Data insertion and archiving in erasure-coding based large-scale storage systems

Pamies-Juarez, Lluis; Oggier, Frederique; Datta, Anwitaman

2013

Pamies-Juarez, L., Oggier, F., & Datta, A. (2013). Data Insertion and Archiving in Erasure-Coding Based Large-Scale Storage Systems. Distributed Computing and Internet Technology, 7753, pp.47-68.

https://hdl.handle.net/10356/95760

https://doi.org/10.1007/978-3-642-36071-8_4

Data Insertion & Archiving in Erasure-coding Based Large-scale Storage Systems

Lluis Pamies-Juarez[1], Frédérique 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. Given the vast volume of data that needs to be stored reliably, many data-centers and large-scale file systems have started using erasure codes to achieve reliable storage while keeping the storage overhead low. This has invigorated the research on erasure codes tailor made to achieve different desirable storage system properties such as efficient redundancy replenishment mechanisms, resilience against data corruption, degraded reads, to name a few prominent ones. A problem that has mainly been overlooked until recently is that of how the storage system can be efficiently populated with erasure coded data to start with. In this paper, we will look at two distinct but related scenarios: (i) *migration to archival* - leveraging on existing replicated data to create an erasure encoded archive, and (ii) *data insertion* - new data being inserted in the system directly in erasure coded format. We will elaborate on coding techniques to achieve better throughput for data insertion and migration, and in doing so, explore the connection of these techniques with recently proposed locally repairable codes such as self-repairing codes.

1 Introduction

The ability to store securely and reliably the vast amount of data that is continuously being created by both individuals and institutions is a cornerstone of our digital society. A study sponsored by the information storage company *EMC* estimated that the world's data would have reached 1.8 zettabytes of data to be stored by the end of 2011.[3]

The massive volume of data involved means that it would be extremely expensive, if not impossible, to build a single piece of hardware with enough storage as well as I/O capabilities to meet the needs of most organizations and businesses. A practical alternative is to scale out (or horizontally): resources from multiple interconnected storage nodes are pooled together, and more such nodes can be organically added as and when the demand for storage resources grows. We call these systems *Networked Distributed Storage Systems* (NDSS). NDSS come in many flavors such as data centers and peer-to-peer (P2P) storage/backup systems, which have their unique characteristics, but also share several common

[3] `http://www.emc.com/about/news/press/2011/20110628-01.htm`

properties. Given the system scale, failure of a significant subset of the constituent nodes, as well as other network components, is a norm rather than the exception. To enable a highly available overall service, it is thus essential to both tolerate short-term outages of some nodes and to provide resilience against permanent failures of individual components. Fault-tolerance is achieved using redundancy while long-term resilience relies on replenishment of lost redundancy over time. To that end, erasure codes have become popular to achieve system resilience while incurring low storage overhead. Recent years have accordingly witnessed the design of erasure codes tailor made to meet distributed storage system needs, more specifically, a lot of work has been done to improve the storage system's repairability. This line of work has been surveyed in [4, 6].

This article focuses on a different aspect of erasure code design. A relatively unexplored problem in the literature is, how does the erasure coded data come into being to start with?

In a replication based NDSS, when a new object needs to be stored, the first node receiving the same can forward it to another node to replicate the data, and so on. Such a pipelined approach allows quick data insertion and replication in the system, the load of data insertion is distributed among multiple nodes, and a single node is not overloaded with the task.

In contrast, in an erasure coding based NDSS, traditionally, one node has the burden to first encode the new object (after obtaining it if necessary), and then distribute the encoded pieces to other storage nodes. The computational and communication resources of this node thus become a bottleneck. In this paper we summarize some recent results delving into two distinct scenarios, where distributing the load to create erasure coded data improves the throughput of populating the NDSS with erasure coded data.

Note that in the following, we use the term 'source' to mean whichever node has a full copy of the data. It could be a gateway node that receives the data from an end-user who 'owns' and uploads the same to the NDSS, or it could be an NDSS node where the data is created locally by some application.

Migration to archival. If the data is originally stored in the system in replicated format, but subsequently needs to be migrated into erasure coding based archive, then existing replicas of the data can be exploited to distribute the load of the erasure coding process. This scenario is typical since newly arrived objects are often stored as replicas, which ensures efficient reads and fault tolerance while the objects are being frequently manipulated. When accesses to the objects become rarer, they are archived using erasure coding, and the replicas are discarded.

Data insertion. When new data is being inserted into the system, if such a data is to be stored in NDSS directly in erasure coded format, then the computational resources of the storage nodes can be utilized to reduce the amount of redundant data that the source needs to itself create and inject individually to the different storage nodes.

For these two distinct scenarios - migration to archival and data insertion - we have devised (so far unrelated) mechanisms [7–9] to improve the system's throughput. Before summarizing these approaches, we next provide a brief background of erasure codes for NDSS.

2 Background on Erasure Coding for Distributed Storage

We can formally define the erasure encoding process as follows. Let the vector $\mathbf{o} = (o_1, \ldots, o_k)$ denote a data object of $k \times q$ bits, that is, each symbol o_i, $i = 1, \ldots, k$ 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, $i = 1, \ldots, k$ then belong to the binary extension field $\mathbb{F}_{2^q}$ containing 2^q elements. 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 $\mathbf{c} = (c_1, \ldots, c_n)$ 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, $m = n - k$, 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 reconstructed without requiring a decoding process by accessing these systematic pieces.

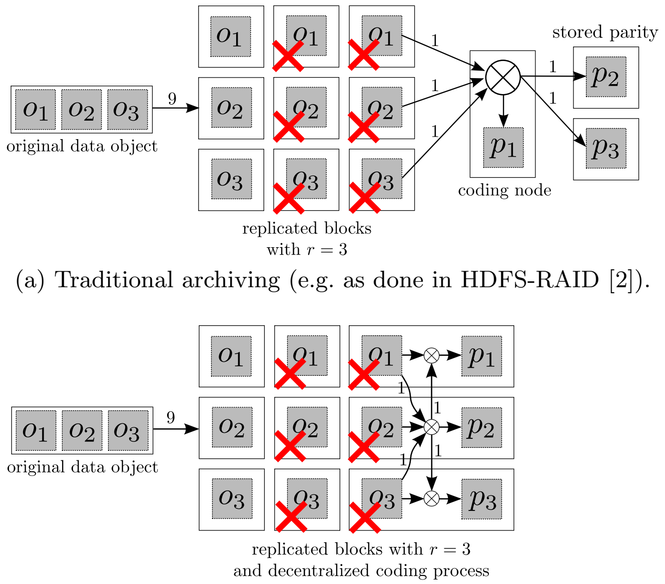
If any arbitrary k of the c_is can be used to reconstruct the original information $\mathbf{o} = (o_1, \ldots, o_k)$, then the code is said to be *maximum distance separable* (MDS). For any given choice of n and k, an MDS code provides the best fault-tolerance for up to $n - k$ arbitrary failures.

