A

Reducing Repair Traffic in P2P Backup Systems: Exact Regenerating Codes on Hierarchical Codes

ZHEN HUANG, National University of Defense Technology
Ernst Biersack, Eurecom
Yuxing Peng, National University of Defense Technology

Peer to peer backup systems store data on "unreliable" peers that can leave the system at any moment. In this case, the only way to assure durability of the data is to add redundancy using either replication or erasure codes. Erasure codes are able to provide the same reliability as replication requiring much less storage space. Erasure coding breaks the data into blocks that are encoded and then stored on different nodes. However, when storage nodes permanently abandon the system, new redundant blocks must be created, which is referred to as repair. For "classical" erasure codes, generating a new block requires the transmission of k blocks over the network, resulting in a high repair traffic. Recently, two new classes of erasure codes, Regenerating Codes and Hierarchical Codes, have been proposed that significantly reduce the repair traffic: Regenerating Codes reduce the amount of data uploaded by each peer involved in the repair, while Hierarchical Codes reduce the number of nodes participating in the repair. In this paper we propose to combine these two codes to devise a new class of erasure codes called ER-Hierarchical Codes that combine the advantages of both.

Categories and Subject Descriptors: E.5 [**FILES**]: Backup/recovery

General Terms: Storage, P2P backup systems, Maintenance, durability, reliability

Additional Key Words and Phrases: Regenerating Codes, Hierarchical Codes, repair degree

1. INTRODUCTION

1.1. Motivation

In P2P backup systems, redundancy is the basic technique to assure durability of the data. What kind of redundancy scheme to use has been extensively discussed in the literature. Many papers focus on the comparison between replication and erasure codes [Weatherspoon and Kubiatowicz 2002; Lin et al. 2004; Rodrigues and Liskov 2005]. The simplest way to add redundancy is replication, which produces multiple copies (replicas) that are stored on different nodes. A more complex redundancy scheme is **erasure codes** that divide the data into k blocks, encode them into h parity blocks and store the $(k + h)$ blocks on different nodes. The original data can be reconstructed by taking a subset of the $(k + h)$ blocks. Compared to replication, erasure codes are more space efficient, but also require more maintenance traffic to reconstruct a lost block of data [Lin et al. 2004; Rodrigues and Liskov 2005]. In this paper, we present a new code called ER-Hierarchical Code that significantly *reduces the maintenance traffic* as compared to other erasure codes.

Author's address: Z. Huang and Y. Peng are with the National Laboratory of Parallel and Distributed Processing, Department of Computer, National University of Defense Technology, Changsha, Hunan, P. R. China 410073.
E. Biersack is with Eurecom, Sophia Antipolis, France.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1.2. Data Maintenance

A backup system must execute three types of operations.

— Data insertion, to encode and store new data in the backup system
— Data reconstruction, to recover the original data and
— Data maintenance, to repair the loss of data that were stored on nodes in the backup system that have left the system.

Data maintenance introduces network traffic between the nodes that participate in the repair of a lost block. As pointed out previously [Rodrigues and Liskov 2005] the available network bandwidth of these nodes can become the bottleneck resource that limits the total amount of data that can be stored safely in the system.

The most "straightforward" way of repairing a lost block is as follows: We need a set $\mathcal{R}$ of k nodes that store k blocks of the group of $k+h$ blocks the lost block was part of and we need an additional node, referred to as **newcomer**, who will store the block that "replaces" the lost block. The nodes in $\mathcal{R}$ transmit one block each to the newcomer. The newcomer uses the k blocks to first reconstruct the k original data blocks to then produce a new block, which will be bit-by-bit identical with the lost block. Such a repair requires to transfer an amount of data that is k *times as big as the lost data*.

To see if we can do better, we need to understand the factors that determine the total amount of data transmitted to repair a lost block, namely (i) the number of nodes involved in a repair, which is referred to as the **repair degree** d and (ii) the amount of data transmitted by each node that participates in the repair, which is referred to as **repair block size** S_{RB}. The purpose of this paper is to introduce new codes that allow to reduce the repair traffic by reducing either one or both these two factors. As we will see, there are ways to repair a lost block without reconstructing first the original data.

1.3. Codes

Erasure codes can be classified into **optimal codes** and **near-optimal codes**. Optimal codes have the property that the original data can be reconstructed using *any subset* of k blocks from the set of $(k+h)$ blocks. Optimal codes are maximum distance separable codes, or **MDS** codes for short.

Reed-Solomon Codes are a well known class for an optimal codes. Consider a Galois Field GF(2^q), where the elements of such a field can be expressed by q-bit words. Let us denote as o_i and p_i the i^{th} original block and the i^{th} parity block. A linear code can be built using the following linear operations in GF(2^q):

$$p_i = \begin{cases} o_i & i \leq k \\ \sum_{j=1}^{k} c_{i,j} o_j & k < i \leq k+h,\ c_{i,j} \in GF(2^q) \end{cases} \quad (1)$$

We see from Eq. (1) that every parity block that is not identical to an original block is a linear combination of *all* the original blocks. The code is optimal since *any subset* of k parity blocks allows to reconstruct the k original blocks.

Near-optimal codes, on the hand (i) either need $(1+\epsilon)k$ blocks to recovery the original data (where $\epsilon > 0$) or (ii) can reconstruct the original data only for some subsets of k blocks but not for any arbitrary subset. Examples for near-optimal codes are Tornado codes, Low-density parity check codes [Richardson and Urbanke 2008] or hierarchical codes [Duminuco and Biersack 2009]. Near-optimal codes bring big reductions in terms of communication and computational overhead [Mitzenmacher 2004; Plank 2005; Duminuco and Biersack 2009].

Hierarchical Codes [Duminuco 2009] are near-optimal codes since not any set of k blocks allows to reconstruct the original data. Hierarchical Codes that have the same storage efficiency as Reed-Solomon Codes but require in many cases a much lower

repair degree d, which allows to reduce the total amount of traffic for repairing a lost block and also allows to complete the download from the d nodes faster [Duminuco and Biersack 2009].

Regenerating Codes are optimal codes. However, as compared to other codes, they drastically reduce the repair block size, while their repair degree and block size that is at least as big as for Reed-Solomon Codes. In the distributed storage systems, Regenerating Codes apply the concept of network coding to erasure codes. Regenerating Codes are a few years old [Dimakis et al. 2010; Rashmi et al. 2009; Duminuco and Biersack 2009; Dimakis et al. 2010] and have been continuously improved since. Particularly interesting for backup storage systems are **Exact Regenerating Codes** [Wu and Dimakis 2009; Rashmi et al. 2009; Suh and Ramchandran 2010] that not only reduce the computational complexity of Regenerating Codes but are also able, to regenerate back exactly, i.e bit by bit, the lost block.

Given that Hierarchical Codes reduce the repair degree and that Exact Regenerating Codes reduce the repair block size, we propose to "combine" these two codes to even further reduce the total repair traffic. We devise a new coding scheme called **ER-Hierarchical Codes** where the ideas of Exact Regenerating Codes are applied to the *minimum group*[1] of an Hierarchical Code. We take the coding scheme proposed for Exact Regenerating Codes [Rashmi et al. 2009] to demonstrate how to construct ER-Hierarchical Codes that provide the same reliability as Hierarchical Codes and significantly reduce the repair traffic by reducing both, the *repair degree* and the *repair block size*.

The next two sections present background material on codes (section 2) and propose metrics that allow to compare the different redundancy schemes (section 3). The following two sections present new codes: In section 4 we propose Exact Hierarchical Codes, and in section 5 we demonstrate how to build ER-Hierarchical Codes and state some propositions about ER-Hierarchical Codes. In section 6 we evaluate ER-Hierarchical Codes in comparison to other erasure codes through analysis and simulation. We conclude the paper in section 7 with a summary and an outlook.

2. BACKGROUND ON REGENERATING CODES AND HIERARCHICAL CODES

A key issue in P2P backup systems is data maintenance: storage nodes may leave the system and the data stored on these nodes need to be repaired to maintain data availability and durability. Data repair introduces network traffic between the surviving nodes and the available network bandwidth of these nodes can become the bottleneck resource [Rodrigues and Liskov 2005] that limits the total amount of data that can be stored safely in the system.

In response to this problem, Regenerating Codes and Hierarchical Codes have been proposed. We will show how to combine them to obtain ER-Hierarchical Codes, which achieve an even higher reduction in repair traffic. The symbols used throughout the paper are summarized in Table I.

The focus of this paper is erasure codes. We consider an original file of size of S_{file} that is encoded into $n = k+h$ blocks that are stored on n different nodes. The data from k out of the n nodes allow to *reconstruct* the original file. Every node stores a block of size S_{SB} that consists of α fragments, with $\alpha \in \mathbb{N}$. When a block is lost because of a node failure, d storage nodes can be used to *repair* the lost block: Each of the d nodes uploads β fragments to the newcomer, with $\beta \in \mathbb{N}$. While for Reed Solomon codes α and β both take the value of one, α and β can take values larger than one for Regenerating Codes.

[1] *Minimum group* is defined later in section 5.3.

Table I. Variables Used

Symbol	Description
k	number of original blocks
h	number of parity blocks
n	total number of blocks, $n = k + h$
d	repair degree for one repair process
2^q	Galois field size
N_l	number of concurrent losses
R	reliability for the coding scheme
α	amount of fragments stored on one node
β	amount of fragments uploaded by one node
S_{file}	size of the original file
S_{fraq}	fragment size
S_{SB}	storage block size in one node
S_{RB}	repair block size uploaded by one node
V_{fi}	file insert traffic for the system
V_{rp}	total repair traffic for one block
V_{rc}	reconstruction traffic
$V_{rp}^{(ec)}$	total repair traffic for one block using code ec

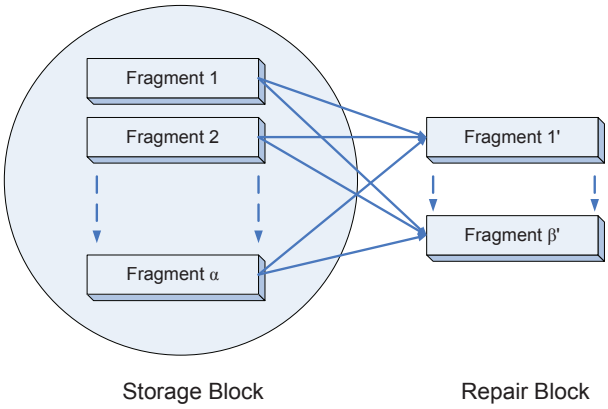


Fig. 1. Regenerating Codes: Storage block used to generate repair block

2.1. Regenerating Codes for distributed storage system

We will now explain the differences between Regenerating Codes and the traditional erasure codes, present the different types of Regenerating Codes and illustrate how we derive ER-Hierarchical Codes. For a survey of Regenerating Codes see [Dimakis et al. 2010].

2.1.1. Repair block size for Regenerating Codes. As depicted in Fig. 1, Regenerating codes differ from Reed-Solomon Codes in the way the block stored at a node is produced and also by the fact that this block is not simply transferred to the newcomer to repair the loss of another block. The block stored at a node, referred to as **storage block** consists of α fragments of size S_{frag} each. At *repair*, the node who stores a block first linearly combines the α fragments of its storage block to produce β parity fragments, which make up a **repair block** that is uploaded to the newcomer. In this case, the repair block size is $S_{RB} = \beta S_{frag}$ and the storage block size is $S_{SB} = \alpha S_{frag}$.

To illustrate the idea of Regenerating codes, we present in Fig. 2 a small example with four nodes $N_1, ..., N_4$. The original data are broken into four fragments and $\alpha = 2$ original fragments are stored on nodes N_1 and N_2. The nodes N_3, N_4 store two parity

fragments each. The parameters of the code are $k = 2$, $h = 2$ and $\alpha = 2$. If any of the nodes fails, the lost data can be reconstructed in two different ways.

— "Classical Reed-Solomon type repair:" Since the code is an MDS code, the newcomer node who will hold the newly generated storage block downloads from any two of the three remaining nodes all the data stored on these two nodes. In this case the repair block is identical to the storage block and the repair traffic (volume of data transmitted) is $k \cdot S_{SB} = 2 \cdot S_{SB}$.
However, there is a second way to repair a lost storage block that requires less network traffic:
— "Regenerating-code type repair:" The newcomer node downloads from each of the three remaining nodes a repair block of the size of *one* storage fragment. Assume N_1 has failed; The newcomer downloads fragment o_3 from N_2, fragment $o_1 - o_3$ from N_3, and fragment $o_2 + o_3$ from N_4 to reconstruct the two fragments o_1, o_2. Sometimes, the newcomer node does not download simply one of the fragments stored on a remaining node, but a fragment that is a *linear combination of the two fragments* stored on that node: Lets assume that node N_4 has failed. In this case the newcomer node downloads o_2 from N_1, o_3 from N_2 and $(o_1 - o_3) - (o_2 - o_4)$ from N_3.
In both cases, the total repair traffic will be $3 \cdot S_{frag} = 1.5 \cdot S_{SB}$, which constitutes a reduction in repair traffic of 25%. While the reduction of the repair traffic in this example is quite modest, there are other cases where Regenerating Codes reduce the repair traffic by one order of magnitude or more [Dimakis et al. 2010].

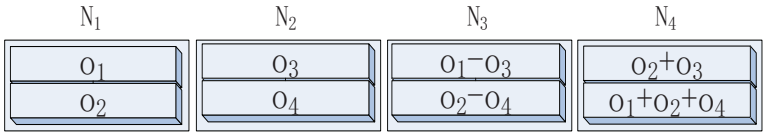


Fig. 2. Example for Regenerating Code

2.1.2. Repair types for Regenerating Codes. Regenerating Codes allow for two different repair types called **functional repair** and **exact repair**. In both cases, we can choose *any* $d > k$ alive blocks to repair the lost block. [Dimakis et al. 2010] maps the repair process onto an **Information Flow Graph** and casts the problem of reducing the repair traffic into finding the minimum cut of the Information Flow. In doing so, the repaired data just need to maintain the MDS property, i.e. any k out of $k + h$ blocks allow to reconstruct the original data, without requiring the lost block to be replaced by exactly the same, in the sense of being identical in every bit. This kind of repair can be regarded as **functional repair**. However, *functional repair* has several disadvantages [Suh and Ramchandran 2010]: (i) it can not efficiently support the applications that frequently access the original data. (ii) its computational complexity of encoding-and-decoding is very high.

To address these issues, several authors [Wu and Dimakis 2009; Rashmi et al. 2009; Suh and Ramchandran 2010] have proposed **Exact Regenerating Codes** that can exactly regenerate back a lost block, which can be considered as an **exact repair**. In our work, we first propose the *exact repair* for Hierarchical Codes and then transform exact Hierarchical Codes into ER-Hierarchical Codes by applying the Regenerating Codes to the *minimum group* of Hierarchical Codes.

2.1.3. Optimal Regenerating Codes. There are two optimal instances of Regenerating Codes: **Minimum-Bandwidth Regenerating (MBR)** and **Minimum-Storage Regenerating (MSR)** [Dimakis et al. 2010].

MBR uses more storage space on every node than Reed-Solomon Codes, but minimizes the repair traffic for repairing one block, which is $S_{SB}^{(MBR)} = dS_{RB}^{(MBR)} = \frac{2dS_{file}}{2kd-k^2+k}$. In the process to repair one block, d nodes upload S_{RB}, the total repair block size $V_{rp}^{(MBR)} = dS_{RB}^{(MBR)}$. Then we can obtain $V_{rp}^{(MBR)} = S_{SB}^{(MBR)}$. The repair degree is d, with $d > k$.

