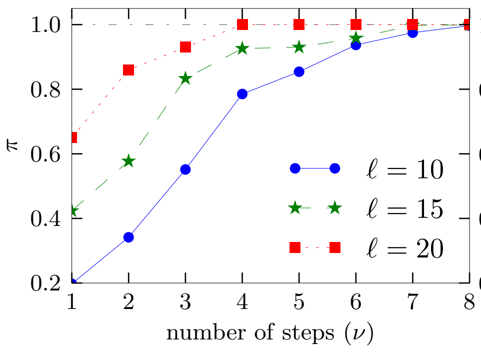
(b) Results for (6,3) code.



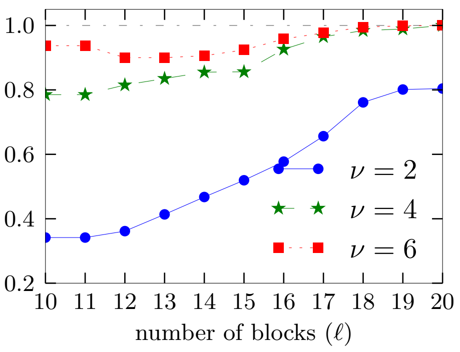
(c) Results for (10,6) code



(d) Results for (10,6) code.



(e) Results for (14,10) code



(f) Results for (14,10) code.

Fig. 2. Fault tolerance achieved by our decentralized erasure coding process as a function of the number of encoding steps, ν, and the number of co-located block replicas, ℓ. The fault tolerance π is expressed as the proportion of k-subsets of the codeword $\mathbf{c}$ that do not contain linear dependencies. When this value is one, the code is MDS and has maximum fault tolerance.

$$\mathbf{c} = (o_1,\ o_2,\ o_3,\ \alpha_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3,\ \beta_1 o_1 + \beta_2 o_2 + \beta_3 o_3,\ \gamma_1 o_1 + \gamma_2 o_2 + \gamma_3 o_3)$$

for some fixed $\alpha_i, \beta_i, \gamma_i \in \mathbb{F}_{2^q}$, $i = 1, 2, 3$, where $\mathbf{o} = (o_1, o_2, o_3)$ is the object to be encoded. We assume that every α_i, β_i, γ_i is nonzero, so that it is invertible. Note that if any of them were to be zero, then the code cannot be MDS.

Let us assume a replica placement policy using $\ell = 3$ that allocates these ℓ replicas within the m coding blocks as follows:

$$\mathcal{N}_1 = \{o_1\},\quad \mathcal{N}_2 = \{o_2\},\quad \mathcal{N}_3 = \{o_3\}.$$

Then, an iterative encoding process of $\nu = 4$ steps allows to compute the generic parity blocks as given above:

1. Node 1 sends o_1 to node 2,
2. Node 2 uses o_2 and $x_1 = o_1$ to compute $x_2 = o_1\gamma_1 + o_2\gamma_2$.
3. Node 3 receives x_2 and sends $x_3 = \gamma_2^{-1}\alpha_2 x_2 + \alpha_3 o_3 = \gamma_2^{-1}\alpha_2\gamma_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3$.
4. Node 1 forwards x_3 to node 2.

After this phase, node 1 to 3 are locally storing:

$$\begin{aligned}
\mathcal{N}_1 &= \left\{o_1, x_3 = \gamma_2^{-1}\alpha_2\gamma_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3\right\}\\
\mathcal{N}_2 &= \left\{o_1, o_2, x_3 = \gamma_2^{-1}\alpha_2\gamma_1 o_1 + \alpha_2 o_2 + \alpha_3 o_3\right\}\\
\mathcal{N}_3 &= \{o_3, x_2 = o_1\gamma_1 + o_2\gamma_2\}
\end{aligned}$$

from which they compute the final m parity blocks:

$$\begin{aligned}
p_1 &= o_1(\gamma_2^{-1}\alpha_2\gamma_1 + \alpha_1) + x_3\\
p_2 &= \alpha_3^{-1}\beta_3 x_3 + (\alpha_3^{-1}\beta_3\gamma_2^{-1}\alpha_2\gamma_1 + \beta_1)o_1 + (\alpha_3^{-1}\beta_3\alpha_2 + \beta_2)o_2\\
p_3 &= \gamma_3 o_3 + x_2.
\end{aligned}$$

The final codeword is $\mathbf{c} = [\mathbf{o}, \mathbf{p}] = (o_1, o_2, o_3, p_1, p_2, p_3)$. Thus any (6,3) MDS code can be obtained through this iterative encoding.