Systematic MDS codes have thus traditionally been preferred in storage systems, given the practical advantages they provide.

Other system considerations may however prompt for non-MDS and/or non-systematic codes as well. For instance, dependencies within small subsets of codewords may allow for better and faster repairs [5] - which is desirable for long-term resilience of the system, even if it marginally deteriorates the system's tolerance of the number of simultaneous faults. In fact, in the recent years, repairable erasure codes have been vigorously researched [4, 6], and it continues to be an open-end and popular topic of study. Likewise, non-systematic codes may provide a basic level of confidentiality [3] of the stored content since an adversary with access to any one (or very small number of) storage node(s) will not be able to see the content.

3 Migration from Replication to Erasure Coded Archive

Often, when new data is introduced in a storage system, it is replicated (3-way replication is a popular approach) for fault-tolerance. Furthermore, such a replication strategy can be leveraged to support higher throughput of data access when different applications are trying to read the same data simultaneously, or by migrating computing tasks to a relatively less loaded subset of the replicas instead of moving the data around. Data is often accessed and manipulated



(a) Traditional archiving (e.g. as done in HDFS-RAID [2]).

(b) Decentralized encoding.

Fig. 1. An example of migration of replicated data into an erasure coded archive. The squares represent storage nodes and the arrows across boxes denote data (labels indicate the actual amount) transferred over the network. We can see how (a) requires a total of *five* network transfers while (b) needs only *four* network transfers. The "X" symbol denotes the replicas that are discarded once the archival finishes, and the symbol $\otimes$ denotes an encoding operation.

frequently when it has been acquired recently, but over time, once the need to access it decreases, it is desirable to reduce the storage overhead by replacing the replicas and instead archive the data using erasure coding [2]. We propose to effectuate this migration via a decentralized encoding process, which, as we will see later, provides significant performance boosts.

3.1 Motivating Examples

We will use Figure 1 for an illustration of an abstracted toy example of migration of a replicated system into an erasure coded archival. Suppose that we have a data object comprising of three parts (o_1, o_2, o_3), and the parts o_i are stored at three different nodes each (i.e., using a 3-way replication). Traditionally, one node (the coding node) would require (o_1, o_2, o_3) to be collocated, based on which it can compute some parity coefficients (p_1, p_2, p_3), keeping p_1 for itself and then distributing p_2 and p_3. This will require a total of five network transfers including the communication cost of downloading the three original parts to start with. Alternatively, one of the nodes which already stores o_2 could download o_1 and o_3, from which it can compute the parity p_2, and then distributes p_2 to one of the nodes which stored o_1 (respectively o_3), each of which could compute parity pieces p_1 and p_3 respectively using p_2 and o_1 (respectively o_3). This process where the existing replicas are exploited costs only four network transfers. In both cases, the codeword $(c_1, \dots, c_6) = (o_1, o_2, o_3, p_1, p_2, p_3)$ has been computed and distributed over six different nodes. The remaining pieces of the original data can be garbage collected subsequently. Also note that if each storage node had a communication bottleneck, such that it could only upload or download one piece in an unit time, then the centralized approach would have required five time units for data transfer, in addition to the computation time. If p_2 is pipelined to o_3 via o_1 (not shown in the figure) then the decentralized approach would take three time units for data transfer, in addition to the computation time at the three different nodes, some of which happens in parallel.

The above toy example illustrates that there is a potential benefit both in terms of reduced network traffic, and possibly also in terms of the time taken to carry out a single encoding, by decentralizing the process and leveraging on existing replicas. The example however does not reveal what the parity pieces p_is are, and what specific computations need to be carried out. In fact, as we will see next, different strategies can be devised, depending on how many replicas are present in the system when the migration to erasure coded archive starts, what is the original placement of these replicas, how is the encoding process distributed and what are the properties of the resulting codeword.

We will next give two explicit toy examples, to illustrate two types of decentralized encoding. The first one ends up in a non-systematic codeword, while the second yields a systematic codeword. The first one assumes that each piece of the object and its replica(s) are initially placed in different nodes. The latter assumes that some of these pieces are cleverly collocated, in order to achieve further savings in communication. Subsequently, we will elaborate on a more general theory of decentralized coding, along with some early results.

Example 1. Suppose that an object $\mathbf{o} = (o_1, o_2, o_3, o_4),\ o_i \in \mathbb{F}_{2^q}$, of $k = 4$ blocks is stored over $n = 8$ nodes using two replicas of $\mathbf{o}$, which are initially scattered as follows:

$$\mathcal{N}_1 :\ o_1, \mathcal{N}_2 :\ o_2, \mathcal{N}_3 :\ o_3, \mathcal{N}_4 :\ o_4,$$
$$\mathcal{N}_5 :\ o_1, \mathcal{N}_6 :\ o_2, \mathcal{N}_7 :\ o_3, \mathcal{N}_8 :\ o_4.$$

Clearly, such a setup can guarantee fault-tolerance against only a single arbitrary failure, though some combinations of multiple failures may also be tolerated, if one copy of each piece survives in the system. Consider that it needs to be archived using a codeword $\mathbf{c} = (c_1, \dots, c_8)$. In this example, the resulting erasure coded data coincidentally has the same storage overhead as the original replication scheme (though this is not necessary, as we will see latter when we generalize the ideas), but will have significantly higher fault tolerance.

Unlike in Figure 1 where one node distributes its data to two other storage nodes, in this example, since more nodes are involved, the coding process is done in a pipelined manner, namely node 1 forwards $o_1\psi_1$, i.e. some multiple of o_1, to node 2, which computes a linear combination of the received data with o_2, and forwards it again to node 3, and so on. More generally, node i encodes the data it gets from the previous node together with the data it already has and forwards it to the next node. We denote the data forwarded from node i to its successor, node j, by $x_{i,j}$, which is defined as follows:

$$\begin{aligned}
x_{1,2} =&\ o_1\psi_1, \\
x_{2,3} =&\ x_{1,2} + o_2\psi_2 = o_1\psi_1 + o_2\psi_2, \\
x_{3,4} =&\ x_{2,3} + o_3\psi_3 = o_1\psi_1 + o_2\psi_2 + o_3\psi_3, \\
x_{4,5} =&\ x_{3,4} + o_4\psi_4 = o_1\psi_1 + o_2\psi_2 + o_3\psi_3 + o_4\psi_4, \\
x_{5,6} =&\ x_{4,5} + o_1\psi_5 = o_1(\psi_1 + \psi_5) + o_2\psi_2 + o_3\psi_3 + o_4\psi_4, \\
x_{6,7} =&\ x_{5,6} + o_2\psi_6 = o_1(\psi_1 + \psi_5) + o_2(\psi_2 + \psi_6) + o_3\psi_3 + o_4\psi_4, \\
x_{7,8} =&\ x_{6,7} + o_3\psi_7 = o_1(\psi_1 + \psi_5) + o_2(\psi_2 + \psi_6) + o_3(\psi_3 + \psi_7) + o_4\psi_4,
\end{aligned}$$