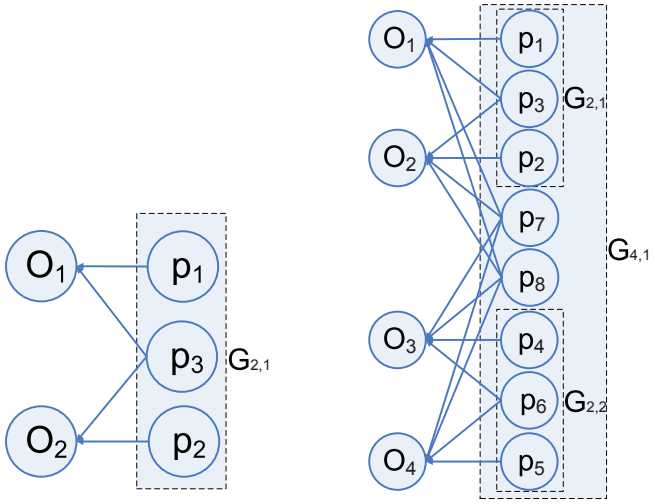
In this paper, we focus on **MSR** that can reduce the *repair traffic* without requiring more *storage* than Reed-Solomon Codes, and we apply one coding scheme proposed in [Rashmi et al. 2009] called **EMSR**. The repair degree of *MSR* is $d = k + \alpha - 1$ and the repair traffic is $V_{rp}^{(MSR)} = \frac{dS_{file}}{k(d-k+1)}$. As α increases, d increases but $V_{rp}^{(MSR)}$ will decrease. For $\alpha = 2$ we get $d = k + 1$ and $V_{rp}^{(MSR)} = \frac{S_{file}(k+1)}{2k}$. This means that for large k, the repair traffic will be about half of the repair traffic of Reed-Solomon Codes. See Fig. 2 for an example of a MSR code.

2.2. Hierarchical Codes

Hierarchical Codes have been explicitly designed for storage systems and aim to reduce the repair traffic [Duminuco and Biersack 2009].

2.2.1. Hierarchical Codes with functional repair. Hierarchical codes reduce the repair degree d, which in most cases will be much smaller than the number of original blocks k. This is possible since most parity blocks are a linear combination of *only a few* original blocks. However, the price to pay is that not all parity blocks are "equally useful" in reparing a lost block and that not all subsets of k blocks allow to reconstruct the original blocks. Hierarchical codes partition the original blocks into **groups** and define an iterative generation rule on how to combine the original blocks of one or more groups to produce the parity blocks. When the parities are generated for the first time, a hierarchical code-(k, h) is a *systematic* code with $k + h$ parity blocks, where k parity blocks are *original* blocks and h *parity* blocks are linear combinations of the original blocks. A general instance of a Hierarchical Code can be generated through its *code graph* built according to the following **generation rule**:

(1) Choose two parameters k_0 and h_0 and build a code-(k_0, h_0) using Eq. (1) with the coefficients $c_{i,j}$ *chosen randomly* in $GF(2^q)$. If we set k_0=2 and h_0=1 we obtain the *code graph* in Fig. 3(a). The generated blocks constitute a group denoted as $G_{d_0,1}$, where d_0=k_0 is the degree used to generate the blocks and is called the **combination degree**. In Fig. 3(a), d_0=2.
(2) Choose two parameters g_1 and h_1. Replicate the group structure $G_{d_0,1}$ g_1 times to obtain g_1 groups denoted as $G_{d_0,1} \dots G_{d_0,g_1}$. Then add other h_1 encoded blocks, obtained combining (with random coefficients) all the existing g_1k_0 original blocks. This corresponds to a combination degree d_1=g_1k_0=g_1d_0. If we set g_1=2 and h_1=2 we obtain the *code graph* in Fig. 3(b). All the blocks constitute a group denoted as $G_{d_1,1}$, which corresponds to a hierarchical code-(d_1, H_1), where H_1=g_1h_0+h_1. The example in Fig. 3(b) is a hierarchical code-$(4, 4)$.
(3) The previous step can be repeated multiple times, adding levels to the code. In step l, choose two parameters g_l and h_l. Replicate g_l times the structure of the group $G_{d_{l-1},1}$. Then add other h_l encoded blocks, obtained combining all the existing original blocks, which corresponds to a degree d_l=g_ld_{l-1}. All the blocks constitute a group denoted as $G_{d_l,1}$, which corresponds to a hierarchical code-(d_l, H_l), where $H_l = g_lH_{l-1}$+h_l.



(a) HC-(2,1) (b) HC-(4,4)

Fig. 3. Code Graphs for Hierarchical Codes.

After L steps, we get the code HC-(d_L, H_L). From the iterative generation rule, we obtain the following recurrence relations that allow us to compute (d_L, H_L):

$$\begin{cases} d_l = g_l d_{l-1}, \forall l \in [1, L] \\ H_l = g_l H_{l-1} + h_l, \forall l \in [1, L], H_0 = h_0 \end{cases} \tag{2}$$

Solving Eq. (2), we can obtain k and h:

$$k = d_L = (\prod_{l=1}^{L} g_l) d_0 \tag{3}$$

$$\begin{aligned} h &= H_L \\ &= g_L H_{L-1} + h_L \\ &= g_L(g_{L-1} H_{L-2} + h_{L-1}) + h_L \\ &= g_L g_{L-1} H_{L-2} + g_L h_{L-1} + h_L \\ &= \ldots \\ &\overset{(H_0 = h_0)}{=} \sum_{l=0}^{L-1} (\prod_{i=l+1}^{L} g_i) h_l + h_L \end{aligned} \tag{4}$$

Given the HC is constructed as outlined above, [Duminuco and Biersack 2009] have established the following Rules for reconstructing the original blocks and for reparing a lost block. Since a graph consists of nodes and edges and since each node will store a single block, we will use in the rest of the paper the terms of block and node interchangeably.

Reconstruction Condition [Duminuco and Biersack 2009]:

PROPOSITION 1. *Consider P^k, a set of k nodes in the code graph of a hierarchical code-(k,h). **If** the nodes in P^k are chosen fulfilling the following condition:*

$$|G_{d,i} \cap P^k| \leq d \ \ \forall G_{d,i} \text{ belonging to the code} \tag{5}$$

which means that in P^k there can be a maximum of d nodes chosen from any group $G_{d,i}$, ***Then*** *the nodes in P^k are sufficient to reconstruct the original nodes.*

Repair Condition [Duminuco and Biersack 2009]:

PROPOSITION 2. *Consider an* Information Flow Graph *of a hierarchical code at time step t. Consider a node p that must be repaired at time step t. Denote as $G(p)$ the hierarchy of groups that contains p and as $R(p)$ the set of nodes in P_{t-1} that have been combined to repair p.* ***If*** *$\forall t$ and $\forall p$, $R(p)$ fulfills the following conditions:*

$$|G_{d,i} \cap R(p)| \leq d \ \ \forall G_{d,i} \text{ belonging to the code} \tag{6}$$

and

$$\exists\, G_{d,i} \in G(p) : R(p) \subseteq G_{d,i}, |R(p)| = d \tag{7}$$

Then *the code does not degrade, i.e. preserves the properties of the code graph expressed in Proposition 1.*

Low repair degree: Let us illustrate the use of the repair condition to repair the loss of a single block for the HC-(4,4) in Fig. 3(b)

— Repair of block p_3 in *Group* $G_{2,1}$: Select $d_0 = 2$ blocks p_1 and p_2 and combine them using randomly chosen coefficients[2] from $GF(2^q)$. Independent of the size of k and h, the repair degree of any block in *Group* $G_{2,i}$ is 2 as long as the other two blocks in $G_{2,i}$ are available.
— Repair of block p_8 in *Group* $G_{4,1}$: Select $d_1 = 4$ blocks, such as p_1, p_2, p_4, p_5, or p_1, p_2, p_7, p_5 or p_2, p_7, p_4, p_5 and combine them using randomly chosen coefficients from $GF(2^q)$. In this case, condition (6) tells us not to take more than 2 blocks from $G_{2,1}$ or $G_{2,2}$.

Thanks to Proposition 2, a block stored on the failed node can be repaired by d_l blocks in *Group* $G_{d_l,i}$. The selection of blocks from a *Group* depends on the state of the storage nodes in the system. From the Eq. (2), we see that $d_0 < d_1 < ... < d_L = k$. So the repair degree d is typically much *lower* than k and only the h_L parity blocks in the highest level need $d_L = k$ blocks to repair. Conversely, because of the iterative code construction, many blocks are in the lowest level: Let N_0 denote the total number of the blocks in level 0. We can calculate N_0 as:

$$\begin{aligned} N_0 &= (\prod_{l=1}^{L} g_l)(d_0 + h_0) \\ &\overset{(Eq.\ (3))}{=} \frac{k}{d_0}(d_0 + h_0) \\ &\overset{(h_0 \geq 1)}{\geq} k + \frac{k}{d_0} \end{aligned} \tag{8}$$

If we take an existing peer-to-peer backup system such as Wuala [Wuala 2010], where k is set to a large value such as $k = 100$ and we design a hierarchical code using Eq. (2) with $d_0 = 5$ and $k = d_L = 100$, there are at least 120 blocks at level 0 (see Eq. (8)). In the best case, we need only $d_0 = 5$ blocks to repair a lost block, instead of $k = 100$ as for classical erasure codes.

[2]Since we do functional repair, the newly generated block that replaces p_3 is not bit-by-bit identical with p_3.

3. METRICS TO EVALUATE REDUNDANCY SCHEMES

Many metrics are possible to compare the different codes. We will select metrics that allow to evaluate and compare the different codes with respect to our design objectives.

3.1. Storage Space and Network Traffic

A node that stores one block of backup data will consume $S_{SB} = \alpha S_{frag}$ of space. With the rapid increase in disk capacity [HardDriver 2010], the storage capacity may not be the bottleneck of P2P backup systems if we use erasure codes for redundancy. But this does not mean that we can use all the available disk space of the nodes to store backup data. The volume of repair traffic is a function of the amount of data stored at each node and the nodes only support a limited amount of network traffic for repairing lost blocks.

In P2P backup systems, network traffic results from data transfers that occur at the following occasions:

— **Insert:** In this phase, the file owner encodes the file to add redundancy. The encoded data is distributed across n nodes. Every node stores one block of size of S_{SB}, which implies that the insert traffic is $V_{fi} = nS_{SB}$.
— **Reconstruction:** When the owner loses the original file he contacts the backup system to recover the original data. In this case, the user needs to download enough data from k alive storage peers to rebuild the original file. By downloading k parity blocks stored in the backup system, user can reconstruct the original file, resulting in a reconstruction traffic of $V_{rc} = kS_{SB}$.
— **Maintenance:** If d alive storage peers are involved in the repair process and every peer uploads S_{RB} data, then the total repair traffic is $V_{rp} = dS_{RB}$. Compared to the insert operation, which occurs only once at the beginning, the of repair lost blocks can occur frequently because of churn.

3.2. Repair degree

The repair degree d is an important parameter for the repair process. It is important to keep d small: (i) The larger the repair degree, the more peers are involved in the repair process and the longer time it takes to complete one repair. As discussed in [Pamies-Juarez et al. 2010], when the repair degree d is much larger than k, the repair time increases quickly. (ii) In many cases, a smaller repair degree also allows to support more concurrent node failures while still being able to repair the lost blocks. Conversely, when the repair degree is $d = k + h - 1$, the failure of *more than one node*, will make it impossible to *directly repair* the lost blocks. Instead, one first needs to *reconstruct the original blocks*. Depending on the code that is used, reconstruction can result in a higher network traffic, namely $V_{rc} = kS_{SB}$ instead of $V_{rp} = dS_{RB}$. (iii) When the repair degree increases, the computational cost of encoding and decoding increases quickly [Duminuco and Biersack 2009]. (iv) The larger the number of nodes involved in the repair process, the higher the chance that one or more nodes involved fail during the repair, which further increases the complexity of the repair process and the total repair time [Zhang et al. 2010; Chun et al. 2008].

Different codes impose different constraints on the repair degree:

— $d = k$ is the repair degree for *Reed-Solomon Codes*.
— $d \leq k$ is the repair degree for *Hierarchical Codes*. In fact, for most of the block repairs, the repair degree will be much smaller than k.
— $d \in [k + 1, k + h - 1]$ is the repair degree for *Regenerating Codes*. As mentioned in section 2.1, we will use EMSR, which does exact repair with a minimum repair degree of $d = k + 1$.

3.3. Reliability

There are many ways to lose data, such as disk errors, node crashes, or nodes leaving the system. However, loss of some blocks must not impact the capability of the backup system to reconstruct the original data. Therefore, a very high reliability is the design objective for all the backup systems. As stated in [Duminuco and Biersack 2009], one of the most important threats for *reliability* of a P2P backup system is *concurrent block losses*. When the number of concurrently lost blocks increases, the system may lose the ability to reconstruct the original data using the remaining alive blocks. So *reliability* is defined as $1-P(failure|N_l)$, where $P(failure|N_l)$ is the probability of data loss (*failure*) given that N_l concurrent losses will occur.

4. EXACT HIERARCHICAL CODES

Hierarchical Codes as defined in [Duminuco and Biersack 2009] do *functional repair*. We now show how to do *exact repair* for Hierarchical Codes, which means that lost blocks will be regenerated by their exact replica. As a consequence, blocks *do not change over time* and we just need to focus on the properties of the code graph *when it was first generated*. We call the resulting code **Exact Hierarchical Code**.

In the following we explain how to do repair for Exact Hierarchical Codes and state the advantages that *exact repair* brings to Hierarchical Codes.

Exact Repair for Hierarchical Codes: The repair condition for exact repair is similar to the one for functional repair, which was stated in Proposition 2. To do exact repair of a lost block p in *Group* $G_{d,i}$, we first need to find a group $G_{d_l,j}$ with $G_{d,i} \subseteq G_{d_l,j}$ for which a set P^{d_l} of d_l parity blocks is available that meets the following generic selection rule for ***Exact Hierarchical Codes***:

PROPOSITION 3. ***EHC-type repair:*** *$\forall l \in [0, L]$, consider P^{d_l}, a set of d_l nodes in the* Group $G_{d_l,i}$.

If *the nodes in P^{d_l} are chosen fulfilling the following condition:*

$$\forall G_{d_m,j} \subseteq G_{d_l,i}, \ \ |G_{d_m,j} \cap P^{d_l}| \leq d_m \tag{9}$$

which means that in P^{d_l} there can be a maximum of d_m nodes chosen from any group $G_{d_m,j}$,

Then *the nodes in P^{d_l} are sufficient to repair any one node in $G_{d_l,i}$.*

PROOF. According to the generation rule for Hierarchical Codes, the *Group* $G_{d_l,i}$ can be mapped onto a subgraph of HC-(d_l, H_l), whose parity blocks are linear combinations of d_l original blocks. Thanks to the Proposition 1, the nodes in P^{d_l} are sufficient to reconstruct all the original blocks in HC-(d_l, H_l). It means that all the original blocks in $G_{d_l,i}$ can be reconstructed. Then any one block in $G_{d_l,i}$ can be exactly repaired by linearly combing the d_l original blocks. So Proposition 3 holds. □

Given we have a set P^{d_l} of parity blocks that were selected according to Proposition 3, we can now repair the lost block p performing *exact repair* in two steps:

— Use the parity blocks in P^{d_l} to *decode* the d_l original blocks of $G_{d_l,j}$: The coefficient vectors of the d_l parity blocks build an invertible matrix M that is used to decode the d_l original blocks.
— Then linearly combine the original blocks to *exactly* repair the failed block p. This way, the problem of repairing a parity block p is cast into that of repairing all the original blocks p connects to.

Exact repair brings two advantages for the applications: (i) it maintains the *original blocks* in the system over time, which can improve the read efficiency and get rid of

the computational cost for decoding. (ii) it allows the data reconstruction of different blocks to be distributed and parallelized. For instance in Fig. 3(b), we can use $G_{2,1}$ to reconstruct the original blocks o_1 and o_2, and concurrently use $G_{2,2}$ to reconstruct the original blocks o_3 and o_4. This can not only save the reconstruction time but also distribute the computational overhead.

