A Practical Study of Regenerating Codes for Peer-to-Peer Backup Systems

Alessandro Duminuco and Ernst Biersack
EURECOM
Sophia Antipolis, France
{*duminuco,biersack*}*@eurecom.fr*

Abstract

In distributed storage systems, erasure codes represent an attractive solution to add redundancy to stored data while limiting the storage overhead. They are able to provide the same reliability as replication requiring much less storage space. Erasure coding breaks the data into pieces that are encoded and then stored on different nodes. However, when storage nodes permanently abandon the system, new redundant pieces must be created. For erasure codes, generating a new piece requires the transmission of k pieces over the network, resulting in a k times higher reconstruction traffic as compared to replication.

Dimakis proposed a new class of codes, called Regenerating Codes, which are able to provide both the storage efficiency of erasure codes and the communication efficiency of replication. However, Dimakis gave only a theoretical description of the codes without discussing implementation issues or computational costs. We have done a real implementation of Random Linear Regenerating Codes that allows us to measure their computational cost, which can be significant if the parameters are not chosen properly. However, we also find that there exist parameter values that result in a significant reduction of the communication overhead at the expense of a small increase in storage cost and computation, which makes these codes very attractive for distributed storage systems.

1. Introduction

P2P(Peer-to-Peer) systems have received a lot of attention in recent years. In particular, the research community has shown an increasing interest in the use of P2P systems for file storage [1], [2], [3], [4]. This application can be very attractive for two main reasons: (i) centralized solutions are expensive (ii) common PCs are equipped with high-capacity local disks, which are often underutilized.

The main challenge in designing storage systems is to guarantee the persistence of the stored data. This is non-trivial because storage peers are not totally reliable: they may face failures, data corruption or accidental data losses.

Adding redundancy to the stored data is the basic tool to achieve data durability in spite of failures of the storing peers. The simplest redundancy scheme is replication, which consists in creating multiple copies of data and spreading them in different locations. A more complex method is represented by erasure codes: the data are processed in order to produce $k + h$ pieces such that any k of them are sufficient to reconstruct the original file, then these pieces are spread across different peers. Previous works have shown [5], [6] how erasure codes are able to provide the same level of reliability as replication with much lower storage requirements.

Redundancy alone is not enough to provide data durability. Since peers might leave permanently the system, some of the initial redundancy might be lost. This means that the number of pieces or replicas present in the system diminishes with time. Periodically this number must be refurbished by the maintenance, which is performed by the means of repairs. A repair consists in rebuilding a lost replica or piece using the available ones. A number of works [5], [7], [8] showed how every piece repaired in erasure codes require that k other available pieces must be read (which corresponds to the size of the original data), while in replication the repair of one replica needs that only one other replica is read. In distributed systems data accesses translate into network transfers, for this reason, under bandwidth constraints, like in P2P systems, erasure codes become impractical and replication is the only feasible solution.

Different solutions have been proposed to couple the storage efficiency of erasure codes with the communication efficiency of replication. Rodrigues and Liskov [5] proposed an hybrid replication/erasure code solution, in which a full replica of the file is held by a special peer, while other peers store erasure coded pieces. Repairs are always performed using the full replica, with a communication cost equal to the replication case. This method, however, introduces an asymmetry in the data maintenance, causing both a higher complexity of the system and a loss in terms of storage efficiency. Duminuco and Biersack [8] proposed a class of codes called Hierarchical Codes, in which the repair communication cost is on average much smaller than for erasure codes. However, Hierarchical Codes have the disadvantage that not all the subsets of k pieces are sufficient to reconstruct the original file. Finally Dimakis et al. [7]

proposed a generalization of traditional erasure codes called Regenerating Codes, for which the communication costs during repair is significantly reduced as compared to Reed Solomon Erasure codes. Dimakis et al. in [7] presented a theoretical framework that allows to prove the existence of these codes while Wu et al. showed in [9] how to build deterministic Regenerating Codes based on linear codes.

However, none of the two papers did investigate the computational cost or propose how to implement such codes. We believe that Regenerating Codes represent a very attractive solution for redundancy schemes in P2P storage systems and deserve a deeper analysis with respect to implementation and deployment, which is the subject of this paper.