where $\psi_j \in \mathbb{F}_{2^q}$, $j = 1, \dots, 7$, are some predetermined values. After the nodes have distributed their stored data, they are left to generate an element of the final codeword c_i by encoding the received data together with the locally available data as follows:

$$\begin{aligned}
&c_1 = o_1\xi_1, \\
&c_2 = x_{1,2} + o_2\xi_2 = o_1\psi_1 + o_2\xi_2, \\
&c_3 = x_{2,3} + o_3\xi_3 = o_1\psi_1 + o_2\psi_2 + o_3\xi_3, \\
&c_4 = x_{3,4} + o_4\xi_4 = o_1\psi_1 + o_2\psi_2 + o_3\psi_3 + o_4\xi_4, \\
&c_5 = x_{4,5} + o_1\xi_5 = o_1(\psi_1 + \xi_5) + o_2\psi_2 + o_3\psi_3 + o_4\psi_4, \\
&c_6 = x_{5,6} + o_2\xi_6 = o_1(\psi_1 + \psi_5) + o_2(\psi_2 + \xi_6) + o_3\psi_3 + o_4\psi_4, \\
&c_7 = x_{6,7} + o_3\xi_7 = o_1(\psi_1 + \psi_5) + o_2(\psi_2 + \psi_6) + o_3(\psi_3 + \xi_7) + o_4\psi_4, \\
&c_8 = x_{7,8} + o_4\xi_8 = o_1(\psi_1 + \psi_5) + o_2(\psi_2 + \psi_6) + o_3(\psi_3 + \psi_7) + o_4(\psi_4 + \xi_8),
\end{aligned}$$

where $\xi_j \in \mathbb{F}_{2^q}$, $j = 1, \dots, 8$, are also predetermined values.

Note that the coding process can be implemented in a pipelined manner, and both phases can be executed simultaneously: as soon as node i receives the first few bytes of $x_{i-1,i}$ it can start generating the first bytes of c_i, and concurrently forward $x_{i,i+1}$ to node $i+1$.

The end result is a non-systematic codeword $(c_1, \dots, c_8)$ that has a high fault tolerance. By selecting values of ψ_i and ξ_i that do not introduce linear dependencies within the codeword, the original object $\mathbf{o}$ can be reconstructed from any combination of four codeword symbols except the combination $\{c_1, c_2, c_5, c_6\}$ [8]. In this specific case the symbols c_1, c_2, c_5 and c_6 are linearly dependent (this can be checked using symbolic computations) since:

$$c_1\left[(\psi_1\xi_6\xi_2^{-1} + \psi_5 + \xi_5)\xi_1^{-1}\right] + c_2\left[\xi_6\xi_2^{-1}\right] + c_5 + c_6 = 0,$$

recalling that $2 \equiv 0$ in $\mathbb{F}_{2^q}$. Then, this (8,4) code has $\binom{8}{4} - 1 = 69$ possible 4-subsets of codeword symbols that suffice to reconstruct the original object $\mathbf{o}$. It represents a negligible degradation with respect to the fault tolerance of an MDS code, where $\binom{8}{4} = 70$ such possible 4-subsets exist.

Example 2. The second example is that of a systematic (10,6) erasure code, which provides $m = 10 - 6 = 4$ blocks of redundancy (parity blocks). Consider 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, $c_1 = o_1, \ldots, c_k = o_k$. Of the $(r-1)k = 12$ remaining replica pieces 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:

$$\mathcal{N}_1 = \{o_1, o_2, o_3\}\,;\ \mathcal{N}_2 = \{o_4, o_5, o_6\}\,;\ \mathcal{N}_3 = \{o_1, o_2\}\,;\ \mathcal{N}_4 = \{o_3, o_4\}\,,$$

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. Such a collocation of multiple pieces reduces the amount of fault-tolerance enjoyed by the data while it is stored using replication.[4] Note also that no node stores any repeated block, since this would further 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:

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

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

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

[4] By carrying out erasure coding of subset of pieces from different objects, one may be able to alleviate this problem of initial fault-tolerance, while still using precisely the same scheme.

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

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

from which they compute the final m parity blocks:

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

As in Example 1, all values $\psi_i, \xi_i \in \mathbb{F}_{2^q}$ are also predetermined to optimize fault-tolerance.

The final codeword is $\mathbf{c} = [\mathbf{o}, \mathbf{p}] = (o_1, \ldots, o_6, p_1, \ldots, p_4)$. 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 classic encoding process, which requires $n - 1 = 9$ blocks, thus achieving a 22% reduction of the traffic.

We will next elaborate the general theory of each of the two variations of the decentralized codes, along with summary of some results.

3.2 Generating a non-systematic erasure code

Example 1 is a particular case of RapidRAID codes [8] for $k = 4$ and $n = 8$. We next present a general definition of RapidRAID codes [8] for any pair (n, k) of parameters, where $n \leq 2k$. We start by stating the requirements that RapidRAID imposes on how data must be stored:

- When $n < 2k$, two of the originally stored replicas should be overlapped between n storage nodes: a replica of $\mathbf{o}$ should be placed in nodes 1 to k, and a second replica of $\mathbf{o}$ in nodes from $n - k$ to n.
- The final n redundancy blocks forming $\mathbf{c}$ have to be generated (and finally stored) in nodes that were already storing a replica of the original data.

We then formally define the temporal redundant block that each node i in the pipelined chain sends to its successor as:

$$x_{i,i+1} = x_{i-1,i} + \sum_{o_j \in \text{node i}} o_j\psi_i,\ 1 < i < n - 1, \tag{1}$$

with $x_{0,1} = 0$, while the final redundant block c_i generated/stored in each node i is:

$$c_i = x_{i-1,i} + \sum_{o_j \in \text{node i}} o_j\xi_i,\ 1 < i < n, \tag{2}$$

where $\psi_i, \xi_i \in \mathbb{F}_{2^q}$ are static predetermined values specifically chosen to guarantee maximum fault tolerance.

3.3 Generating a systematic erasure code

We will next assume that only $m = n - k$ nodes will participate in the encoding process (k nodes are storing the systematic pieces), as illustrated in Example 2. However, the proposed strategy also requires a carefully chosen collocation of several distinct replica pieces within the same node that participates in the decentralized encoding process.

Then, a total of ℓ different block replicas are allocated (collocated) 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 Example 2, trying to even out the number of blocks assigned to each node. A formal description of this allocation is provided in Algorithm 1.