5. ER-HIERARCHICAL CODES

To explain how ER-Hierarchical Codes work, we proceed in two steps: We first show through an example how a Hierarchical Code can be *transformed* into an ER-Hierarchical Code and we prove two proposition for ER-Hierarchical Codes concerning code construction and blocks repair. To describe the different coding operations in a precise yet succinct way, we use notion from linear algebra such as matrices and vectors

$O_{k\times 1}$ Vector of original blocks
$P_{n\times 1}$ Vector of parity blocks
$C_{n\times k}$ Coefficient matrix.

If we have a HC-(d_L, H_L) with $k = d_L, h = H_L, n = k + h$, then we can express the generation of the parities as a linear combination of the original blocks:

$$\begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nk} \end{pmatrix} \begin{pmatrix} o_1 \\ \vdots \\ o_k \end{pmatrix} \tag{10}$$

We use the same formalism for ER-Hierarchical Codes to emphasize the similarities and differences with respect to Hierarchical Codes. We will introduce ER-Hierarchical Codes by describing how to *convert* a Hierarchical code into an ER-Hierarchical Code and then show how to perform repair.

5.1. Deriving ER-Hierarchical Codes from Hierarchical Codes

Assume we have a HC-(d_L, H_L) with a vector O of original blocks, a coefficient matrix C, and the resulting parity block vector P, with $P = CO$.

ER-Hierarchical Codes combine concepts of Hierarchical Codes and of Regenerating Codes, namely that most parity blocks are linear combinations of only a small subset of all original blocks (c.f. Fig. 3) and that a storage block consists of α fragments, while a repair block has only β, fragments, with , $\beta < \alpha$ (c.f. Fig. 1).

If we want to transform a Hierarchical Code into an ER-Hierarchical code, we first need to partition each original block $o_i \in O$ into α fragments. For the rest of this article, we use in our examples $\alpha = 2$. All the propositions we prove are valid for any value of α. In practice, however, one should use small values for α to keep the repair degree low, which is defined as $d_r = d_l + \alpha - 1$. If we set α to a large value, we may obtain a repair degree d_r, with $d_r > d_l + h_l$, which means that there are not enough nodes in a group to use regenerating codes for repair. Conversely, for $\alpha = 2$, we can do the repair using Regenerating Codes in a small group, which means a low repair degree and a less repair traffic.

To describe the coding and decoding operations for ERHC, the vectors of blocks used in HC need to be replaced by matrices of fragments. We also need an additional coefficient matrix:

$O_{k\times 2}$ Matrix of original *fragments*
$P_{n\times 2}$ Matrix of parity *fragments*

$C_{n\times k}$ Coefficient matrix
$U_{n\times k}$ Coefficient matrix. The coefficients u_{ij} take arbitrary values in $GF(2^q)$.

The transformation of original fragments into parity fragments is defined in Eq. (11) and Eq. (12). We note the close resemblance between Eq. (10) and Eq. (11). In Eq. (11) the first parity fragment of each parity block is generated using only the first parity fragment of each original block. The coefficients c_{ij} ($i \in \{1, ..., n\}$, $j \in \{1, ..., k\}$) used in Eq. (11) are identical to the ones used in Eq. (10) for HC-(d_L, H_L). On the other hand, the second parity fragment of each parity block is generated using both, the first and second fragment of an original block (see Eq. (12)).

$$\begin{pmatrix} p_{11} \\ \vdots \\ p_{n1} \end{pmatrix} = \begin{pmatrix} c_{11} & \dots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nk} \end{pmatrix} \begin{pmatrix} o_{11} \\ \vdots \\ o_{k1} \end{pmatrix} \tag{11}$$

$$\begin{pmatrix} p_{12} \\ \vdots \\ p_{n2} \end{pmatrix} = \begin{pmatrix} u_{11} & \dots & u_{1k} \\ \vdots & \ddots & \vdots \\ u_{n1} & \dots & u_{nk} \end{pmatrix} \begin{pmatrix} o_{11} \\ \vdots \\ o_{k1} \end{pmatrix} + \begin{pmatrix} c_{11} & \dots & c_{1k} \\ \vdots & \ddots & \vdots \\ c_{n1} & \dots & c_{nk} \end{pmatrix} \begin{pmatrix} o_{12} \\ \vdots \\ o_{k2} \end{pmatrix} \tag{12}$$

If we define $P_i = (p_{1,i}, ..., p_{n,i})^T$ and $O_i = (o_{1,i}, ..., o_{n,i})^T$, ($i \in \{1, 2\}$), where the superscript T denotes the transpose of a vector, we can express Eq. (11) and Eq. (12) as Eq. (13), which is a variant of ***EMSR*** proposed in [Rashmi et al. 2009]. The area of regenerating codes is rapidly evolving and frequently new codes are developed. The code we use is an EMSR code, however other MSR codes such as Product-Matrix Construction [Rashmi et al. 2010] could also be used in combination with HC.

$$\begin{cases} P_1 = CO_1 \\ P_2 = UO_1 + CO_2 \end{cases} \tag{13}$$

5.2. An Example

In Fig. 4 we see an ERHC-(4,4,2) code that was derived from the HC-(4,4) code of Fig. 3(b) The original blocks o_i in Fig. 4 are split into two fragments $(o_{i1}, o_{i2}), i \in \{1...4\}$, and the parity blocks p_j are replaced by fragments pairs $(p_{j1}, p_{j2}), j \in \{1...8\}$).

If the *parity blocks* in the HC-(4,4) depicted in Fig. 3(b) can be computed as

$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ c_{31} & c_{32} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & c_{63} & c_{64} \\ c_{71} & c_{72} & c_{73} & c_{74} \\ c_{81} & c_{82} & c_{83} & c_{84} \end{pmatrix} \begin{pmatrix} o_1 \\ o_2 \\ o_3 \\ o_4 \end{pmatrix} \tag{14}$$

then the *parity fragments* of the ERHC-(4,4,2) (c.f. Fig. 4) can be computed as shown in Eq. (15) and Eq. (16). When the parity fragments are *first generated*, the coefficients of the matrix U in Eq. (16) can take arbitrary values. However, whenever a *repair* of a

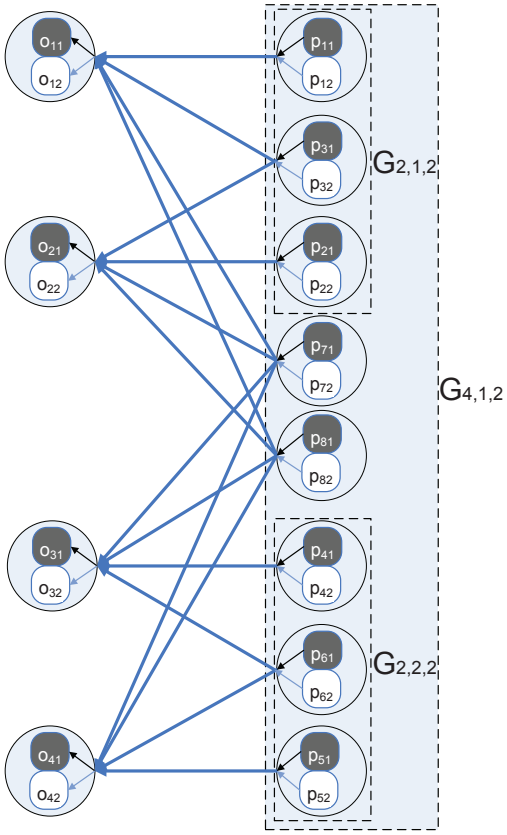


Fig. 4. ER-Hierarchical Codes-(4,4,2)

parity fragment is performed the matrix U must be updated (see appendix A.4)

$$\begin{pmatrix} p_{11} \\ p_{21} \\ p_{31} \\ p_{41} \\ p_{51} \\ p_{61} \\ p_{71} \\ p_{81} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ c_{31} & c_{32} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & c_{63} & c_{64} \\ c_{71} & c_{72} & c_{73} & c_{74} \\ c_{81} & c_{82} & c_{83} & c_{84} \end{pmatrix} \begin{pmatrix} o_{11} \\ o_{21} \\ o_{31} \\ o_{41} \end{pmatrix} \tag{15}$$

and

$$\begin{pmatrix} p_{12} \\ p_{22} \\ p_{32} \\ p_{42} \\ p_{52} \\ p_{62} \\ p_{72} \\ p_{82} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ u_{31} & u_{32} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & u_{63} & u_{64} \\ u_{71} & u_{72} & u_{73} & u_{74} \\ u_{81} & u_{82} & u_{83} & u_{84} \end{pmatrix} \begin{pmatrix} o_{11} \\ o_{21} \\ o_{31} \\ o_{41} \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ c_{31} & c_{32} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & c_{63} & c_{64} \\ c_{71} & c_{72} & c_{73} & c_{74} \\ c_{81} & c_{82} & c_{83} & c_{84} \end{pmatrix} \begin{pmatrix} o_{12} \\ o_{22} \\ o_{32} \\ o_{42} \end{pmatrix} \tag{16}$$

Having defined how to generate the parity fragments of an ERHC, we can now focus on reconstruction and repair. Again, we will insist on the similarities and differences between HC and ERHC for each of these operations.

5.2.1. Reconstruction the original data:. Suppose original data of HC-(4,4) in Fig. 3(b) is reconstructed using the parity nodes p_1, p_3, p_6 and p_7. In ERHC-(4,4), we also choose these 4 nodes and reconstruct the original data using the parity fragments (p_{11}, p_{12}), (p_{31}, p_{32}), (p_{61}, p_{62}) and (p_{71}, p_{72}) as follows:

— Select the first parity fragments p_{11}, p_{31}, p_{61} and p_{71} to decode the first original fragments o_{11}, o_{21}, o_{31} and o_{61}.

$$\begin{pmatrix} o_{11} \\ o_{21} \\ o_{31} \\ o_{41} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ c_{31} & c_{32} & 0 & 0 \\ 0 & 0 & c_{63} & c_{64} \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix}^{-1} \begin{pmatrix} p_{11} \\ p_{31} \\ p_{61} \\ p_{71} \end{pmatrix} \tag{17}$$

— Subtract the first original fragments o_{11}, o_{21}, o_{31} and o_{41} from p_{12}, p_{32}, p_{62} and p_{72}. To obtain o_{12}, o_{22}, o_{32} and o_{42}, invert the sub-matrix of C as in Eq. (17).

$$\begin{pmatrix} o_{12} \\ o_{22} \\ o_{32} \\ o_{42} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ c_{31} & c_{32} & 0 & 0 \\ 0 & 0 & c_{63} & c_{64} \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix}^{-1} \begin{pmatrix} p_{12} \\ p_{32} \\ p_{62} - u_{63}o_{31} - u_{64}o_{41} \\ p_{72} - u_{71}o_{11} - u_{72}o_{21} - u_{73}o_{31} - u_{74}o_{41} \end{pmatrix} \tag{18}$$

5.2.2. Repair of a lost block:. Suppose the parity block p_8 is lost. In HC-(4,4) of Fig. 3(b) p_8 can be repaired using p_1, p_2, p_4 and p_7 that fulfill the condition (9). When (p_{81}, p_{82}) is lost in ERHC-(4,4,2), we must do the repair differently (Fig. 5 depicts the repair process), which is a *RC-type repair*.

(1) Select $d_r = 5$ storage blocks from *Group* $G_{4,1,2}$, such as (p_{11}, p_{12}), (p_{21}, p_{22}), (p_{41}, p_{42}), (p_{51}, p_{52}) and (p_{71}, p_{72}). Let $I = \{1, 2, 4, 5, 7\}$.
(2) Each of the five nodes that stores a pair of parity fragments $(p_{i1}, p_{i2}), i \in I$, does a *linear combination* of its two parity fragments using a coefficient $\nu_i, i \in I$, whose value was computed by the newcomer and then communicated to node i to compute then repair fragment λ_i, with $\lambda_i = (\nu_i, 1)(p_{i1}, p_{i2})^T = (\nu_i \underline{c_i} + \underline{u_i}, \underline{c_i})(O_1, O_2)^T$, where the $\underline{c_i}$ and $\underline{u_i}$ are respectively the i^{th} row vector of matrices C and U in Eq. (13). The repair fragment will be transmitted to the newcomer.
(3) The newcomer, does a *linear combination* of these five repair fragments λ_i to obtain two new parity fragments (p_{81}, p'_{82}) that replace the lost fragments (p_{81}, p_{82}). Note that p_{81} will be replaced by a fragment that is bit-by-bit identical with p_{81}. However, p_{82} *can not* be replaced by a fragment that is bit-by-bit identical. ERHC therefore does the next best thing, namely, **approximately exact repair**. Approximately exact repair means that, when RC-type repair is performed, the first parity fragment stored in a node can be *exactly repaired*, while the second parity fragment can only be *functionally repaired*, i.e. is not bit-by-bit identical. The reason for this is quite technical and we refer to the literature for the exact details [Shah et al. 2010]. At a high level, the explanation is as follows: The problem of exact repair regenerating codes can be expressed as an interference-alignment problem. In Fig. 2, for instance, to repair the lost fragments (o_1, o_2) of node N_1, we use o_3, $(o_1 - o_3)$ and $(o_2 + o_3)$, in which o_3 is the interference to get the information o_1, o_2. To do exact repair, requires satisfying multiple interference-alignment conditions simultaneously, which turns out to be over-constrained for $\alpha < k - 2$. However, for the code presented in Fig. 2 we have $2 = \alpha \geq k - 2 = 0$, which means we can do exact repair of both fragments. The details of how to compute (p_{81}, p'_{82}) are given in appendix A.

We can now evaluate the reduction in repair traffic due to ERHC: Suppose the block size for HC-(4,4) is 1 MB. Block p_8 is repaired by uploading *four* parity blocks of 1 MB each, which makes a total repair traffic of 4 MB. For ERHC-(4,4,2) on the other hand, there are *five* nodes that upload one repair fragment of size 0.5 MB each, which makes a total repair traffic of only 2.5 MB.

Now that we have illustrated the operation of ERHC through an example, we must formalize the process of parity generation and parity repair for ERHC.

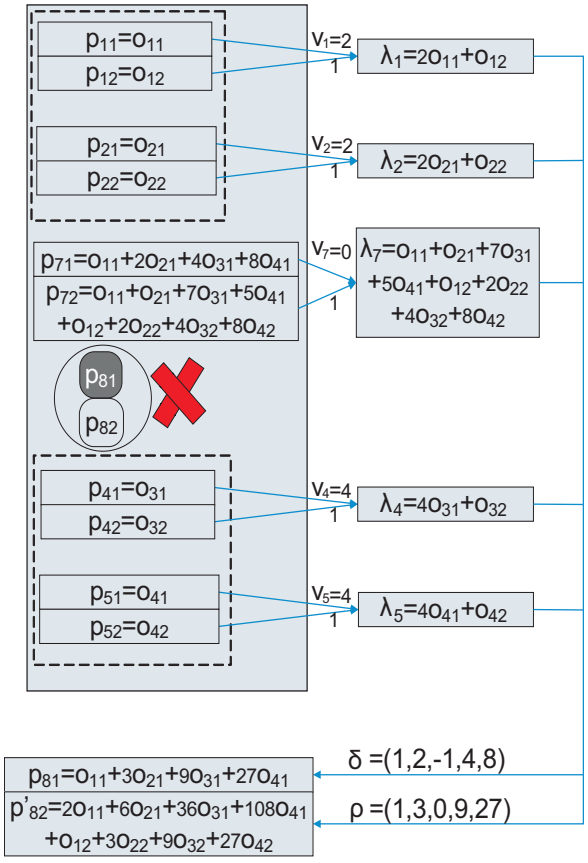


Fig. 5. Repair of (p_{81}, p_{82}) in ERHC-(4,4,2)

5.3. Formalizing ER-Hierarchical Codes

Assume we have a HC-(d_L, h_L) that we need to convert into an ERHC-(d_L, h_L, α). We first split each original block into α fragments. The following proposition defines a *conversion rule* that must be respected when using the original fragments of one original block to produce the α parity fragments that make up a parity block.