In section 2 we provide an introduction to data redundancy schemes in P2P storage systems and recall the main theoretic results on Regenerating Codes ([7]). In section 3 we describe our implementation of Random Linear Regenerating Codes, while we perform in section 4 an analytical evaluation of their cost in terms of storage and computation. In section 5 we test our implementation and evaluate the different cost performance trade-offs. Finally, section 6 concludes the paper.

2. Background

2.1. Data Redundancy Schemes

In this section we give a formal description of the operations performed in a common redundancy scheme. These operations can be grouped in three distinct phases, which follow the life cycle of a generic file that is stored in the system.

1) **Insertion:** The insertion consists in processing the file, creating $(k+h)$ redundant pieces and distributing them over distinct peers. The processing can be as trivial as building replicas of the file[1], or can be a complex coding operation. No matter which redundancy scheme is used, the property of these pieces is that any k of them are sufficient to reconstruct the original file[2].
2) **Maintenance:** Maintenance consists in rebuilding the redundancy lost due to peer failures. Maintenance is performed by the means of repairs. A repair requires the cooperation of d peers that send data to a new peer[3], called newcomer, which in turn processes the received data to obtain a new piece. We refer to d as the **repair degree**. If the repair is correctly executed, the new piece has the same properties as all the others, i.e. with any $(k-1)$ other pieces it forms a set of pieces sufficient to reconstruct the original file.
3) **Reconstruction:** If the owner of the file wants to retrieve it from the system, a reconstruction needs to be performed. The reconstruction consists in downloading data from k peers and processing them to obtain the original file.

In the rest of the paper we refer to the size of the file as $|\mathit{file}|$ and to the size of a piece as $|\mathit{piece}|$ (in general we refer to the size of a generic object x as $|x|$).

From the description it is clear that a redundancy scheme implies three kinds of costs:

1) **Storage:** Redundancy implies that the stored file consumes more storage space than the original file. The storage requirement is easily computed by:
$$|\mathit{storage}| = (k+h) \cdot |\mathit{piece}| > |\mathit{file}|$$
2) **Communication:** All three phases in the life cycle require data to be transferred among peers. At *insertion*, all the pieces must be transferred, which amounts to a volume of $|\mathit{storage}|$. At *maintenance*, for every repair d peers upload each an amount of data equal to $|\mathit{repair}_{up}|$ to the newcomer for a total of $|\mathit{repair}_{down}|$, with the obvious relation: $|\mathit{repair}_{down}| = d \cdot |\mathit{repair}_{up}|$. At *reconstruction*, the file owner needs to download at least an amount of data equal to $|\mathit{file}|$ (See section 3.2 for details).
3) **Computation:** When coding is used, all the three phases described require processing of data[4]. At *insertion*, all the pieces need to be coded with a cost of *CPU(encoding)*. At *repair*, part of the processing is done on the d participating peers, denoted as $\mathit{CPU(repair)}_{up}$ and part is done on the newcomer, denoted as $\mathit{CPU(repair)}_{down}$. At *reconstruction*, the original file must be reconstructed from k pieces with a cost *CPU(reconstruction)*.

The particular redundancy scheme defines how the redundant data are generated and handled and what is the cost in terms of computation, communication, and storage. As an example let us consider traditional erasure codes (like Reed-Solomon codes [10]). For these codes, the following two constraints hold w.r.t. the repair degree d and the piece size:

$$\begin{aligned} d &= k \\ |\mathit{piece}| &= |\mathit{file}|/k \end{aligned} \quad \text{(E1)}$$

which means that every repair is performed collecting data from $d = k$ existing peers and that every peer stores an amount of data equal to $1/k$ of the file size. It can be shown that given these constraints, *the amount of data that needs to be transferred from every participating peer to the newcomer is equal to the size of a piece, which means that in total an amount equivalent to the size of the whole file will be transmitted*. In terms of maintenance, the communication costs are: $|\mathit{repair}_{up}| = |\mathit{piece}|$ and $|\mathit{repair}_{down}| = |\mathit{file}|$.