Algorithm 1 Replica placement policy.

```
$i \leftarrow 1$
for $j = 1, \ldots, m$ do
    $\alpha \leftarrow \lfloor \ell/m \rfloor$
    if $j \leq (\ell \mod m)$ then
        $\alpha \leftarrow \alpha + 1$
    end if
    $\mathcal{N}_j = \{o_l : l = (j \bmod k),\ j = i, \ldots, i + \alpha\}$
    $i \leftarrow i + \alpha$
end for
```

This assignment policy imposes some restrictions on the location of the different replicated blocks (block collocation), which might require changes on the random assignment policy commonly used in NDSS. Furthermore, collocating block replicas in a same node reduces the fault tolerance of replicated data. This problem can be of special importance for the extreme case when $\ell = (r-1)k$ (all replicas are stored within only m nodes), although for small values of ℓ the assignment policy provides some flexibility on where to assign the $(r-1)k-\ell$ remaining replicas. However, as we will show in Section 3.5, small ℓ values increase the chances of introducing linear dependencies during the distributed encoding process, reducing the resiliency of the encoded data. In this last case the negative effects of a small ℓ value can be counterbalanced by adopting larger ν values. There is then a trade-off between the values ℓ and ν, and the fault tolerance of the replicated and encoded data.

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

Given the replica assignment policy, the decentralized encoding process is split into 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 $i = 1, \ldots, \nu$, node $j = (i \bmod m)$ which stores the set of blocks $\mathcal{N}_j = \{z_1, z_2, \ldots\}$ locally computes a temporary block $x_i \in \mathbb{F}_{2^q}$ as follows:

$$x_i = z_1\psi_1 + z_2\psi_2 + \cdots + z_{|\mathcal{N}_j|}\psi_{|\mathcal{N}_j|}, \tag{3}$$

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 $l = (i + 1 \bmod m)$, which stores locally the new temporary block: $\mathcal{N}_l = \mathcal{N}_l \cup \{x_i\}$, after which, node l computes x_{i+1} as defined in (3) and forwards it to 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:

$$p_i = z_1\xi_1 + z_2\xi_2 + \cdots + z_{|\mathcal{N}_i|}\xi_{|\mathcal{N}_i|}. \tag{4}$$