Definition 5.1. Assume we have a graph such as the one depicted in Fig. 4 that expresses the dependencies between parity fragments and original fragments. We refer to the nodes on the left as **original nodes** and on the nodes on the right as **parity nodes**. If the original fragment j of node o is used to produce the parity fragment j

of node p we say that there is a **connection** from node p to node o, with respect to fragment j, denoted as $path_{p\to o}^{(j)}$.

PROPOSITION 4. *Consider any parity node p in ER-Hierarchical Codes with the connections to a set $O(p)$ of original nodes.*
***If** the parity node p fulfills the following condition:*

$$\forall o \in O(p) : \exists \bigcup_{j=1}^{\alpha} \{path_{p\to o}^{(j)}\} \text{ such that } \forall i,k \overset{i\neq k}{\in} \{1,...,\alpha\} : \ path_{p\to o}^{(i)} \neq path_{p\to o}^{(k)} \tag{19}$$

which means that there exist α disjoint paths from the parity fragments stored on node p to the original fragments stored on node o.
***Then** Proposition 3, which defines how to select the nodes used to repair a lost node, is valid for ER-Hierarchical Codes.*

In order to prove Proposition 4, we need the following lemma about hierarchical codes, which has been proven in [Duminuco and Biersack 2009].

LEMMA 1. *At any time t, any possible selection of k nodes P_t^k is sufficient to reconstruct the original fragments only if the* disjoint paths *condition is provided at time step $t = 1$ (by the* code graph*) and the repair degree is $d \geq k$.*

PROOF. Suppose d_l nodes in $G_{d_l,i,\alpha}$ are selected that fulfill condition (9), which means d_l *disjoint paths* from these nodes to the source nodes can be found according to Lemma 1. Because of condition (19), the disjoint paths from the fragments in these nodes to the original fragments can be found. This means, we can find $d_l \cdot \alpha$ disjoint paths from the fragments in P^{d_l} to the original fragments. Then these $d_l \cdot \alpha$ fragments are sufficient to reconstruct the original fragments (See the proposition 1 of [Dimakis et al. 2010]). So Proposition 4 holds. □

Using Proposition 4, we can make a statement about the reliability of ERHC:

PROPOSITION 5. *The* reliability *of ER-Hierarchical Codes is the same as Hierarchical Codes.*

PROOF. As mentioned in section 3, the probability $1 - P(failure|N_l)$ denotes the reliability of the code. When losses occur in ER-Hierarchical Codes, the same available nodes can be selected as Hierarchical Codes that fulfill condition (9). Thanks to the Proposition 3, which applies to $G_{d_L,1}$ and to Proposition 4, ER-Hierarchical Codes can reconstruct all the original nodes. So Proposition 5 holds. □

According to proposition 4, there is no degradation in *reliability* when using ER-Hierarchical Codes as compared to Hierarchical Codes. Repair of a lost parity node can be done according to Proposition 4 by selecting a set of nodes that fulfill the condition (9). Each of these nodes then uploads $\alpha = 2$ parity fragments to the newcomer node.

However, as we have shown in the example given in section 5.2.2 (c.f. Fig. 5), we can also repair a lost parity node by choosing a slightly larger set of nodes, each of which uploads only $\beta = 1$ fragment to the newcomer node. How to determine this set of nodes will be subject of proposition 6.

We first must define the notion of a minimum group.

Definition 5.2. Consider parity node p that belongs to groups $G_{d_{l_1},j,\alpha}, ..., G_{d_{l_s},j,\alpha}$. If $d_l = \min\{d_{l_1}, ..., d_{l_s}\}$, then $G_{d_l,j,\alpha}$ is the **minimum group** for node p.

We can now give a selection rule for nodes, contained in the minimum group of a lost node, that allows to repair the lost node using repair techniques from Regenerating Codes.

PROPOSITION 6. ***RC-type repair:*** *Consider a lost parity node located in the minimum group $G_{d_l,i,\alpha}$ of ERHC-(d_L, H_L, α). Let P^{d_r}, with $d_r = d_l + \alpha - 1$, be a set of d_r nodes in $G_{d_l,i,\alpha}$.*

If *the nodes in P^{d_r} fulfill the following condition:*

$$|G_{d_m,j,\alpha} \cap P^{d_r}| \leq d_m \quad \forall G_{d_m,j,\alpha} \subset G_{d_l,i,\alpha} \tag{20}$$

which means that P^{d_r} can contain a maximum of d_m nodes from any subset $G_{d_m,j,\alpha}$,

Then *if each node in P^{d_r} uploads β fragments, the nodes in P^{d_r} are sufficient to repair any lost node in $G_{d_l,i,\alpha}$.*

PROOF. The condition (20) implies that any d_l nodes fulfill the condition (9). In this case, any d_l nodes are linearly independent and each of them is a linear combination of d_l original nodes. This is the MDS property of Regenerating Codes for the minimum group. Then we can do repair using *Exact Regenerating Codes* by uploading β fragments from each parity node in P^{d_r}. After the repair, the new parity node is again a linear combination of the d_l original nodes, which means that the new parity node will remain part of the same minimum group. So Proposition 6 holds. □

5.4. Reconstruction and Repair

Reconstruction can be realized as the repair of the original fragments in the highest *Group*. Due to proposition 4, we can reconstruct the original fragments by selecting nodes that fulfill condition (9). The total traffic is $V_{rc} = d_L \cdot \alpha \cdot S_{frag} = k \cdot S_{SB} = S_{file}$, which is the same as for Reed-Solomon Codes. Also ER-Hierarchical Codes allow, as do Exact Hierarchical Codes, to distribute and parallelize the reconstruction of different blocks.

Repair: For repair, one can choose a *Group* of nodes using either one of the following two rules:

— (R1) **RC-type repair:** Select d_r nodes that fulfill condition (20) in the *minimum group* and repair by uploading β fragments from each of the d_r nodes to the newcomer. The details on how to generate the parity fragments by the newcomer were given in section 5.2.2 and in the appendix A.
— (R2) **EHC-type repair:** Select d_l nodes that fulfill condition (9) and are part of one *Group* $G_{d_l,i,\alpha}$ and upload all α fragments from each of the d_l nodes to the newcomer who will then generate α parity fragments.

Rule (R2) performs exact repair by reconstructing first all the original blocks corresponding to group $G_{d_l,i,\alpha}$, which are then used to produce the lost parity block. Since we have $d_r = d_l + \alpha - 1$, rule (R2) has a smaller repair degree than rule (R1); however, each of the d_l nodes needs to transmit a larger amount of data to the newcomer node.

In practice, which repair rule to use depends on which nodes of a group are currently available. For the analytical evaluation in section 6.2, we compute the repair traffic using rule (R1). For the experimental evaluation in section 6.3, we consider both rules and prefer rule (R1), which minimizes the repair traffic. If rule (R1) does not allow to repair the lost block we try rule (R2). For example, if we take the ERHC-(4,2,2) of Fig. 4, to repair any lost block in groups $G_{2,1,2}$ and $G_{2,2,2}$ we need to use rule (R2) since we do not have $d_r = 3$ blocks in the minimum group as required by rule (R1). For more details see also section 6.3.7 and Fig. 12.

6. EVALUATION

6.1. Scope of the Comparison

Many different types of regenerating codes are possible (see section 2.1.3). We carefully considered which type of regenerating code to use in order to both, reduce the repair bandwidth and also to allow for a meaningful comparison with existing codes such as Reed Solomon codes or Hierarchical Codes. We choose MSR and not MBR since MSR allows us to compare the repair bandwidth with other existing codes that all require the *same* amount of storage space. However, for sake of completeness we have added a comparison between ERHC and MBR in appendix B of the paper. For a cost analysis of different types codes including regenerating codes, replication, and Reed Solomon codes we refer the reader to [Pamies-Juarez and Biersack 2011]. We now evaluate and compare ER-Hierarchical Codes with Reed Solomon codes, Hierarchical codes, and Exact Regenerating Codes (EMSR) both, analytically and through simulation.

6.2. Analytical Evaluation

We compare the different codes using the metrics from section 3.

6.2.1. Storage. Because MSR just splits the block into α fragments stored on each node, the storage requirement of ER-Hierarchical Codes is equal to the storage requirement of Hierarchical Codes, namely $S_{SB} = \alpha S_{frag} = \frac{M}{k}$. It is the same storage consumption as for the traditional Reed-Solomon code.

6.2.2. Reliability. Thanks to the Proposition 5, there is no degradation in reliability as compared to Hierarchical Codes.

6.2.3. Traffic. $d^{(ERHC)}$ denotes the repair degree in ER-Hierarchical Codes. When one node fails, every node that participates in the repair will upload β fragments. So the repair traffic in ER-Hierarchical Codes is $V_{rp}^{(ERHC)} = d^{(ERHC)}\beta S_{frag}$. We want to compare ER-Hierarchical Codes with the other erasure codes. ***EM-ERHC*** denotes the ***EMSR*** on Hierarchical Codes. $IV_{rp}^{(ec)}$ denotes the reduction in repair traffic due to ER-Hierarchical Codes as compared to another code *ec*.

— **Comparison with Reed-Solomon Codes:** In Reed-Solomon Codes, one failed block will be repaired using $k = d_L$ nodes that each upload entire blocks, i.e. all the α fragments. So the repair traffic is $V_{rp}^{(RS)} = d_L \alpha S_{frag}$. Then we can obtain an improvement of

$$\begin{aligned} IV_{rp}^{(RS)} &= 1 - \frac{V_{rp}^{(ERHC)}}{V_{rp}^{(RS)}} \\ &\overset{Eq.\ (3)}{=} 1 - \frac{d^{(ERHC)}\beta}{(\prod_{i=1}^{L} g_i) d_0 \alpha} \end{aligned} \tag{21}$$

In the case of ***EM-ERHC***, $\alpha = 2\beta$. Suppose we use the ***Group*** in level l to do repair in ER-Hierarchical Codes. Then the repair traffic can be improved $IV_{rp}^{(RS)} = 1 - \frac{(1+d_l)}{2(\prod_{i=0}^{L} g_i) d_0}$.

— **Comparison with Exact Regenerating Codes:** From the repair process, we can see that the amount of data uploaded by the nodes that participate in a repair is the same as for both, for Exact Regenerating Codes and ER-Hierarchical Codes. However, the repair degree is different. In ER-Hierarchical Codes, the repair degree

is dynamic and depends on the state of the *Groups*.

$$IV_{rp}^{(ER)} = 1 - \frac{V_{rp}^{(ERHC)}}{V_{rp}^{(ER)}} = 1 - \frac{d^{(ERHC)}}{d^{(ER)}} \quad (22)$$

For Exact Regenerating Codes the repair degree is constant: $d^{(ER)} = d_L + 1$. Suppose ER-Hierarchical Codes use nodes that are part of a Group at level l to do the repair, then $d^{(ERHC)} = d_l + 1$. The repair traffic can be improved by $IV_{rp}^{(ER)} = 1 - \frac{d_l+1}{(\prod_{i=0}^{L} g_i)d_0+1}$.

— **Comparison with Hierarchical Codes:** In Hierarchical Codes, the exact repair degree depends on the nodes that are available to participate in the repair. Let $d^{(HC)}$ denote the repair degree for Hierarchical Codes. As for Reed-Solomon Codes, each participating nodes uploads an entire block to the newcomer.

$$IV_{rp}^{(HC)} = 1 - \frac{V_{rp}^{(ERHC)}}{V_{rp}^{(HC)}} = 1 - \frac{d^{(ERHC)}\beta}{d^{(HC)}\alpha} \quad (23)$$

In the case of ***EM-ERHC***, $\alpha = 2\beta$. And for the repair in one ***Group***, $d^{(ERHC)} = d^{(HC)}+1$. Then the repair traffic can be improved $IV_{rp}^{(HC)} = 1 - \frac{d^{(HC)}+1}{2d^{(HC)}}$.

To get an intuition for reduction in repair traffic, we look at two examples.

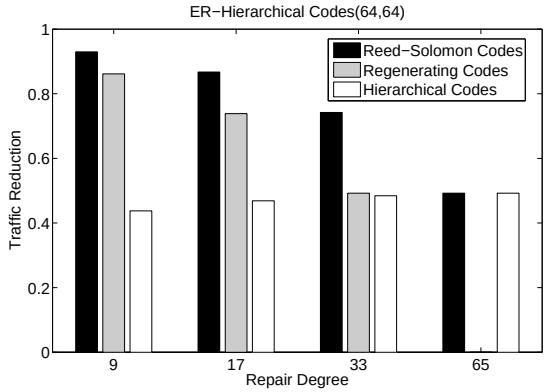
— (**S1**) Code-(64,64) with parameters: $d_0 = 8$, $g_1 = g_2 = g_3 = 2$, $h_0 = h_1 = h_2 = 4$ and $h_3 = 8$.
— (**S2**) Code-(32,32) with parameters: $d_0 = 4$, $g_1 = g_2 = g_3 = 2$, $h_0 = h_1 = h_2 = 2$, and $h_3 = 4$.

For all four codes, we use the same values (k, h). Note that the comparison is done for the case of ***EM-ERHC*** where the coding scheme for Regenerating Codes is ***EMSR*** (see section 2.1.3). Using Eq. (21)– Eq. (23), we can compute the results depicted in Fig. 6. Compared to Hierarchical Codes, independent of the repair degree, the reduction in repair traffic due to ER-Hierarchical Codes is around 50%. When compared to Reed-Solomon or Regenerating codes, the reduction in repair traffic is about one order of magnitude when the repair degree is $d_0 + 1$, which is the case for most of the repairs of ER-Hierarchical Codes.

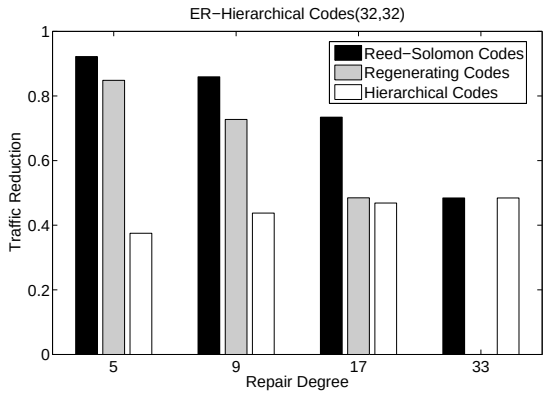
6.3. Experimental Evaluation

For the analytical evaluation, we have assumed that rule (R1) is always used for repair. Therefore the analytical results obtained indicates the maximum possible improvement. To get a more comprehensive and realistic view of the improvement due to ERHC, we use two real peer availability traces that define the peer churn. We simulate the repair process using these traces to obtain various metrics such as repair traffic, number of repairs for the different codes. Before we can present the results, we need to give a few more details about the repair process and the availability traces.