4.3 Reduction of the Encoding Traffic

Finally, we aim to evaluate the traffic savings that the decentralized erasure coding provides on the data archival process. For that we take the results presented in Figure 2 and measure the encoding traffic of the MDS codes obtained when $\ell = k$ and $\ell = 2k$. For each ℓ value the decentralized encoding traffic corresponds to the minimum value of ν required to achieve the MDS property. Additionally, for the same value of collocated replicas ℓ we also evaluate the encoding traffic that a traditional erasure would require, considering that the coding node stores more than one object block.

In Figure 3 we depict the encoding traffic comparison between a centralized coding process, denoted by RS[2], and a decentralized MDS coding process, denoted by DE. In the case of the (6,3) code, there are only traffic savings when

[2] The acronym refers to the classical Reed-Solomon coding process.

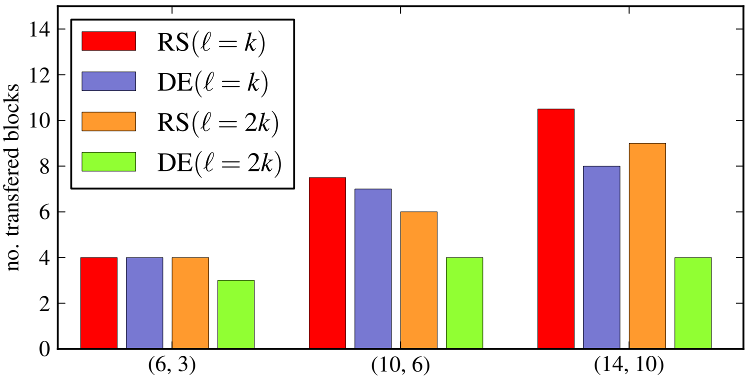


Fig. 3. Comparison of the number of transferred blocks during the encoding of a classical Reed-Solomon code (RS) and the decentralized coding (DE) for two different replica collocation values: $\ell = k$ and $\ell = 2k$. All DE codes are MDS codes optimized to minimize the number of coding steps ν.

the $m = 3$ coding nodes store all the $(r-1)k$ replicas. In this case the decentralized coding saves one block transfer. In the case of the (10,6) the decentralized coding process always requires less network traffic, even for low replica collocation levels, and these traffic savings are amplified for the (14,10) code. In this last case the savings range from a 24% in the case of the low replica collocation ($\ell = k$), up to 56% for high collocation values ($\ell = 2k$).

5 Related Work

Despite widespread use of erasure coding for archiving data in distributed storage systems, the study of the actual migration process from replication based redundancy to erasure code based redundancy is in its infancy.

The most relevant related work is that of Fan et al. [5], who propose to distribute the task of erasure coding using the Hadoop infrastructure, as MapReduce tasks. Any individual object is however encoded at a single node, and hence the parallelism achieved in their approach is only at the granularity of individual data objects. Besides, their data archiving process relies on the traditional erasure code redundancy generation process, which does not exploit previously existing replicas.

In [12] we recently proposed RapidRAID codes, a family of pipelined erasure code that aim to speedup the archival process in distributed storage systems. Although such fast data archival is also achieved by a decentralized coding process, the RapidRAID approach differs fundamentally from the one presented in this paper in two ways. First, RapidRAID codes are non-systematic codes that require decoding operations to read the archived data, complicating the access to archived data. Second, RapidRAID codes are aimed to reduce the encoding time, but achieves no reduction of traffic as compared to what is required by a traditional coding processes. The presented work in contrast is specifically aimed at traffic reduction, and any collateral benefits in terms of speed-up of the process is left for future study.

Finally, a somewhat unrelated line of work worth mentioning are codes designed for storage in sensor networks [6, 11]. However, in such a setting, the (disjoint) data generated by k sensors is jointly stored over $n > k$ storage sensors based on erasure coding redundancy. This is achieved using network coding techniques by creating random linear combinations of the already distributed data. Such a technique has however not been explored for the specific migration problem studied in this work.

6 Conclusions

In this paper we introduce a new decentralized erasure coding process to reduce the network traffic required to archive replicated data in distributed storage systems. The decentralized process distributes the coding tasks among those nodes storing data block replicas of the object to be archived. These nodes collaboratively generate and store all the parity data.