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 (w.h.p.) 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 (3).
 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 (4).
17:         $p_i \leftarrow p_i + \xi_l \cdot x$
18:         $l \leftarrow l + 1$
19:     end for
20: end for
```

3.4 General Distributed Encoding Framework

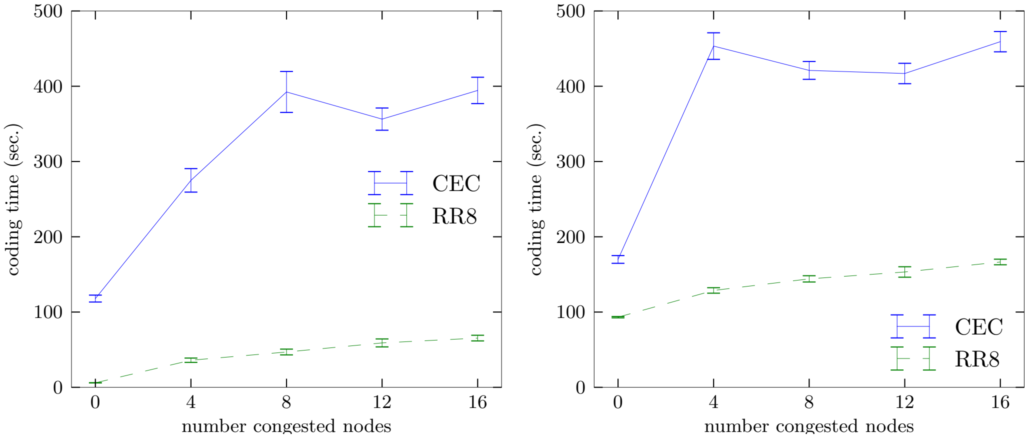
Both distributed encoding schemes follow the same approach: a set of n nodes, labelled from node 1 to node n, store an original configuration of data pieces. Then one node starts the encoding process by transmitting to the next node (next according to the labeling) a linear combination of the pieces it stores. The second node then keeps what it receives, what it owns, and sends a combination of both to the next node, and this is iterated until a codeword is computed.

- In the first case, the first node starts the process, which is repeated until reaching the nth node. The original configuration assumes, if $n = 2k$, that node 1 to node k each stores one piece of the data object, as does every node, from node $k + 1$ to node n. If $n < 2k$, the two copies of the original object are initially overlapped. This results in a non-systematic codeword.
- In the second case, to ensure that the codeword will be systematic, node 1 to node k store the k pieces of the original object, while nodes $k + 1$ to n store different configurations of the same. The iterative encoding only involves the $n - k$ latter nodes, which compute the parity coefficients. As a result, one round of this process (going from node $k + 1$ to node n) is typically not enough to ensure a good fault tolerance, and thus the encoding often necessitates several rounds.

3.5 Results

Encoding Times Analysis. One of the main advantages of distributing the erasure code redundancy generation across several nodes is that the required encoding times can be potentially reduced, providing more efficient ways of archiving replicated data. In this section we report performance results of an implementation of a (16,11)-RapidRAID [8] code (Section 3.2) which achieves significantly shorter encoding times in a testbed of 50 HP ThinClient computers, as compared to an efficient implementation of a (16,11) classical Cauchy Reed-Solomon erasure code.

Note however that the encoding speedup of RapidRAID is obtained at the expense of involving a total of $n = 16$ nodes in the encoding process, instead of the single node involved in classic encoding process. Thus, it is important to measure the encoding throughput of a classic erasure code involving the same number of nodes, i.e., when $n = 16$ classic encoding process are executed in parallel. Besides that, in practical scenarios storage nodes might be executing other tasks concurrently along with data archival processes, which might cause some nodes to experience overload or network congestions, which in turn might affect the coding times. Hence measuring the encoding times of both strategies (decentralized encoding vs. classic encoding) when some of the n nodes or networks are overloaded is another interesting aspect to study. In the experiments reported next, such bottlenecks are emulated in our testbed by reducing the network bandwidth of some nodes from 1GBps to 500MBps, and adding to these nodes a 100ms network latency (with a deviation of up to ±10ms).



(a) Encoding a single object.

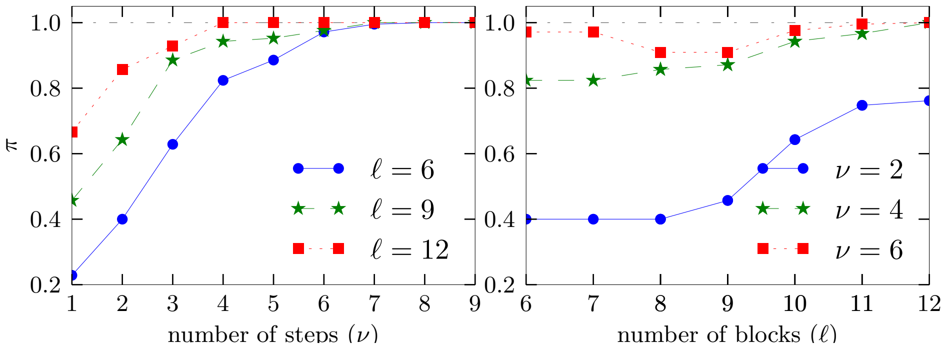
(b) Encoding 16 objects concurrently.

Fig. 2. Average time required to encode 16 concurrent objects using a (16,11) Cauchy Reed-Solomon code and a (16,11) RapidRaid code implemented using 8-bit finite-field operations. Nodes have 500Mbps connections with a latency of 100ms±10ms. Error bars depict the standard deviation value.

In Figure 2 we depict the encoding times of the (16,11) RapidRAID implementation (RR8) and the (16,11) Cauchy Reed Solomon Code (CEC). In Figure 2a we show the encoding times for a single data object and different number of congested nodes. In this case a single data object is encoded in a totally idle system. We see how when there are no congested nodes the RapidRAID implementations have of the order of 90% shorter coding times as compared to the classic erasure code implementation. Distributing the network and computing load of the encoding process across 16 different nodes reduces the encoding time significantly. In Figure 2b we depict the per-object encoding times obtained by executing 16 concurrent classic encoding processes and 16 concurrent RapidRAID encoding processes on a group of 16 nodes. According to further experiments on EC2 (not reported here, but details can be found in [8]), the two RapidRAID implementations achieve a reduction of the overall coding time by up to 20% when there are no congested nodes.

We further observe from Figure 2 that the coding times of RapidRAID codes have a quasi-linear behavior when the number of congested nodes increases. However, in the case of classic erasure codes, we observe that even a single congested node has major impact on the coding times. In general, these results show how RapidRAID codes significantly boost the performance of the encoding process over congested networks.

Fault Tolerance Analysis. As was illustrated in Example 1, RapidRAID codes offer a high fault tolerance. Some numerical analysis in [8] show how for short codes RapidRAID are MDS codes when $k \geq n-3$, and that for practical values of k out of this range RapidRAID codes offer a fault tolerance comparable to that of MDS codes. One of the reasons for this high fault tolerance is that the encoding process is distributed across a large number of storage nodes (across n nodes), which due to the random nature of the encoding process, reduces the chances of introducing linear dependencies within the codeword symbols. However, in the case of the systematic erasure code presented in Section 3.3, the encoding process is distributed across a smaller set $m = n - k$ nodes, and then the chances to introduce linear dependencies within the codeword symbols increase.



(a) Results for (10,6) code

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

Fig. 3. 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.

In this latter case it is then important to evaluate the fault tolerance of the obtained code for different code parameters. We divide the fault tolerance analysis in two experiments, one aiming at evaluating the effects of the number of encoding steps ν, and another one at the effects of the collocated replicas ℓ.

In Figure 3a we show the fault tolerance π of the code (proportion of linearly independent k-subsets) 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. 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 Figure 3b, we display the fault tolerance as a function of the number of blocks stored within the m coding nodes ℓ. For each code we also present the results for three different values of ν, which aim at showing the fault tolerance (i) when 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.