6.3.1. Repair Policy. We adopt the policy also used by Duminuco [Duminuco and Biersack 2009], which is a hybrid policy that performs (i) immediate repair if there is a danger of loosing the data and (ii) delayed repair otherwise. The repair policy assumes the presence of an entity able to monitor the availability of the nodes and to trigger a



(a) (S1):ERHC-(64,64)



(b) (S2):ERHC-(32,32)

Fig. 6. Traffic reduction due ER-Hierarchical Codes as compared to other erasure codes

repair operation accordingly. The aim of a repair policy is to assure the data reliability in face of data loss due to node churn. If the code used is a Reed-Solomon Code or a Regenerating Code, the data reliability solely depends on the number of available blocks: As long as there are at least k out of $k + h$ blocks available, the reliability is 100 %. Otherwise the reliability is zero

In case a Reed-Solomon Code or a Regenerating Code is used, the following repair policy is adopted (See Fig. 7 for an illustration of the repair policy):

— **Immediate repair:** When a peer A disconnects and the number of available peers N_a is smaller or equal to TH: $N_a \leq TH \rightarrow$ *immediately* perform the repair of the block stored on peer A.
— **Delayed repair:** When a peer A disconnects and $N_a > TH \rightarrow$ wait for a time duration T_o and if A is then still unavailable, perform a repair of the block stored on A.

The **threshold** TH is needed to assure of the data reliability. In the case of Reed-Solomon Codes, we set $TH = k+N_l$, which means that in the moment of minimum reliability, i.e. in the moment of *maximum risk*, the system can still support N_l more losses while maintaining 100 % reliability. Similarly, for Regenerating Codes the threshold is $TH = k + N_l + 1$.

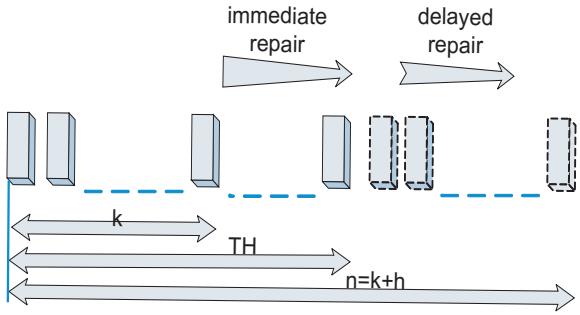


Fig. 7. Repair Policy for Reed Solomon and Regenerating Codes

In the case of Hierarchical Codes and ER-Hierarchical Codes, the situation is slightly different since the reliability does not only depend on the number of blocks that are available but also on *which* blocks are available. In this case we define reliability as $1 - P(\textit{failure}|N_l)$. and adopt the following repair policy:

— **Immediate repair:** When a peer A disconnects and $P(\textit{failure}|N_l) > 10^{-4} \rightarrow$ *immediately* perform the repair of the block stored on peer A.
— **Delayed repair:** When a peer A disconnects and $P(\textit{failure}|N_l) \leq 10^{-4} \rightarrow$ wait for a time duration T_o and if A is then still unavailable, perform a repair of the block stored on A.

As we discussed in section 3, peers leave the system concurrently and dynamically which leads to *concurrent failures*. To provide high reliability, we have introduced the parameter N_l. The two repair policies (i) for the RS and RG code and (ii) for HC and ERHC are defined in such a way that repair will be delayed only if at least another N_l failures can be tolerated before the data will become unavailable. This assures that we evaluate all codes using the same guarantees in terms of data availability. The particular values chosen for N_l and $P(\textit{failure}|N_l)$ throughout the simulations are $N_l = 10$ and $P(\textit{failure}|N_l) = 10^{-4}$ For both parameters, we have explored a wider range of values but did not see that results were sensitive to the particular choice of value.

The **timeout** T_o is used to distinguish between transient and permanent failures, which is very important since the transient failures occur frequently in P2P systems and may result in unnecessary repairs. If a node has been offline for more than T_o and its blocks were subject to delayed repair, than the blocks are assumed to be lost and will be repaired. Choosing the right value for T_o is challenging: When T_o is set to a low value, it means the *delayed repair* will be **eager**. When T_o is set to a large value, it means the *delayed repair* will be **lazy**. To compare the performance of our codes and the impact of *timeout* for different erasure codes, we will use different values for T_o. As usually done in the literature [Druschel and Rowstron 2001; Dabek et al. 2001; Adya et al. 2002; Kubiatowicz et al. 2000; Blake and Rodrigues 2003; Haeberlen et al. 2005; Chun et al. 2006], we use the same timeout value for all the peers.

6.3.2. Trace Analysis. In the experiments we use two different availability traces from **Skype** and **PlanetLab** [Godfrey 2006] as input to our simulator. We *randomly* place the blocks on the online peers without optimization. To mediate the impact of placement, we run the simulation 10 times with different random seeds and get the *mean* of each cases as the final result for the experiments.

As discussed in [Kondo et al. 2010], the peer *volatility* in the Skype trace is high and in the PlanetLab trace is low. The peer *availability* in the Skype trace is low and in the PlanetLab trace is medium. We can therefore say that the Skype trace represents a

very dynamic environment and the PlanetLab traces represent a *stable* environment. To help characterize these two traces, we define the following metrics:

— N_p: *number of peers*.
— $R_o^{(i)}$: *online rate* of peer i, formally $R_o^{(i)} = \frac{T_{on}^{(i)}}{T_{on}^{(i)}+T_{off}^{(i)}}$
— $\overline{R_o}$: *average online rate*, formally $\overline{R_o} = \frac{\sum_{i=1}^{N_p} R_o^{(i)}}{N_p}$.
— $N_{p\{R_o>0.5\}}$: *number of peers* whose online rate is larger than 50%.
— T_{dur}: duration time of the traces.

In these two traces, some peers exist for only one session and then abandon the system. Since real storage systems such as Wuala [Wuala 2010] use incentive mechanism to make peers come back, we exclude peers with only one session from the traces. After eliminating these peers, we can compute the following metrics presented in table II.

Table II. Trace Characteristics

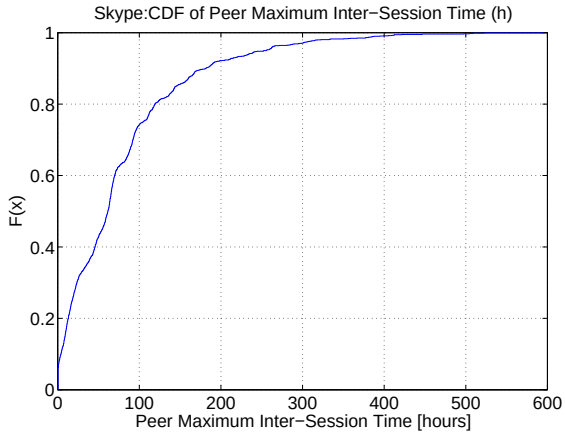
Trace	N_p	$\overline{R_o}$	$N_{p\{R_o>0.5\}}$	$T_{dur}[days]$
Skype	1598	56.13%	797	30
PlanetLab	659	75.37%	560	500

6.3.3. Maximum Inter-Session Time. *Inter-session time* is defined as the time period a peer k is absent from the system, i.e. $IST_j(k) = t_{on}^{(j+1)}(k) - t_{off}^{(j)}(k)$. Then let $IST_{max}(k)$ denote the *maximum inter-session time* for peer k, formally $IST_{max}(k) = max_{\forall j}(IST_j(k))$. If we knew for each peer its $IST_{max}(k)$ we could efficiently distinguish a permanent failure from transient one. To avoid any unnecessary block repairs we would set the *timeout* value T_o for peer k to be $IST_{max}(k)$. However, in a real system the $IST_{max}(k)$ values are not known. Therefore, to select a timeout T_o, we look at the *cumulative distribution function* (*CDF*)) of the peer *maximum inter-session time* in the traces denoted as ***F(*IST_{max}*)***, which is depicted in the Fig. 8. We set the timer T_o to be the value of one of the *deciles* of the ***F(*IST_{max}*)***; namely for **eager repair**, such that $F(T_o) = 0.2$, for **slow repair**, such that $F(T_o) = 0.5$ and for **lazy repair**, such that $F(T_o) = 0.8$.

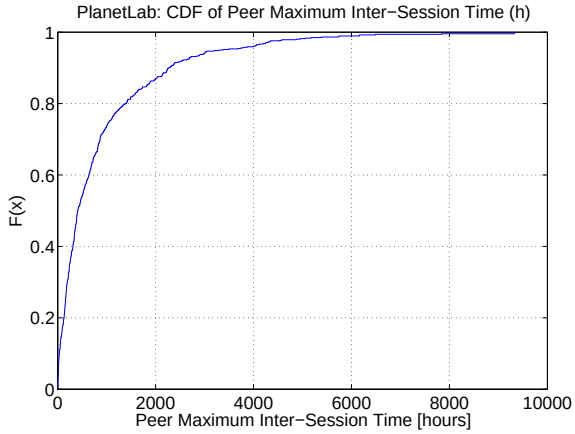
To get an intuition how these three repair modes impact block availability, we do a simple experiment for the RS(64,64) using the Skype trace to obtain the *distribution of the number of available blocks* when a repair occurs, which is depicted in the Fig. 9. The curves show that (1) The case $N_a \leq TH = k + N_l = 74$ occurs frequently for *lazy repair mode*, which means lots of *immediate repairs* are triggered in this mode. Conversely, few *immediate repairs* are triggered in the *eager repair mode*. (2) The *delayed repairs* mainly occur when N_a is large and many more *delayed repairs* occur in the *eager repair mode* than the *slow repair mode* or *lazy repair mode*.

6.3.4. Metrics. In order to sufficiently study the performance of different cases, we define the following metrics and test most of them:

— N_{imrp}: total number of *immediate repairs* using (P1) during the whole simulation.
— N_{dlrp}: total number of *delayed repairs* using (P2) during the whole simulation.
— N_{trp}: total number of repairs during the whole simulation.
— P_{imrp}: percentage of *immediate repairs*, $P_{imrp} = \frac{N_{imrp}}{N_{trp}}$.
— P_{dlrp}: percentage of *delayed repairs*, $P_{dlrp} = \frac{N_{dlrp}}{N_{trp}}$.
— V_{trp}: total repair traffic during the whole simulation, unit is S_{file}.
— V_{arp}: average traffic per repair, $V_{arp} = \frac{V_{trp}}{N_{trp}}$, unit is S_{file}.



(a) Skype



(b) PlanetLab

Fig. 8. Cumulative distribution function of peer maximum inter-session time in the traces

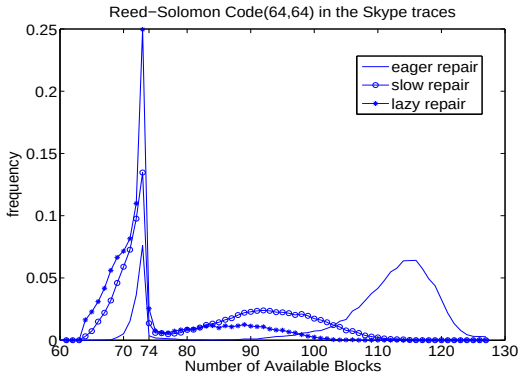


Fig. 9. Distribution of the number of available blocks.

— $IV_{trp}^{(ec)}$: the improvement on reducing *total repair traffic* by ER-Hierarchical Codes as compared to the *ec*.
— $IV_{arp}^{(ec)}$: the improvement on reducing *average traffic per repair* by ER-Hierarchical Codes as compared to the *ec*.

The most important metrics is V_{trp} as the amount of repair traffic impacts the total amount of data that can be safely stored in the system [Rodrigues and Liskov 2005]. The other metrics are mainly useful to compare different coding schemes or to understand in how the system is operating (e.g. P_{imrp} and P_{dlrp}).

6.3.5. Results and Analysis of Different Cases. Now that we have discussed the details of simulation we can evaluate two cases (S1) a (64,64) Code and (S2) a (32,32) Code defined in section 6.2.3. The redundancy for both codes is $r = \frac{k+h}{k} = 2$. We also carried out the same the experiments for $r = 2.5$ with a code whose structure is $d_0 = 4$, $g_1 = g_2 = g_3 = 2$, $h_1 = h_2 = 2$, and $h_0 = h_3 = 4$. In this case the result for ERHC was even better since more repair could be done using rule (R1).

The following tables Tab.III and Tab.IV show the results for the different erasure codes using the two availability traces.

Table III. Skype Traces Result (the unit of traffic is S_{file})

Timeout	Codes	P_{imrp}	P_{dlrp}	N_{trp}	V_{trp}	$IV_{trp}^{(ec)}$	V_{arp}	$IV_{arp}^{(ec)}$
eager repair 11.29 $[hours]$	ERHC(S1)	0.251	0.749	447.6	81.638	0.000	0.182	0.000
	HC(S1)	0.257	0.743	449.9	119.250	0.315	0.265	0.312
	RG(64,64)	0.161	0.839	378.3	192.106	0.575	0.508	0.641
	RS(64,64)	0.145	0.855	365.8	365.800	0.777	1.000	0.818
	ERHC(S2)	0.581	0.419	382.2	67.927	0.000	0.178	0.000
	HC(S2)	0.578	0.422	368.5	94.862	0.284	0.258	0.309
	RG(32,32)	0.314	0.686	227.7	117.408	0.421	0.516	0.655
	RS(32,32)	0.292	0.708	232.0	232.000	0.707	1.000	0.822
slow repair 60.49 $[hours]$	ERHC(S1)	0.651	0.349	377.3	115.869	0.000	0.306	0.000
	HC(S1)	0.657	0.343	374.8	140.315	0.174	0.374	0.182
	RG(64,64)	0.523	0.477	266.5	135.332	0.144	0.508	0.397
	RS(64,64)	0.504	0.496	250.1	250.100	0.537	1.000	0.694
	ERHC(S2)	0.887	0.113	419.6	84.869	0.000	0.203	0.000
	HC(S2)	0.890	0.110	444.7	132.150	0.358	0.297	0.318
	RG(32,32)	0.737	0.263	217.5	112.148	0.243	0.516	0.607
	RS(32,32)	0.706	0.294	199.2	199.200	0.574	1.000	0.797
lazy repair 118.17 $[hours]$	ERHC(S1)	0.864	0.136	427.0	149.683	0.000	0.352	0.000
	HC(S1)	0.856	0.144	428.9	182.205	0.178	0.426	0.174
	RG(64,64)	0.793	0.207	288.0	146.250	**-0.023**	0.508	0.308
	RS(64,64)	0.777	0.223	272.3	272.300	0.450	1.000	0.648
	ERHC(S2)	0.957	0.043	457.8	95.739	0.000	0.209	0.000
	HC(S2)	0.956	0.044	471.5	145.500	0.342	0.308	0.320
	RG(32,32)	0.878	0.122	249.8	128.803	0.257	0.516	0.594
	RS(32,32)	0.870	0.130	226.3	226.300	0.577	1.000	0.791

From the results, we can draw the following conclusions:

— (Refer to columns T_o and P_{dlrp}):
The *Timeout* value impacts the *delayed repairs* directly (see the repair policy). For the *eager repair mode*, delayed repairs occur frequently (see Fig. 9) and the *percentage of delayed repairs* is high. Conversely, the *percentage of delayed repairs* is low in the *lazy repair mode*.
— (Refer to column $V_{arp}(M\%)$):
In all the repair cases, the average traffic per repair is minimized for ER-

Table IV. PlanetLab Traces Result (the unit of traffic is S_{file})

Timeout	Codes	P_{imrp}	P_{dlrp}	N_{trp}	V_{trp}	$IV_{trp}^{(ec)}$	V_{arp}	$IV_{arp}^{(ec)}$
eager repair 5.36 $[days]$	ERHC(S1)	0.005	0.995	433.5	54.842	0.000	0.127	0.000
	HC(S1)	0.012	0.988	435.9	98.588	0.444	0.226	0.441
	RG(64,64)	0.000	1.000	429.7	218.207	0.749	0.508	0.751
	RS(64,64)	0.000	1.000	429.7	429.700	0.872	1.000	0.873
	ERHC(S2)	0.281	0.719	281.5	44.850	0.000	0.160	0.000
	HC(S2)	0.271	0.729	284.2	68.787	0.348	0.242	0.341
	RG(32,32)	0.015	0.985	213.6	110.138	0.593	0.516	0.691
	RS(32,32)	0.011	0.989	213.4	213.400	0.790	1.000	0.840
slow repair 17.06 $[days]$	ERHC(S1)	0.192	0.808	230.6	39.497	0.000	0.171	0.000
	HC(S1)	0.186	0.814	227.7	59.237	0.333	0.260	0.342
	RG(64,64)	0.085	0.915	204.6	103.899	0.620	0.508	0.663
	RS(64,64)	0.072	0.928	201.6	201.600	0.804	1.000	0.829
	ERHC(S2)	0.686	0.314	258.4	48.584	0.000	0.188	0.000
	HC(S2)	0.671	0.329	249.8	71.662	0.322	0.287	0.344
	RG(32,32)	0.231	0.769	120.1	61.926	0.215	0.516	0.635
	RS(32,32)	0.203	0.797	117.0	117.000	0.585	1.000	0.812
lazy repair 57.64 $[days]$	ERHC(S1)	0.609	0.391	208.9	58.224	0.000	0.272	0.000
	HC(S1)	0.586	0.414	196.9	67.213	0.134	0.339	0.198
	RG(64,64)	0.343	0.657	130.7	66.371	0.123	0.508	0.464
	RS(64,64)	0.315	0.685	124.6	124.600	0.533	1.000	0.728
	ERHC(S2)	0.909	0.091	311.2	62.475	0.000	0.201	0.000
	HC(S2)	0.904	0.096	319.1	96.750	0.354	0.304	0.339
	RG(32,32)	0.709	0.291	125.3	64.608	0.033	0.516	0.610
	RS(32,32)	0.652	0.348	110.3	110.300	0.434	1.000	0.799

Hierarchical Codes. Especially in the case of *eager repair mode*, the average traffic per repair is smallest as we can choose a lower repair degree.