Additionally, we provide a formal definition of the decentralized erasure coding process and symbolically analyze the fault tolerance of the obtained codes for different parameters. We show that in already deployed systems where the placement of the replicated data can not be changed, our decentralized coding process can reduce the redundancy generation traffic by 20% upto 38% for typical code configurations used in current systems. However, when the replica placement of newly inserted data can be manipulated to co-locate more block replicas in some specific manner in the nodes participating in the coding process, the redundancy generation traffic can be reduced by 40% upto 70%.

The design of decentralized erasure coding process to archive replicated data is a relatively unexplored problem that needs to be further studied. We identify two problems that we plan to address in future works. Specifically, we aim to look methodically at the effects that different replica placement policies have on the network traffic required during the archiving process. We also want to generalize the idea to decentralize the redundancy generation of existing standard erasure codes such as Reed-Solomon codes. Ultimately, we want to develop a holistic theory, which explores any possible trade-offs between the network traffic generated by and the speed of the archival process, subject to various other system design choices such as the initial replica placement.

Acknowledgments. We would like to thank Dr. Punarbasu Purkayastha for his valuable comments on this research. L. Pamies-Juarez and F. Oggier's research is supported by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03. A. Datta's work is supported by A*Star TSRP grant number 1021580038 and NTU/MoE Tier-1 grant number RG 29/09.

References

1. Acedański, S., Deb, S., Médard, M., Koetter, R.: How good is random linear coding based distributed networked storage. In: Workshop on Network Coding, Theory, and Applications, NetCod (2005)
2. Amazon.com. Amazon S3, http://aws.amazon.com/s3
3. Apache.org. HDFS, http://hadoop.apache.org/hdfs/
4. Apache.org. HDFS-RAID, http://wiki.apache.org/hadoop/HDFS-RAID
5. Fan, B., Tantisiriroj, W., Xiao, L., Gibson, G.: DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing. Technical Report Technical Report CMU-PDL-11-112, Carnegie Mellon Univsersity, Parallel Data Laboratory (2011)
6. Dimakis, A., Prabhakaran, V., Ramchandran, K.: Decentralized erasure codes for distributed networked storage. IEEE/ACM Transactions on Networking 14 (2006)
7. Fan, B., Tantisiriroj, W., Xiao, L., Gibson, G.: DiskReduce: RAID for Data-Intensive Scalable Computing. In: The 4th Annual Workshop on Petascale Data Storage, PDSW (2009)
8. Ford, D., Labelle, F., Popovici, F.I., Stokely, M., Truong, V.A., Barroso, L., Grimes, C., Quinlan, S.: Availability in Globally Distributed Storage Systems. In: The 9th USENIX Conference on Operating Systems Design and Implementation, OSDI (2010)
9. Ghemawat, S., Gobioff, H., Leung, S.: The Google File System. In: Proceedings of the ACM Symposium on Operating Systems Principles, SOSP (2003)
10. Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., Yekhanin, S.: Erasure Coding in Windows Azure Storage. In: Proceedings of the USENIX Annual Technical Conference, ATC (2012)
11. Kamra, A., Misra, V., Feldman, J., Rubenstein, D.: Growth codes: maximizing sensor network data persistence. In: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM (2006)
12. Pamies-Juarez, L., Datta, A., Oggier, F.E.: RapidRAID: Pipelined Erasure Codes for Fast Data Archival in Distributed Storage Systems. CoRR, abs/1207.6744 (2012)
13. Reed, I., Solomon, G.: Polynomial Codes Over Certain Finite Fields. Journal of the Society for Industrial and Applied Mathematics 8(2), 300–304 (1960)
14. Rodrigues, R., Zhou, T.H.: High Availability in DHTs: Erasure Coding vs. Replication. In: van Renesse, R. (ed.) IPTPS 2005. LNCS, vol. 3640, pp. 226–239. Springer, Heidelberg (2005)
15. Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A.G., Vadali, R., Chen, S., Borthakur, D.: Novel Codes for Cloud Storage (2012), https://mhi.usc.edu/files/2012/04/Sathiamoorthy-Maheswaran.pdf
16. Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., Liu, H.: Data warehousing and analytics infrastructure at facebook. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010 (2010)
17. Weatherspoon, H., Kubiatowicz, J.D.: Erasure Coding Vs. Replication: A Quantitative Comparison. In: Druschel, P., Kaashoek, M.F., Rowstron, A. (eds.) IPTPS 2002. LNCS, vol. 2429, pp. 328–337. Springer, Heidelberg (2002)