Network Traffic Analysis. Finally, we report the network traffic required to encode a specific data object with the novel decentralized erasure codes presented in Sections 3.2 and 3.3. Recall that in classic erasure codes the single encoding node downloads k data fragments, encodes them, and finally uploads $m-1$ parity blocks (where $m = n - k$). Each encoding process requires then a total of $n-1$ block transmissions. This is the same amount of transmissions required by RapidRAID codes (Section 3.3) where the n nodes involved in the encoding process transfer a total of $n-1$ temporal blocks among them. However, in the case of the non-systematic code presented in Section 3.2, the number of block transmissions is exactly one less than the number of encoding steps ν, i.e., $\nu-1$.

To evaluate this encoding traffic in Figure 4 we depict a comparison between a classic coding process, denoted by RS, and a decentralized systematic erasure code that achieves the MDS property (maximum fault tolerance), denoted by DE. In both cases we measure the encoding traffic required by both codes when $\ell = k$ and $\ell = 2k$, and in the case of DE, using the minimum value of ν required to achieve the MDS property. We show the comparison for three different code parameters. For the (6,3) code there are traffic savings only when the $m = 3$ coding nodes originally stored 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 ($\ell = 2k$).

4 Encoding Data During the Insertion Process

In the previous section we presented two distributed erasure codes that allow to efficiently archive an object $\mathbf{o}$ that is originally replicated. In this section we will present a technique to directly store the object $\mathbf{o}$ in an encoded format without first (temporarily) creating replicas at all. For instance, if multimedia content is being stored it may as well be stored directly in erasure coded format, unlike data that is used for analytics, and is archived only after the processing is completed.

First we will introduce the concept of *locally repairable codes*, and then we will show how such codes can be exploited to encode data *on-the-fly* while being introduced in the system in order to improve data insertion throughput.

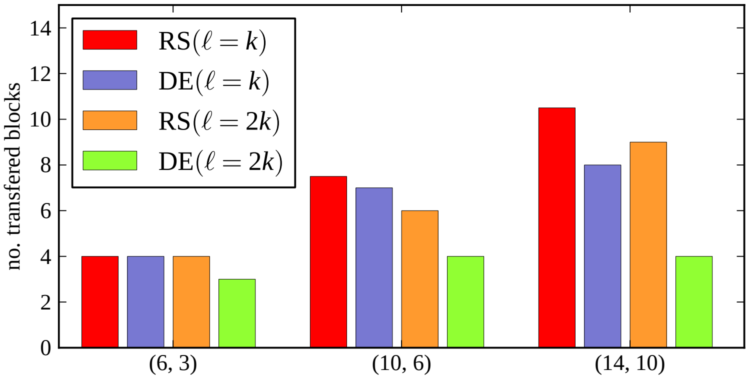


Fig. 4. Comparison of the number of transferred blocks during the encoding of a classical Reed-Solomon code (RS) and the decentralized coding (DE) for two different replica collocation values: $\ell = k$ and $\ell = 2k$. All DE codes are MDS codes optimized to minimize the number of coding steps ν.

4.1 Locally Repairable Erasure Codes

In classic erasure codes described in Section 2, the generation of each codeword symbol $c_i \in \mathbf{c}$ requires the access to the whole original data vector $\mathbf{o}$. When a storage node fails, repairing one missing codeword symbol $c_i \in \mathbf{c}$ requires to access k different codeword symbols and reconstruct $\mathbf{o}$, which entails a high I/O cost (accessing k different storage disks across the network). In contrast, locally repairable erasure codes allow to repair particular codeword symbols c_i by retrieving only d symbols, for small values of d, $d < k$, which can be as small as $d = 2$ [5]. Reducing the number of nodes needed to carry our a repair simplifies the repair mechanism, requires less I/O operations, and reduces the repair traffic with respect to classical erasure codes for a wide range of failure patterns, and can also speed-up the repair process.

Example 3. Let us present a simple locally repairable erasure code, specifically a (7,3)-code with the following generator matrix:

$$G^T = \begin{pmatrix} 1\,0\,0\,1\,1\,0\,1 \\ 0\,1\,0\,1\,0\,1\,1 \\ 0\,0\,1\,0\,1\,1\,1 \end{pmatrix} .$$

This code takes an object $\mathbf{o} = (o_1, o_2, o_3)$, $o_i \in \mathbb{F}_{2^q}$, and generates a codeword $\mathbf{c} = (c_1, \ldots, c_7)$ that contains the three original symbols (i.e., systematic symbols) plus all their possible xor combinations:

$$\begin{array}{lll} c_1 = o_1; & c_4 = o_1 + o_2; & c_7 = o_1 + o_2 + o_3; \\ c_2 = o_2; & c_5 = o_1 + o_3; & \\ c_3 = o_3; & c_6 = o_2 + o_3. & \end{array}$$

It is easy to see that it is possible to reconstruct the original object by downloading a minimum of $k = 3$ redundant fragments, e.g., c_5, c_6 and c_7, although not all k-subsets hold that property –i.e., it is a non-MDS code. Additionally, each redundant fragment can be generated by xoring two other redundant fragments in three different ways:

$$\begin{array}{lllll}
c_1 = c_2 + c_4; & c_2 = c_5 + o_7; & c_4 = c_3 + c_7; & c_6 = c_1 + c_7; & c_7 = c_3 + c_4; \\
c_1 = c_3 + c_5; & c_3 = c_4 + c_7; & c_4 = c_5 + c_6; & c_6 = c_2 + c_3; & \\
c_1 = c_6 + c_7; & c_3 = c_1 + o_5; & c_5 = c_1 + c_3; & c_6 = c_4 + c_5; & \\
c_2 = c_1 + c_4; & c_3 = c_2 + c_6; & c_5 = c_2 + c_7; & c_7 = c_1 + c_6; & \\
c_2 = c_3 + o_6; & c_4 = c_1 + c_2; & c_5 = c_4 + c_6; & c_7 = c_2 + c_5. &
\end{array}$$

We can represent the locally repairable property in terms of seven different repair groups $R = \{r_1, \ldots, r_7\}$, where:

$$\begin{array}{llll}
r_1 = \{c_1, c_2, c_4\}; & r_2 = \{c_1, c_3, c_5\}; & r_3 = \{c_1, c_6, c_7\}; & r_4 = \{c_2, c_3, c_6\}; \\
r_5 = \{c_2, c_5, c_7\}; & r_6 = \{c_3, c_4, c_7\}; & r_7 = \{c_4, c_5, c_6\}. &
\end{array}$$

Each codeword symbol $c_i \in r_j$ can be repaired/generated by summing the other symbols in r_j, $c_i = \sum_{c_k \in r_j \setminus \{c_i\}} c_k$.

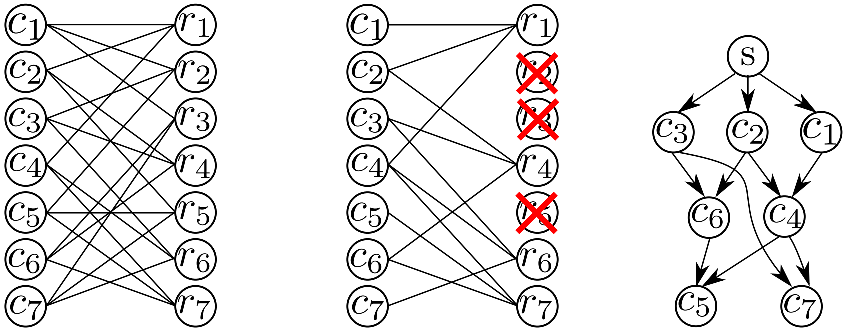
Locally Repairable Code. We can generically define a locally repairable code as an undirected bipartite graph $\mathcal{G} = (C \cup R, E)$, where the set of vertices U represents the set with all the codeword symbols $C = \{c_i : c_i \in \mathbf{c}\}$, the set of vertices R represented all the repair groups $R = \{r_i, \ldots, r_r\}$, and $c_i \in r_j \iff (c_i, r_j) \in E$. Then, for any $c_i \in r_j$, the locally repairable property guarantees that

$$c_i = \sum_{c_k \in r_j \setminus \{c_i\}} \psi_k c_k,$$

for predetermined ψ_k values, $\psi_k \in \mathbb{F}_{2^q}$.

4.2 In-Network Redundancy Generation

As noted in the introduction, pipelining can be trivially used to create replication based redundancy. To do so, a source node sends the data to be stored to a first storage node, which stores it and simultaneously forwards it to a second storage node, and so on. However, as discussed in Section 2, erasure coding schemes do not allow to generate data redundancy of newly inserted data "on-the-fly", and often this process is carried out off-line in a batch process [2]. In this section we show how locally repairable codes are potentially amenable to be used in the on-line redundancy generation of newly inserted data. We will refer to this redundancy generation approach as an *in-network* redundancy generation process.



(a) Bipartite graph of the locally repairable code.

(b) Purging R to obtain the redundancy generation graph.

(c) Redundancy generation graph.

Fig. 5. In-network redundancy generation using the (7,3)-code from Example 3.