— (Refer to columns N_{trp} and V_{trp}):
In most cases, ERHC requires a *higher number of repairs* but results in a *lower total repair traffic* as compared to other erasure codes. A similar observation was previously made by Dimunuco [Duminuco and Biersack 2009] for Hierarchical Codes. In order to provide the same high reliability as the other codes, ERHC and HC must do more repairs but the *average traffic per repair* is much lower than for the other codes so that the total repair traffic will still be lower than for the other codes.

— (Refer to column $IV_{trp}^{(ec)}$):
In terms of total repair traffic, ERHC performs better than other codes in all cases, except one case of *lazy repair* for the Skype trace when using RG(64,64). The explanation for this outlier is two-fold: (1) For *lazy repair* with a timeout of $T_o = 118.17$ hours, it is often not possible to achieve a low repair degree for ERHC, which leads to the highest traffic per repair ($0.352S_{file}$) as compared to other cases of ERHC in the Skype traces, (2) In order to provide the same reliability as RG(64,64), ERHC requires many more repairs.

— (Refer to column N_{trp} to compare *Code*(64,64) with *Code*(32,32)):
There are twice as many nodes involved in storing the blocks of a (64,64) Code than of a (32,32) Code. As a consequence, more node failures will occur for a (64,64) Code and more repairs are triggered, when Regenerating Codes or Reed-Solomon Codes are used. However, for ERHC and HC with *slow* or *lazy* repair mode, the *number of repairs* for the Code (64,64) is lower than that for the Code (32,32). The reason is that in both cases an immediate repair will be triggered whenever $P(\textit{failure}|N_l) > 10^{-4}$. This implies that for the Code (32,32) the number of failures that can be tolerated before immediate repair will be done is much smaller than for the Code (64,64). Since a higher number of immediate repairs implies also a higher number of *unnecessary* repairs, the total number of repairs will be higher for the Code (32,32). On

the other hand, in the case of eager repair mode, the Code (32,32) performs better than the Code (64,64).

— (Refer to column V_{trp} to compare the repair mode:) For RG and RS, the traffic for each repair is the same. So the *total repair traffic* depends only on the *number of repairs*. As discussed in [Blake and Rodrigues 2003; Haeberlen et al. 2005; Duminuco 2009], *slow* or *lazy repair* can perform better than *eager repair* in terms of total repair traffic because they can avoid more useless repairs triggered by transient node failures. ERHC or HC achieve good performances in all the repair modes. The performance under the *lazy repair mode* is the worse compared to eager or slow repair mode: As there are fewer nodes available, the repair degree increases and the more costly EHC-type repairs need to be done. To select the best repair mode for each code, we can consult Tab.V.

Table V. Best repair mode for the different codes

Codes	Skype	PlanetLab
ERHC(S1)/HC(S1)	eager	slow
ERHC(S2)/HC(S2)	eager	eager
RG/RS	slow	lazy

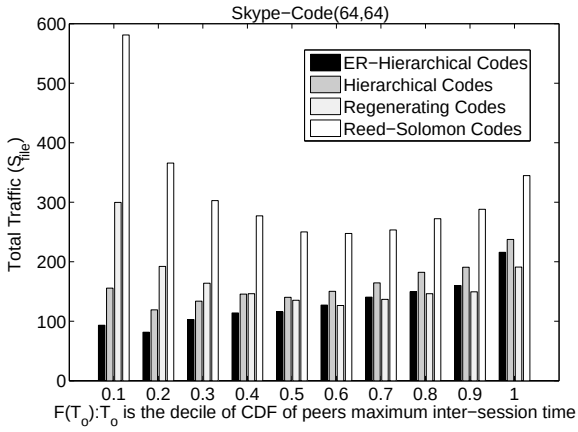
6.3.6. Impact of timeout. To further study the impact of *timeout* T_o on different codes, we define the *variance of total repair traffic* as using erasure code EC, which is denoted as *var(EC)*. To make sense the statical analysis of variance, we repeat the simulation again with the *timeout* to be set to all the deciles of the CDF of the peer maximum inter-session time ($F(T_o) = \frac{i}{10}$). And then we get the result of total repair traffic depicted in Fig. 10 and Fig. 11. In doing so, we get the *var(EC)* as follows. Let $x_i = V_{trp}^{(F(T_o)=\frac{i}{10})}$, then the mean is $\overline{x} = \frac{\sum_{i=1}^{10} x_i}{10}$. So *var(EC)* $= \frac{\sum_{i=1}^{10} (x_i - \overline{x})^2}{10}$. From the results shown in Tab.VI, we can conclude that the impact of the choice of the *timeout* value in the case of ER-Hierarchical Codes is less than for other erasure codes (except one case of Code (64,64) as compared to HC), which indicates that the performance of ER-Hierarchical Codes is less sensitive to the choice of T_o than the other codes.

Table VI. Variance of Total Repair Traffic

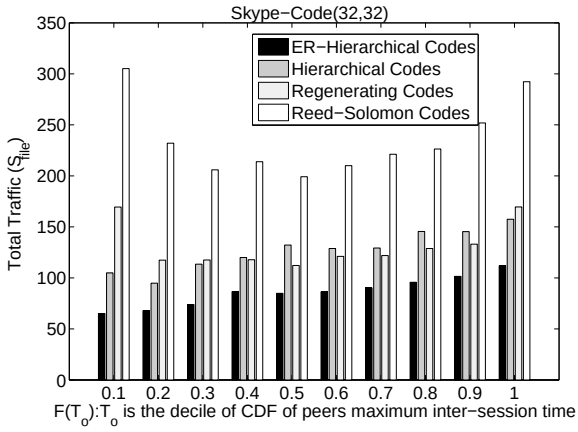
Trace	(k, h)	*ERHC*	*HC*	*RG*	*RS*
Skype	$(64, 64)$	1517.35	1164.60	2614.42	10088.93
Skype	$(32, 32)$	214.59	377.92	449.56	1327.85
PlanetLab	$(64, 64)$	783.01	1673.03	9692.94	38698.98
PlanetLab	$(32, 32)$	235.35	495.16	1670.63	6686.53

6.3.7. Repair type and repair degree. Both, the repair type and the repair degree have an influence on the repair traffic: (i) RC-type repair results in less traffic than EHC-type repair and (ii) the lower the repair degree the lower the repair traffic.

In Fig. 12 we see percentage of RC-type repair as a function of the timeout value used for delayed repair: In case of very small timeout values, RC-type repair is done more than 70% of the time, while this value drops for very large timeout values to 30% to 40%. The larger the timeout, the longer a repair will be deferred and the lower the chances that a enough nodes are available to provide the number of repair blocks needed for RC-type repair. In case RC-type repair is done, the repair traffic is minimal if repair can be done at the lowest level. Given RC-type repair is applied, then P_{llR1} denotes the percentage of the *RC-type* repairs that are done in the lowest level.



(a) Code-$(64, 64)$



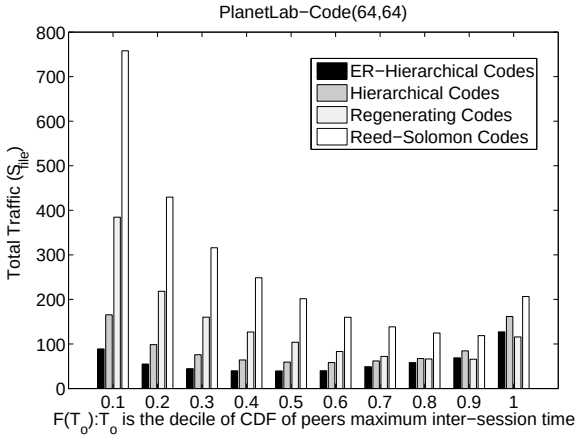
(b) Code-$(32, 32)$

Fig. 10. Total repair traffic V_{trp} in the Skype traces

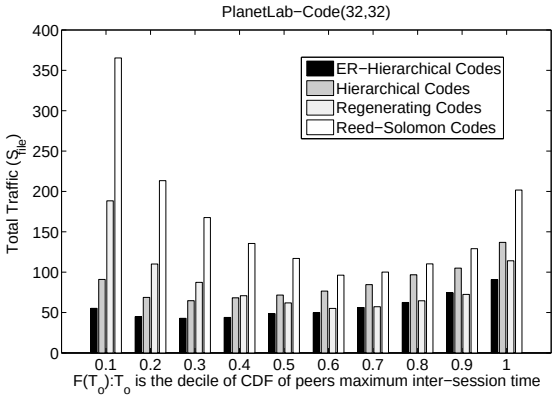
The curves for P_{llR1} in Fig. 13 show a similar trend the ones in Fig. 12, namely the percentage of RC-type repairs at lowest level decreases with increasing timeout value T_o.

In Fig. 14 we see that, independent of the timeout value T_o, only a small fraction of repairs are done at the highest level. On the other hand, in many cases the repair can be done at the lowest level. When T_o is small, over 70% of the repairs are done at lowest level, which not only keeps the repair traffic small but also minimizes the number of nodes uploading blocks to the newcomer.

6.3.8. Impact of percentage of delayed repairs. (i) **Impact on average repair traffic**. As we discussed before, for ERHC or HC the *eager repair mode* often results in a *lower repair degree* so that the *average repair traffic* will be low and the *percentage of delayed repairs* will be high. To better understand the correlation between the number of *delayed repairs* and the *average traffic per repair*, we look at the relation between *percentage of delayed repairs* and the *average repair traffic*, which is shown in Fig. 15(a). We can see that the *average repair traffic* almost linearly decreases as the *percentage*



(a) Code-$(64, 64)$



(b) Code-$(32, 32)$

Fig. 11. Total repair traffic V_{trp} in PlanetLab traces

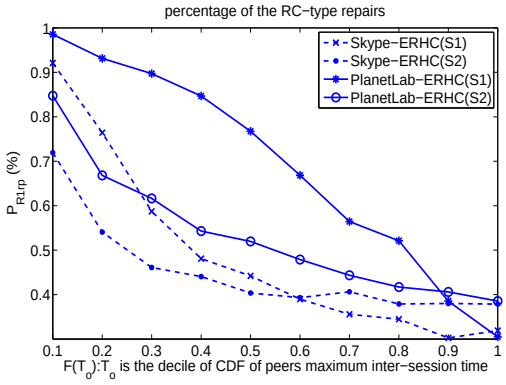


Fig. 12. Percentage of the RC-type repairs.

of delayed repairs increases. To confirm that linear relation, we compute the *correla-*

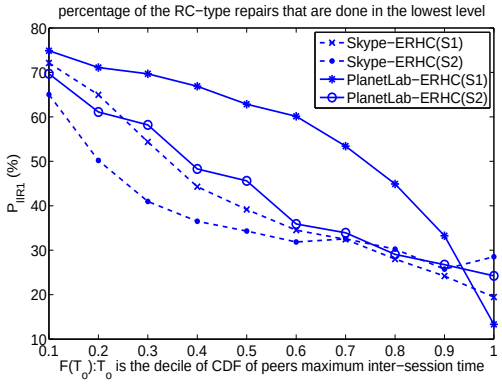


Fig. 13. Percentage of the **RC-type** repairs done at the lowest level.

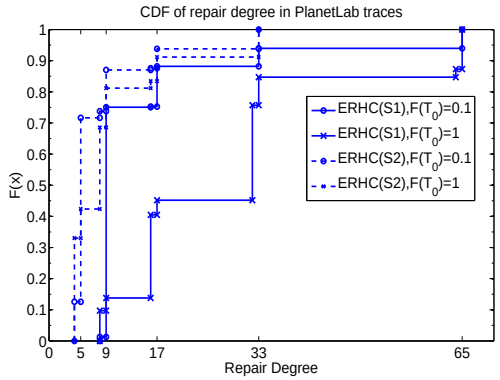
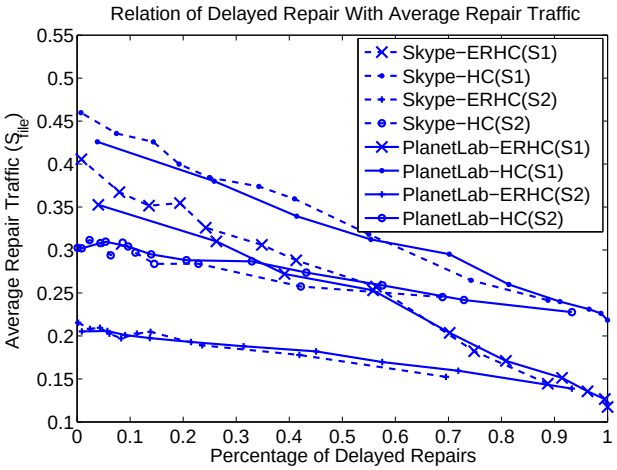


Fig. 14. Distribution of the repair degree for Planetlab trace.

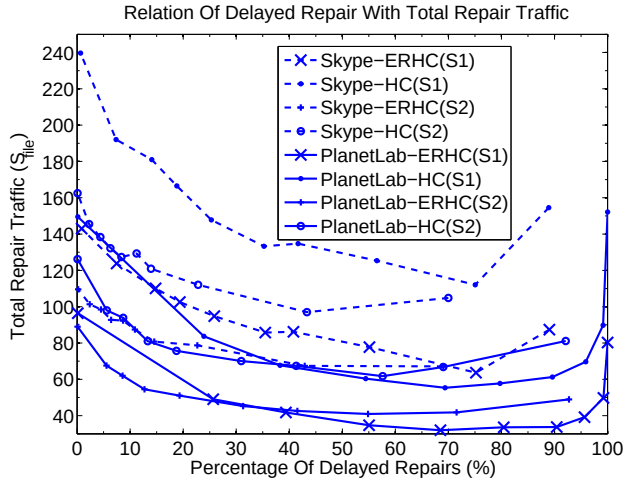
tion coefficient $cc = \frac{10\sum P_{dlrp}(i)*V_{arp}(i)-\sum P_{dlrp}(i)\sum V_{arp}(i)}{\sqrt{10\sum P^2_{dlrp}(i)-(\sum P_{dlrp}(i))^2}\sqrt{10\sum V^2_{arp}(i)-(\sum V_{arp}(i))^2}}$. We get $|cc| > 0.95$ in all the cases for ERHC and HC. So we can conclude that P_{dlrp} and V_{arp} are *strongly correlated* denoted as $P_{dlrp} \propto V_{arp}$. This observation can be exploited in the design of an efficient repair policy for ERHC and HC. While the choice of a suitable *timeout* is very difficult, the *percentage of delayed repairs* is easy to observe.

As we know, the *average repair traffic* directly impacts the time duration of a repair. The longer a repair takes, the higher possibility that other nodes fail, which in turn will increase the repair time. To control the average repair traffic one can measure the *percentage of delayed repairs* and adjust the timeout accordingly. We leave the optimal control of the repair policy as future work.

(ii) **Impact on total repair traffic**. Since $P_{dlrp} \propto V_{arp}$ and $P_{dlrp} = \frac{N_{dlrp}}{N_{trp}}$ and $V_{arp} = \frac{V_{trp}}{N_{trp}}$, we can get $_{dlrp} \propto V_{trp}$ if N_{trp} is constant. However in practice, N_{trp} varies as a function of *timeout*. In Fig. 15(b) we show the *total repair traffic* as a function of the *percentage of delayed repairs*. We can see that the *total repair traffic* is the lowest when the *percentage of delayed repairs* is in the range of $[58\%, 81\%]$ for the Planetlab trace and $[40\%, 75\%]$ for the Skype trace.



(a) *Average repair traffic* as function of *delayed repairs*.



(b) *Total repair traffic* as function of *delayed repairs*.

Fig. 15. Impact of *delayed repairs* on repair traffic.

6.4. Computational Complexity

A major concern when coding is used is the computational complexity of the coding and decoding operations. In the following, we will present a formal comparison of the complexity of the different codes in terms of the number of elementary operations that need to be executed.

6.4.1. Basics. All operations such as addition and multiplications are executed in a Galois Field GF(2^q) where every value is represented as a sequence of q bits. The value of q is typically 16 and the operations are performed on unsigned short integers. Every fragment is a vector of N_{fg} elements of q bits in length. (i) Addition and subtraction correspond to an XOR operation between two elements. (ii) Multiplication and division are performed in the log-space. For example: $a \cdot b$ becomes $\exp(\log a + \log b)$. For $\log$ and $\exp$ all the possible values in GF(2^q) are pre-computed and stored in a table, which requires 256 KB of memory for $q = 16$. The operations $\log$ and $\exp$ can then be imple-

mented as value lookups in a table, which results for the division and multiplication in 3 lookups and 1 addition.

All the operations for coding, decoding, and repair can be reduced to: (i) Linear Combinations and (ii) Matrix inversion. Let us analyze them in detail: (i) A linear combination of n vectors of length l consists of $n \cdot l$ additions and $n \cdot l$ multiplications and will therefore require a total of $5nl$ operations. (ii) The inversion of a square (n, n) matrix consists of n^3 additions and n^3 multiplications that can be implemented $5n^3$ operations.

The cost for matrix inversion is negligible since the number n of coefficients is much smaller than the number N_{fg} of elements a fragment consists of [Duminuco 2009]. In the following, we will therefore ignore the cost for matrix inversion.

Let us now consider the computational cost for each of the codes code:

— **Reed-Solomon Code**: The participating peers only send their parity block to the newcomer. The newcomer will first multiply the parity blocks with the inverse of the coefficient matrix to obtain all the original blocks and then linearly combine the original blocks to create one new parity block. The decoding corresponds to k linear combinations of k parity blocks. The length of each block is $\alpha N_{fg} elements$. The repair cost is:

$$\begin{aligned} CPU(repair)_{down}^{(RS)} &= CPU(decode) + CPU(combine) \\ &= 5 \cdot k^2 \cdot (\alpha N_{fg}) + 5 \cdot k \cdot (\alpha N_{fg}) \\ &= 5k(k+1)\alpha N_{fg} \end{aligned} \tag{24}$$

— **Hierarchical Code**: The operations executed are the same as for a Reed-Solomon Code. However, the number of blocks is d and not k. Using Eq. (24), we get

$$CPU(repair)_{down}^{(HC)} = 5d(d+1)\alpha N_{fg} \tag{25}$$

— **Regenerating Code**: Every participating peer performs one linear combination of α parity fragments, which corresponds to:

$$CPU(repair)_{up}^{(RG)} = 5 \cdot \alpha \cdot N_{fg} \tag{26}$$

The newcomer performs α linear combinations of d parity fragments, which corresponds to:

$$\begin{aligned} CPU(repair)_{down}^{(RG)} &= 5 \cdot \alpha \cdot d \cdot N_{fg} \\ &= 5(k + \alpha - 1)\alpha N_{fg} \end{aligned} \tag{27}$$

The last equality holds if the Regenerating code used is MSR and the repair degree is fixed to $d = k + \alpha - 1$. From Eq. (26) and Eq. (27), we see that newcomer has a larger computational cost, so we only consider the computational cost of newcomer in the following.

— **ER-Hierarchical Code**: We have two types of repair, namely EHC-type repair and RC-type repair:
(i) EHC-type repair: the participating peers only send their parity fragments to the newcomer who will decode the parity fragments received to first obtain $d\alpha$ original fragments, which are then linearly combined the to compute α parity fragments. Decoding requires to multiply the parity fragments by the inverse of its coefficient matrix and corresponds to $d\alpha$ linear combinations of $d\alpha$ parity fragments.

$$\begin{aligned} CPU(repair)_{down}^{(EHC-type)} &= CPU(decode) + CPU(combine) \\ &= 5 \cdot (d\alpha)^2 \cdot N_{fg} + \alpha \cdot 5 \cdot (d\alpha) \cdot N_{fg} \end{aligned}$$

$$= 5d(d+1)\alpha^2 N_{fg} \quad (28)$$

(ii) RC-type repair has a cost as in Eq. (27). However, now the repair degree d varies since it depends on the level at which the repair is performed.

$$CPU(repair)_{down}^{(RC-type)} = 5d\alpha N_{fg} \quad (29)$$

If we compare Eq. (28) and Eq. (29), we see that EHC-type repair is computationally more expensive than RC-type repair.

6.4.2. Cost Comparison. Given the repair cost for the different codes, we can now compare them using as metric the **cost ratio**: $R_{cpu-type}^{(code)}$ denotes the ratio of the repair cost of a code *code* compared to the repair cost of ERHC using repair-*type*, where repair-*type* is either EHC-type or RC-type:

$$R_{cpu-type}^{(code)} = \frac{CPU(repair)_{down}^{(code)}}{CPU(repair)_{down}^{(ERHC-type)}}$$

In the following, the variables k and d play an important role: k refers to the number of original blocks a file was decomposed and d denotes the repair degree. While k will be fix, the value of d depends at what level of the code the repair can be performed. For Hierarchical codes or EHC-type repair, the values for d will be in the range of $d \in [d_0, k]$ and for RC-type repair the d will be in the range of $d \in [d_0 + \alpha - 1, k + \alpha - 1]$, with $\alpha = 2$ in our case. If we look at distribution of the values of d (c.f. Fig. 14) we see that in most cases $d \ll k$.

— *Comparison of ERHC to RS:*

(i) When ERHC does EHC-type repair, using Eq. (24) and Eq. (28), we get $R_{cpu-hc}^{(RS)} = \frac{k(k+1)}{d(d+1)\alpha}$, with k fixed and $d \in [d_0, k]$. For all values $d < k$, the repair cost of RS will be higher. Only in the case of $d = k$ we have a cost ratio of $R_{cpu-hc}^{(RS)} = \frac{1}{\alpha}$. Since $\alpha = 2$, the cost ERHC repair will be – in the worst case – twice the cost for RS.

(ii) When ERHC does RC-type repair, using Eq. (24) and Eq. (29), we get a cost ratio of $R_{cpu-rg}^{(RS)} = \frac{k(k+1)}{d}$, with k fixed and $d \in [d_0 + 1, k + 1]$. Since the largest value of d is $d = k + 1$, we obtain $R_{cpu-rg}^{(RS)} \geq k$, i.e. the repair cost of RS is always at least k times higher.

Given that the repair degree d varies, we present in Tab.VII the cost ratio $R_{cpu-erhc}^{(RS)}$ as function of the repair degree.

Table VII. $R_{cpu-erhc}^{(RS)}$ as the function of repair degree for ERHC(S1).

Repair Degree	8	9	16	17	32	33	64	65
$R_{cpu-erhc}^{(RS)}$	28.9	462.2	7.6	244.7	1.97	126.1	0.50	64

The computational cost of ERHC(S1) is in all but one case lower than for RS.

— *Comparison of ERHC to HC:*

(i) When ERHC does EHC-type repair, it uses the same repair degree as HC. Using Eq. (25) and Eq. (28), we can get a ratio of $R_{cpu-hc}^{(HC)} = \frac{1}{\alpha}$, i.e. the cost of ERHC repair will be twice the one of HC.

(ii) When ERHC does RC-type repair, using Eq. (25) and Eq. (29), we get $R_{cpu-rg}^{(HC)} = \frac{d^{(HC)}(d^{(HC)}+1)}{d^{(RC-type)}}$. Since $d^{(RC-type)} = d^{(HC)} + 1$, we get $R_{cpu-rg}^{(HC)} = d^{(HC)}$, with $d^{(HC)} \in [d_0, k]$.

We give the cost ratios $R^{(HC)}_{cpu-erhc}$ in Tab.VIII.

Table VIII. $R^{(HC)}_{cpu-erhc}$ as the function of repair degree for ERHC(S1).

Repair Degree	8	9	16	17	32	33	64	65
$R^{(HC)}_{cpu-erhc}$	0.5	8	0.5	16	0.5	32	0.5	64

— *Comparison of ERHC to RG:*
(i) When ERHC does EHC-type repair, using Eq. (27) and Eq. (28), we get $R^{(RG)}_{cpu-hc} = \frac{k+\alpha-1}{d(d+1)\alpha}$, with k fixed and $d \in [d_0, k]$. (ii) When ERHC does RC-type repair, using Eq. (27) and Eq. (29), we can get $R^{(RG)}_{cpu-rg} = \frac{k+1}{d}$, with k fixed and $d \in [d_0+1, k+1]$. We give the cost ratios $R^{(RG)}_{cpu-erhc}$ in Tab.IX. Compared to a Regenerating Code, ERHC with EHC-type repair is computationally more expensive, while for RC-type repair ERHC is computationally less demanding.

Table IX. $R^{(RG)}_{cpu-erhc}$ as the function of repair degree for ERHC(S1).

Repair Degree	8	9	16	17	32	33	64	65
$R^{(RG)}_{cpu-erhc}$	0.45	7.2	0.12	3.8	0.03	1.97	0.008	1

6.4.3. Conclusion. All the different cost ratios are summarized in the Tab.X. We see that when ERHC uses RC-type repair, its computational cost is lower than any of the three other codes. When ERHC uses EHC-type repair, its computational cost is worse than HC and RG, but better than RS, except for $d = k$ (c.f. Tab.VII).

Table X. Computational cost ratios.

	ERHC	
Code	EHC-type	RC-type
RS	$\frac{k(k+1)}{d(d+1)\alpha}$	$\frac{k(k+1)}{d}$
HC	$\frac{1}{\alpha}$	$d^{(HC)}$
RG	$\frac{k+1}{d(d+1)\alpha}$	$\frac{k+1}{d}$

In our evaluation we have only considered the computational cost in terms of elementary operations and we did not implement the different codes to benchmark them. However, we are confident that coding and decoding will not be the bottleneck in a *peer to peer* backup system. In peer to peer environments, the available bandwidth is typically less than 10 Mb/sec [Dischinger et al. 2007]. Given such a limited bandwidth, any implementation that is able to encode/decode at a rate of several MB/sec will suffice. For Reed Solomon codes, rates in the order of tens of MB/sec are achievable today [Plank 2008].

We also would like to mention that the Graphics Processing Unit (GPU) available in standard PCs can be used to execute highly parallel implementations of coding and decoding operations to achieve rates in the order of hundreds of MB/sec while completely off-loading the main CPU [Shojania and Li 2009].

7. SUMMARY

7.1. Conclusions

Reducing the repair in peer-to-peer backup systems is crucial for increasing the overall amount of data that can safely maintained in the system. To this purpose, we have designed a new class of erasure codes, called ER-Hierarchical Codes, which efficiently combine the ideas of Hierarchical Codes, namely a low repair degree and of Regenerating Codes, namely a reduced repair block size. Both ideas help reduce the repair traffic and tolerate more concurrent failures.

We have also proposed an *exact repair* mode for Hierarchical Codes, which maintains the original blocks over time and allows to parallelize the data reconstruction.

We show that ER-Hierarchical Codes can reduce the traffic per repair by up to 50% while maintaining the same reliability as Hierarchical Codes. Compared to Regenerating Codes of type MSR, the reduction in repair traffic can be up to one order of magnitude. Simulation experiments that use two different peer availability traces confirm that ER-Hierarchical Codes: (1) can significantly *reduce* the total repair traffic as well as the average repair traffic and that (2) the performance of ER-Hierarchical Codes in terms of repair traffic is less sensitive to the right choice of the *timeout* than other codes.

ER-Hierarchical Codes also have, on average, a much lower computational complexity than Reed Solomon codes.

7.2. Future Work

The work presented here can be extended in several ways to further exploit the possibilities offered by ER-Hierarchical Codes.

The hybrid repair policy used relies in case of delayed repairs on a timeout to distinguish between transient and permanent failures. Delayed repairs reduce the number of unnecessary repairs and improve the overall performance of the system. While it is difficult in practice to choose the right timeout value, it is easy to measure the percentage of delayed repairs. More work is needed to understand how the knowledge about the *percentage of delayed repairs* can be used to adjust the timeout value in order to minimize the repair traffic.

While hybrid repair policies help reduce unnecessary repairs, they may trigger repair events in bursts, e.g. when the number of available blocks is below *TH* and several concurrent losses occur. To avoid bursts of repair events, which result in an uneven utilisation of resources such as network bandwidth, *proactive repair policies* [Chun et al. 2006; Duminuco et al. 2007] have been proposed that produce new blocks at a constant rate that must match the long term block loss rate. If proactive repair is used with a traditional coding scheme such as Reed Solomon Codes, the repair traffic will be high due to an increase in the number of unnecessary repairs. However, we have seen that ERHC with eager repair minimizes the average repair traffic V_{arp} due to a very small repair degree and the fact that RC-type repair will be possible in most of the cases. Combining ERHC with proactive repair should allow to even further reduce the average repair traffic and make proactive repair very attractive.

In our evaluations, we always used $\alpha = 2$. Increasing α will decrease the total repair traffic $V_{trp}^{(MSR)}$ but unfortunately also increases the repair degree d_r. In the case of proactive repair, an increase of d_r may not be much of a problem since repairs will be performed when the number of available nodes is high. Therefore, exploring larger values of α in combination with proactive repair seems very promising.

The Regenerating Code used for the construction of the ER-Hierarchical Code is a Minimum Storage Regenerating Code (MSR). If one is willing to accept a higher storage overhead in order to further reduce the communication bandwidth needed for

repair, a Minimum Bandwidth Regenerating Code (MBR) will be attractive. However, first the details of how to combine Hierarchical Codes with an MBR code need to be worked out.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their numerous and very constructive comments. We thank Alessandro Duminuco for many discussions on Hierarchical Codes and Matteo Dell'Amico for his careful reading of an earlier version of that paper. The first author was visiting Eurecom while carrying out the research presented in this paper. He was partly supported by the China Scholarship Council. The work of Y. Peng was supported by the National Basic Research Program of China under Grant No.2011CB302601.