Example 4. Let us consider the same (7,3) locally repairable code described in Example 3. In Figure 5c we show an in-network redundancy generation example for this code. The source node receives an object $\mathbf{o} = (o_1, o_2, o_3)$ and uploads each of these fragments to nodes from 1 to 3, which respectively store $c_1 = o_1$, $c_2 = o_2$ and $c_3 = o_3$. Then, nodes 1 and 2 send their respective fragments, c_1 and c_2, to node 4, which computes and stores $c_4 = c_1 + c_2$. The rest of the nodes compute the fragments $c_6 = c_2 + c_3$, $c_5 = c_4 + c_6$ and $c_7 = c_3 + c_4$ in a similar manner.

Note that the creation of c_5 and c_7 depends on the previous generation of c_4 and c_6. Although at a first glance it might seem that symbols c_5 and c_7 can be created only some time in the future after the generation of c_4 and c_6, practical implementations can overcome this restriction by allowing nodes 4 and 6 to start forwarding the first generated bytes of c_4 and c_6 to nodes 5 and 7 in a streamlined way, similarly to the pipelined copy used in replication. By doing it, blocks c_4 to c_7 can be generated quasi-simultaneously once nodes 1 to 3 received their blocks from the source node. However, in those situations where data from the source is being generated/received while it is being inserted, fragments o_1 to o_3 will be sequentially uploaded, delaying the generation of symbols c_4 to c_7, and thus, lengthening the overall redundancy generation process.

This example allow us to show how the local repairability groups can be exploited to generate the in-network redundancy generation tree depicted in Figure 5c. However, obtaining such a tree for any arbitrary locally repairable code is not trivial. One possible way to obtain it is to consider the bipartite graph representation of the (7,3) locally repairable code, which we depict in Figure 5a. This graph contains all the codeword symbols c_i in the left-hand side of the graph, and all the repair groups r_i in the right-hand side of the graph. Then, since the original object $\mathbf{o}$ contains $k = 3$ symbols, the source node has to mandatorily upload three symbols to three different nodes (left-hand side vertices). The rest of the $n - k$ codeword symbols will be generated using $n - k$ different repair groups. It means that we can purge all except $n-k$ repair groups from the right-hand side of the graph. In our example, in Figure 5b we show how we remove $|R| - (n - k)$ repair groups from the bipartite graph, which gives us the basic topology of the in-network redundancy generation tree from Figure 5c. Although in this case any combination of $|R| - (n - k)$ repair groups can be removed to obtain a valid redundancy generation tree, determining the repair groups to remove can be more complicated in asymmetric and unbalanced locally repairable codes, where the number of symbols per repair group is not constant, and not all symbols appear in the same number of repair groups. More details on these issues can be found in [7].

4.3 Insertion Times

Inserting a data object $\mathbf{o} = (o_1, \ldots, o_k)$, $o_i \in \mathbb{F}_{2^q}$, using an in-network redundancy generation process requires two different steps: (i) a source node initially generates and uploads k different codewords symbols to k different nodes, and (ii) these k nodes forward the symbols to the remaining $n-k$ nodes in a streamlined manner, allowing them to generate the remaining $n-k$ codeword symbols. Then, the overall time T required to encode and store an object $\mathbf{o}$ is bounded by $T \geq T_S + T_{net}$, where T_S is the time the source needs to upload k symbols, and T_{net} is the time the in-network redundancy generation needs to generate the rest of the $n - k$ symbols. It is important to note that, neglecting encoding times, T_S is proportional to the amount of data uploaded by the source, which is $q \times k$ bits, and T_{net} is proportional to the maximum number of successors a node in the in-network redundancy generation tree has, which is two for the nodes 2, 3 and 4 in the example depicted in Figure 5c. Then, in general, if this maximum number of successors is smaller than $n - k$, the overall insertion time T will be shorter than the insertion time of a classic erasure encoding process.

This simplistic analysis shows that an in-network redundancy generation can indeed increase the storage throughput of the classic erasure code insertion. However, this throughput can be further exacerbated when the source node and the set of storage nodes have additional (mismatched) temporal constraints on resources availability. For example, in datacenters storage nodes might be used for computation processes which require efficient access to local disks. Since inserting encoded data consumes large amounts of local disk I/O, system administrators might want to avoid to store while nodes are executing I/O intensive tasks – e.g., Mapreduce tasks.

To model this temporal constraints we will use the binary variable $a(i,t) \in \{0,1\}$, which represents whether or not node i is available for sending/receiving data during time step t where each time step is of a duration of τ seconds. If we assume that the time step duration τ is equal to the time required to send/receive a symbol $c_i \in \mathbb{F}_{2^q}$, then, when all nodes are available, the source node can insert one full object $\mathbf{o}$ per time step. However, when some of the n nodes are not available, the source will have to wait until these nodes become available again, reducing the insertion throughput. To overcome this problem, the source node can group different objects $(\mathbf{o}_1, \mathbf{o}_2, ...)$ and store them altogether using the same in-network redundancy generation tree. By doing so the source will be able

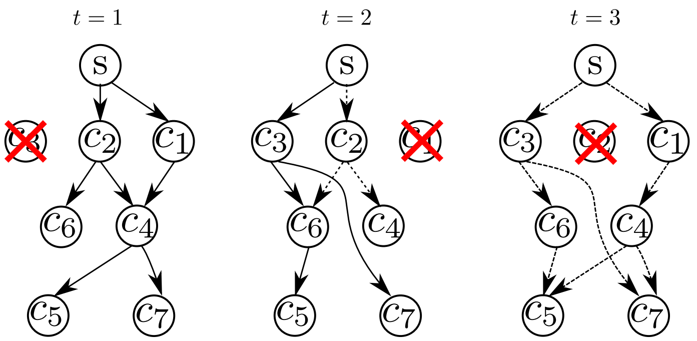


Fig. 6. The source S inserts two objects $\mathbf{o}_1$ (depicted with continuous arrows) and $\mathbf{o}_2$ (depicted with dashed arrows) in three time steps. At each time step there is one node that is not available for sending/receiving data.

to start inserting symbols from $\mathbf{o}_2$ while it is still waiting for completing the insertion of $\mathbf{o}_1$.

Example 4. Assume that a source node aims at storing two objects $\mathbf{o}_1$ and $\mathbf{o}_2$ using the in-network redundancy tree depicted in Figure 5c. Due to temporal constraints, node 3 is unavailable at $t = 1$, node 1 is unavailable at $t = 2$, and node 2 is unavailable at $t = 3$, as depicted in Figure 6. Under these circumstances, during $t = 1$ the source can send the symbols c_1 and c_2 of $\mathbf{o}_1$ (depicted with continuous line arrows) to nodes 1 and 2, and then trigger the generation of c_4. However, c_5, c_6 and c_7 cannot be generated because they depend on c_3, whose corresponding node is unavailable. During $t = 2$, the source can then send the missing symbol c_3 of $\mathbf{o}_1$, and finally trigger the generation of c_5, c_6 and c_7. However, during this very same step, the source can also start sending the symbol c_2 of $\mathbf{o}_2$ (depicted with dashed line arrows). All remaining symbols of $\mathbf{o}_2$ can be finally generated during the third time step $t = 3$.