APPENDIX

A. REPAIR OF STORAGE BLOCK

The newcomer computes a new storage block (p_{81}, p'_{82}) in three steps.

A.1. Computation of the coefficients ν_i by newcomer

The repair fragments (p_{81}, p'_{82}) must have the following structure:
$p_{81} = (c_{81}, \ldots, c_{84})(o_{11}, \ldots, o_{41})^T$,
$p'_{82} = (u'_{81}, \ldots, u'_{84})(o_{11}, \ldots, o_{41})^T + (c_{81}, \ldots, c_{84})(o_{12}, \ldots, o_{42})^T$

Suppose we use the coefficients δ_i and ρ_i for the linear combination of repair fragments λ_i ($i \in I$). To compute (p_{81}, p'_{82}), the newcomer node can write (p_{81}, p'_{82}) as follows:

$$p_{81} = \left(\begin{pmatrix} \delta_1\nu_1 \\ \delta_2\nu_2 \\ \delta_4\nu_4 \\ \delta_5\nu_5 \\ \delta_7\nu_7 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} + \begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_4 \\ \delta_5 \\ \delta_7 \end{pmatrix}^T \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ u_{71} & u_{72} & u_{73} & u_{74} \end{pmatrix} \right) \begin{pmatrix} o_{11} \\ o_{21} \\ o_{31} \\ o_{41} \end{pmatrix} + \begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_4 \\ \delta_5 \\ \delta_7 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} \begin{pmatrix} o_{12} \\ o_{22} \\ o_{32} \\ o_{42} \end{pmatrix} \tag{30}$$

$$p'_{82} = \left(\begin{pmatrix} \rho_1\nu_1 \\ \rho_2\nu_2 \\ \rho_4\nu_4 \\ \rho_5\nu_5 \\ \rho_7\nu_7 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} + \begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_4 \\ \rho_5 \\ \rho_7 \end{pmatrix}^T \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ u_{71} & u_{72} & u_{73} & u_{74} \end{pmatrix} \right) \begin{pmatrix} o_{11} \\ o_{21} \\ o_{31} \\ o_{41} \end{pmatrix} + \begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_4 \\ \rho_5 \\ \rho_7 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} \begin{pmatrix} o_{12} \\ o_{22} \\ o_{32} \\ o_{42} \end{pmatrix} \tag{31}$$

In Eq. (30) the last term must be zero, since p_{81} does not depend on the $(o_{12}, \ldots, o_{42})$. This allows the newcomer to compute the coefficients δ_i ($i \in I$), $\delta_i \neq 0$ as

$$(\delta_1, \delta_2, \delta_4\, \delta_5, \delta_7) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} = (0, 0, 0, 0) \tag{32}$$

And in Eq. (31), the last term must be the same coefficients as the lost one which is $(c_{81}, c_{82}, c_{83}, c_{84})$. This allows the newcomer to compute the coefficients ρ_i ($i \in I$) as

$$(\rho_1, \rho_2, \rho_4, \rho_5, \rho_7) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} = (c_{81}, c_{82}, c_{83}, c_{84}) \tag{33}$$

In Eq. (30) the first term must be $(c_{81}, c_{82}, c_{83}, c_{84})$, since p_{81} is repaired exactly. Since $\delta_i \neq 0$, the newcomer can now compute the coefficients ν_i ($i \in I$) as

$$\begin{pmatrix} \delta_1\nu_1 \\ \delta_2\nu_2 \\ \delta_4\nu_4 \\ \delta_5\nu_5 \\ \delta_7\nu_7 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} + \begin{pmatrix} \delta_1 \\ \delta_2 \\ \delta_4 \\ \delta_5 \\ \delta_7 \end{pmatrix}^T \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ u_{71} & u_{72} & u_{73} & u_{74} \end{pmatrix} = \begin{pmatrix} c_{81} \\ c_{82} \\ c_{83} \\ c_{84} \end{pmatrix}^T \tag{34}$$

A.2. Computation of repair fragments by the repair nodes

The newcomer communicates to repair node i the coefficient ν_i, with $i \in I$.
Node i uses ν_i to compute the repair fragment λ_i as $\lambda_i = (\nu_i, 1)(p_{i1}, p_{i2})^T$ (c.f. Fig. 5).
Node i uploads fragment λ_i to the newcomer.

A.3. Newcomer computes (p_{81}, p'_{82})

Then the newcomer uses the coefficients δ_i, ρ_i computed previously to compute the two fragments (p_{81}, p'_{82}) as,
$p_{81} = (\delta_1, \delta_2, \delta_4, \delta_5, \delta_7)(\lambda_1, \lambda_2, \lambda_4, \lambda_5, \lambda_7)^T$,
$p'_{82} = (\rho_1, \rho_2, \rho_4, \rho_5, \rho_7)(\lambda_1, \lambda_2, \lambda_4, \lambda_5, \lambda_7)^T$.
The repair is done.

A.4. Update of the matrix U

The second fragment p_{82} can not be exactly repaired. For this reason the coefficients $u_{81}, u_{82}, u_{83}, u_{84}$ need to be updated as follows:

$$\begin{pmatrix} u'_{81} \\ u'_{82} \\ u'_{83} \\ u'_{84} \end{pmatrix}^T = \begin{pmatrix} \rho_1\nu_1 \\ \rho_2\nu_2 \\ \rho_4\nu_4 \\ \rho_5\nu_5 \\ \rho_7\nu_7 \end{pmatrix}^T \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ c_{71} & c_{72} & c_{73} & c_{74} \end{pmatrix} + \begin{pmatrix} \rho_1 \\ \rho_2 \\ \rho_4 \\ \rho_5 \\ \rho_7 \end{pmatrix}^T \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ u_{71} & u_{72} & u_{73} & u_{74} \end{pmatrix} \tag{35}$$

We see in Eq. (35) that the evolution of the matrix U not only depends on the matrix U itself but also on the matrix C. A closer look at this example also helps understand why we can not do an exact repair of the second fragment: From Eq. (34) and Eq. (35), we have only 7 variables: 5 for ν_i and a single one in δ_i and a single one ρ_i because all the other δ_i and ρ_i can be expressed in terms of this one variable by Eq. (32) and

Eq. (33). However, for exact repair we must fulfill 8 equations (4 in Eq. (34) and 4 in Eq. (35)), which is not possible. Since this set of 8 equations can not be solved, we have a contradiction. Therefore the U matrix can not be maintained as is but must be updated.

B. COMPARISON OF MBR AND ERHC

The focus of this paper is to compare different coding techniques that all use the *same amount* of storage space for the redundant data. We did not include MBR in our comparison since it requires more storage space. However, MBR is interesting since it minimizes the repair traffic. For this reason we provide here a short comparison between MBR and ERHC. We assume that the original file is partitioned into k blocks, each block is then partitioned into α fragments, and the repair degree d is set to $d = k+\alpha-1$, as it was for MSR.

B.1. Storage Overhead

In case of ERHC, each node holds a storage block of size

$$S_{SB}^{(ERHC)} = \frac{S_{file}}{k}$$

In case of MBR, each node holds a storage block of size

$$S_{SB}^{(MBR)} = \frac{2dS_{file}}{2kd - k^2 + k} = \frac{2(k+\alpha-1)S_{file}}{k(k+2\alpha-1)}$$

In Fig. 16 we compare the storage requirements for some values for k and α. We see that the storage overhead of MBR is about $1.7 - 1.9$ higher than for ERHC and decreases slightly with increasing α.

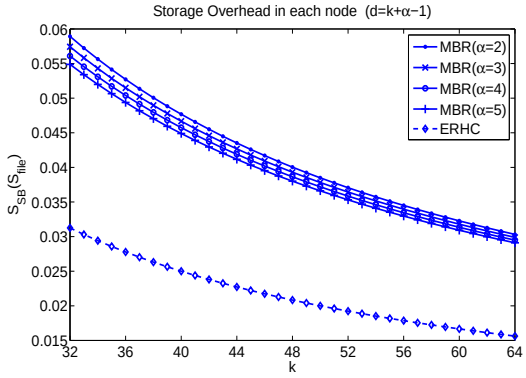


Fig. 16. Storage overhead of MBR and ERHC, $d_0 = 4$

B.2. Repair Traffic

For ERHC the total repair traffic is

$$V_{rp}^{(ERHC)} = \frac{d^{(ERHC)}S_{file}}{k\alpha} = \frac{(d_0+\alpha-1)S_{file}}{k\alpha}$$

For MBR, the total repair traffic is

$$V_{rp}^{(MBR)} = \frac{2dS_{file}}{2kd - k^2 + k}$$

A comparison of the repair traffic is depicted in Fig. 17 for some values of k and d_0, with d_0 being the minimal repair degree for ERHC. We see that MBR starts outperforming ERHC in terms of repair traffic as soon as the minimal repair degree d_0 takes values larger than 2.

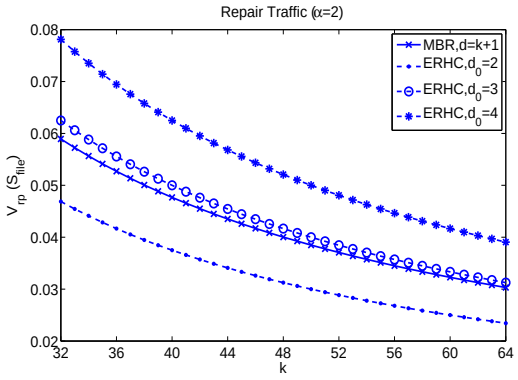


Fig. 17. Repair traffic of MBR and ERHC, $\alpha = 2$

REFERENCES

ADYA, A., BOLOSKY, W., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J., HOWELL, J., LORCH, J., THEIMER, M., AND WATTENHOFER, R. 2002. Farsite: Federated, available and reliable storage for an incompletely trusted environment. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. Vol. 36. 1–14.

BLAKE, C. AND RODRIGUES, R. 2003. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the Usenix Workshop on Hot Topics in Operating Systems*. Vol. 9. Lihue, Hawaii, 1–6.

CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, M., AND KUBIATOWICZ, J. 2006. Efficient Replica Maintenance for Distributed Storage Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*.

CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., M. FRANS KAASHOEK, J. K., AND MORRIS, R. 2006. Proactive replication for data durability. In *Proceedings of the USENIX International Workshop on Peer-to-Peer Systems*.

CHUN, B.-G., RATNASAMY, S., AND KOHLER, E. 2008. Netcomplex: a complexity metric for networked system designs. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. 393–406.

DABEK, F., KAASHOEK, M. F., KARGER, D. R., MORRIS, R., AND STOICA, I. 2001. Wide-area cooperative storage with CFS. In *Proceedings of ACM Symposium on Operating Systems Principles*. 202–215.

DIMAKIS, A. G., GODFREY, P. B., WU, Y., WAINWRIGHT, M. J., AND RAMCHANDRAN, K. 2010. Network coding for distributed storage systems. *IEEE Transactions on Information Theory 56,* 9, 4539–4551.

DIMAKIS, A. G., RAMCHANDRAN, K., WU, Y., AND SUH, C. 2010. A survey on network codes for distributed storage. *CoRR*. http://arxiv.org/abs/1004.4438.

DISCHINGER, M., GUMMADI, K. P., HAEBERLEN, A., AND SAROIU, S. 2007. Characterizing residential broadband networks. In *Proc. of ACM Internet Measurement Conference (IMC)*.

DRUSCHEL, P. AND ROWSTRON, A. 2001. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of USENIX Workshop on Hot Topics in Operating Systems*. 75.

DUMINUCO, A. 2009. Data redundancy and maintenance for peer-to-peer file backup systems. Ph.D. thesis, Telecom ParisTech.

DUMINUCO, A. AND BIERSACK, E. 2009. A Practical Study of Regenerating Codes for Peer-to-Peer Backup Systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*. 376–384.

DUMINUCO, A., BIERSACK, E., AND EN-NAJJARY, T. 2007. Proactive Replication in Distributed Storage Systems Using Machine Availability Estimation. In *Proceedings of ACM Conference on emerging Networking EXperiments and Technologies*.

DUMINUCO, A. AND BIERSACK, E. W. 2009. Hierarchical codes : a flexible trade-off for erasure codes. *Journal of Peer-to-Peer Networks and Applications 2*, 52–66.

GODFREY, B. 2006. Repository of Availability Traces. http://www.cs.berkeley.edu/ pbg/availability/.

HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. 2005. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. Vol. 2. 143–158.

HARDDRIVER. 2010. Cost of hard drive storage space. http://ns1758.ca/winch/winchest.html.

KONDO, D., JAVADI, B., IOSUP, A., AND EPEMA, D. 2010. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. *Proceedings of IEEE/ACM International Conference on Cluster, Cloud and Grid Computing 0*, 398–407.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Vol. 35. 190–201.

LIN, W. K., CHIU, D. M., AND LEE, Y. B. 2004. Erasure Code Replication Revisited. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*. Washington, DC, USA, 90–97.

MITZENMACHER, M. 2004. Digital fountains: A survey and look forward. In *IEEE Information Theory Workshop*. 271–276.

PAMIES-JUAREZ, L. AND BIERSACK, E. 2011. Cost analysis of redundancy schemes for distributed storage systems. *CoRR*. http://arxiv.org/abs/1103.2662.

PAMIES-JUAREZ, L., GARCIA-LOPEZ, P., AND SANCHEZ-ARTIGAS, M. 2010. Availability and Redundancy in Harmony: Measuring Retrieval Times in P2P Storage Systems. In *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*. 1–10.

PLANK, J. S. 2005. Erasure Codes for Storage Applications. In *Proceedings of FAST 2005: USENIX Conference on File and Storage Technologies*.

PLANK, J. S. 2008. A new MDS erasure code for RAID-6. In *Proceedings of FAST 2008: USENIX Conference on File and Storage Technologies*.

RASHMI, K. V., SHAH, N. B., AND KUMAR, P. V. 2010. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. *CoRR abs/1005.4178*. http://arxiv.org/abs/1005.4178.

RASHMI, K. V., SHAH, N. B., KUMAR, P. V., AND RAMCHANDRAN, K. 2009. Explicit construction of optimal exact regenerating codes for distributed storage. In *Proceedings of the annual Allerton conference on Communication, control, and computing*. 1243–1249.

RICHARDSON, T. AND URBANKE, R. 2008. *Modern Coding Theory*. Cambridge University Press, New York, NY, USA.

RODRIGUES, R. AND LISKOV, B. 2005. High Availability in DHTs: Erasure Coding vs. Replication. In *Proceedings of the USENIX International workshop on Peer-To-Peer Systems*. New York.

SHAH, N. B., V., R. K., KUMAR, P. V., AND RAMACHANDRAN, K. 2010. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *CoRR(submitted to the IEEE Transactions on Information Theory)*. http://arxiv.org/abs/1005.1634.

SHOJANIA, H. AND LI, B. 2009. Pushing the envelope: Extreme network coding on the GPU. In *29th International Conference on Distributed Computing Systems (ICDCS)*.

SUH, C. AND RAMCHANDRAN, K. 2010. Exact regeneration codes for distributed storage repair using interference alignment. *CoRR(to be submitted to IEEE Transactions on Information Theory)*. http://arxiv.org/abs/1001.0107.

WEATHERSPOON, H. AND KUBIATOWICZ, J. D. 2002. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proceedings of the USENIX International workshop on Peer-To-Peer Systems*.

WU, Y. AND DIMAKIS, A. G. 2009. Reducing repair traffic for erasure coding-based storage via interference alignment. *Proceedings of the IEEE International Symposium on Information Theory 4*, 2276–2280.

WUALA. 2010. Official website. http://www.wuala.com.

ZHANG, Z., DESHPANDE, A., MA, X., THERESKA, E., AND NARAYANAN, D. 2010. Does erasure coding have a role to play in my data center? Microsoft Research Technical Report MSR-TR-2010-52.