This example shows a specific in-network redundancy generation scheduling that allows to insert two different objects in there time steps. In this case the in-network redundancy generation tree is the same for the three steps, however, in some cases it might also be possible to increase the insertion throughput by using different redundancy generation trees at each time step. Unfortunately, due to the vast scheduling possibilities that arise when nodes have temporal availability constraints, determining an optimal schedule given an arbitrary locally repairable code becomes a complex problem [7], even when the node availabilities $a(i, t)$ are known beforehand.

Instead of finding an optimal in-network redundancy generation schedule, different heuristics can be used to maximize the insertion throughput. In particular, we identify two questions that an heuristic scheduling algorithm can answer at each time step t: (i) Which set of nodes must the source send data to? (ii) Which repair groups are used to generate in-network redundancy? For the first question we propose two different answers:

1. **Random (Rnd)**: The source selects k nodes at random.
2. **Minimum Data (Min)**: The source selects the k nodes that had received less codeword symbols.

And for the second question:

1. **Minimum Data (Dta)**: The scheduling algorithm tries to generate in-network redundancy in those nodes that had received less codeword symbols.
2. **Maximum Flow (Flw):** The scheduling algorithm tries to generate in-network redundancy in those nodes closer to the root of the redundancy generation tree, trying to maximize the redundancy generation flow.

Combining these four different approaches we can obtain four different redundancy generation scheduling algorithms, namely *RndFlw*, *RndDta*, *MinFlw* and *MinDta*. In the next section we will compare the insertion throughput of each of these.

4.4 Evaluation

In this section we evaluate the insertion performance of an in-network redundancy generation algorithm and we compare it with the naive erasure coding insertion process. The erasure code used in both the cases is the locally repairable code described in Example 3.

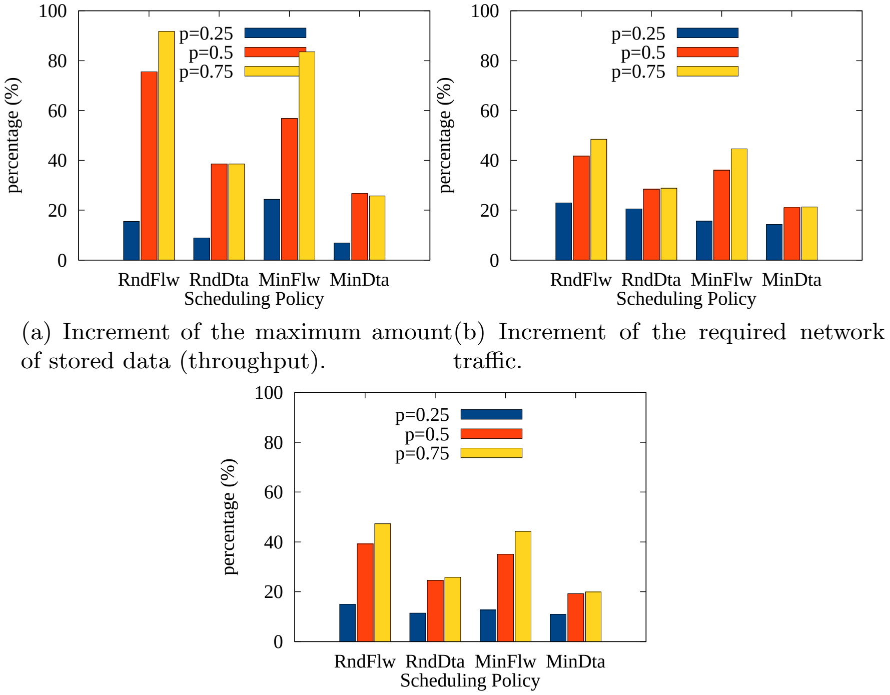
The reported results are obtained using a discrete-time simulator, where node availabilities $a(i,t)$ are modeled using real availability traces collected from a Google data center [5]. These traces contain the normalized I/O load of more than 12,000 servers monitored for a period of one month. Specifically, we consider that a server is available to upload/download data when its I/O load is under the p-percentile load. We consider three different percentiles, $p = 0.25, 0.5, 0.75$, giving us three different node availability constraints.

In Figure 7a we show the increment of the data insertion throughput achieved by the in-network redundancy generation process. This increment is higher when nodes have high availability, and thus it is more likely that all the nodes in a repair group are available.

In Figure 7b we show the increment on the required network traffic of the in-network redundancy generation strategy. The total traffic required for in-network redundancy generation can be up to 50% higher than what is needed by the classic insertion process. The network traffic increment is higher for high node availabilities, since the opportunities to use the local repairability property increase since generating a symbol using in-network redundancy requires twice as much traffic as generating it from the source.

This increase in traffic is approximately the same or even less than the boost in storage throughput, even for low availability scenarios. Thus the in-network redundancy generation scales well by achieving a better utilization of the available ephemeral network resources than the classical storage process.

[5] Publicly available at: http://code.google.com/p/googleclusterdata/



(a) Increment of the maximum amount of stored data (throughput).

(b) Increment of the required network traffic.

(c) Reduction of the data uploaded by the source.

Fig. 7. Performance of the in-network redundancy generation compared to the classic erasure encoding data insertion process.

Finally, in Figure 7c we show the reduction of data uploaded by the source. In the classic insertion approach the source needs to upload $7/3 \simeq 2.33$ times the size of the actual data to be stored; $4/7 \simeq 57\%$ of this data is redundant. The in-network redundancy generation process allows to reduce the amount of data uploaded by the source. In this figure we can see how in the best case (*RndFlw* policy) our approach reduces the source's load by 40% (out of a possible 57%), yielding an 40-60% increment on the overall insertion throughput.

5 Conclusions

Storage technologies have continuously been undergoing a transformation for decades. While erasure coding techniques have long been explored in the context of large-scale systems (among other kind of storage environments), the explosive growth of storage needs in recent years has accelerated the research on and adoption of storage centric erasure codes. The issue of repairability of erasure codes [4, 6] has particularly been a key avenue of investigation in the last years. While repairability remains an open issue under study, the vast amount of existing literature makes it a relatively mature topic. We believe that erasure codes can be tailor-made to achieve other desirable properties. In particular, in this paper we summarize some of our early results on how to optimize the throughput of creating erasure coded data, either from existing replicas within an NDSS, or when data is freshly being introduced in the NDSS. This line of work is in its nascence, and the current works are a first few steps in what we hope will be a new direction of research on storage codes.

Acknowledgement

This paper accompanies the invited talk by Anwitaman Datta titled "*On data insertion and migration in erasure-coding based large-scale storage systems*" at ICDCIT 2013 conference. 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. S. Acedański, S. Deb, M. Médard, and R. Koetter. How good is random linear coding based distributed networked storage. In *Workshop on Network Coding, Theory, and Applications (NetCod)*, 2005.
2. Apache.org. HDFS-RAID. `http://wiki.apache.org/hadoop/HDFS-RAID`.
3. Cleversafe. `http://www.cleversafe.com`.
4. A. Datta and F. Oggier. An overview of codes tailor-made for networked distributed data storage. *CoRR*, abs/1109.2317, 2011.
5. F. Oggier and A. Datta. Self-repairing homomorphic codes for distributed storage systems. In *The 30th IEEE Intl. Conference on Computer Communications (INFOCOM)*, 2011.

6. F. Oggier and A. Datta. Coding techniques for repairability in networked distributed storage systems. *http://sands.sce.ntu.edu.sg/CodingForNetworkedStorage/pdf/longsurvey.pdf*, 2012.
7. L. Pamies-Juarez, A. Datta, and F. Oggier. In-network redundancy generation for opportunistic speedup of backup. *CoRR*, abs/1111.4533, 2011.
8. L. Pamies-Juarez, A. Datta, and F. Oggier. RapidRAID: Pipelined Erasure Codes for Fast Data Archival in Distributed Storage Systems. *CoRR*, abs/1207.6744, 2012.
9. L. Pamies-Juarez, F. Oggier, and A. Datta. Decentralized erasure coding for efficient data archival in distributed storage systems. In *14th International Conference on Distributed Computing and Networking*, 2013.