1. This is the replication case where $k = 1$.

2. There exist redundancy schemes, in which this property is not satisfied like in [8], but these schemes are not of interest in this work.

3. A new peer is a peer that at the moment not storing any piece of the file.

4. In case of replication there is no processing.

Note that this means that for every new bit that we create during a repair, k existing bits needs to be transferred.

The computation costs are implementation dependent (see section 4 for details).

2.2. Description of Regenerating Codes

In this section we give a quick overview of the main properties of Regenerating Codes from [7]. In essence Regenerating Codes try to address the following question: what is the impact on the communication cost if we relax the constraints defined for traditional erasure codes given in eq. E1?

Given k and h, Regenerating Codes can take $k \cdot h$ different values for the pair of parameters $(d, |\mathit{piece}|)$. In fact Regenerating Codes can be considered a generalization of traditional erasure codes, which trade-off increased storage cost for reduced communication cost.

More formally, a generic Regenerating Code denoted by $\mathit{RC}(k, h, d, i)$, sets the following constraints on the repair degree d and the piece size:

$$\begin{array}{rcll} d & \in & [k, k+h-1] & \\ |\mathit{piece}| & = & p(d,i) \cdot |file| & i \in [0, k-1] \end{array} \quad \text{(E2)}$$

Given a repair degree d, the parameter i, which we define as the **piece expansion index**, determines the piece size through the function $p(d, i)$, which is defined[5] as:

$$p(d,i) = 2\frac{d-k+i+1}{2k(d-k+1)+i(2k-i-1)}$$

It can be proved that $\mathit{RC}(k, h, d, i)$ requires that each of the d peers participating to a repair transfer to the newcomer an amount of data **at least** equal to

$$|\mathit{repair}_{up}| = r(d,i) \cdot |\mathit{file}|$$

where $r(d, i)$ is defined as:

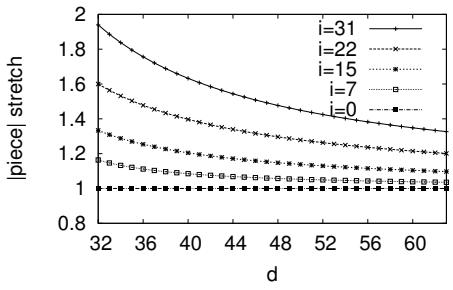
$$r(d,i) = \frac{2}{2k(d-k+1)+i(2k-i-1)}$$

consequently $|\mathit{repair}_{down}| = d \cdot r(d,i) \cdot |\mathit{file}|$.

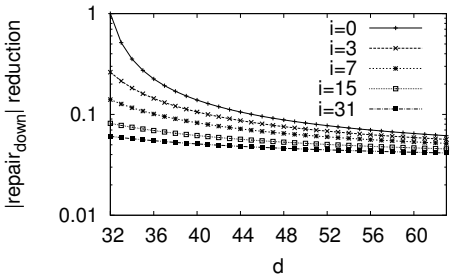
In this paper we fix the values for $k = 32$ and $h = 32$, which allows the system to sustain up to 32 losses. We consider this reasonable under the massive churn we may observe in an Internet scenario [3]. However, results with other parameters show the same trends.

Fig. 1 depicts how the piece size $|\mathit{piece}|$ and the volume of repair traffic $|\mathit{repair}_{down}|$ evolve as a function of d and i for a code with $k = 32$ and $h = 32$. In particular all the values are *relative* to the piece size and the volume of repair traffic required by a traditional erasure code, which in the framework of Regenerating Codes would correspond to $\mathit{RC}(32, 32, 32, 0)$, i.e. with $d = 32$ and $i = 0$. As described in the previous section these reference values are: $|\mathit{piece}| = |\mathit{file}|/32$ and $|\mathit{repair}_{down}| = |\mathit{file}|$.

5. We reformulate the expressions given in [7] in a different way to facilitate the successive computations.



(a) $|\mathit{piece}|$



(b) $|\mathit{repair}_{down}|$

Figure 1. Size of the pieces and repair communication cost (in log-scale) normalized by the reference values of a traditional erasure code, for $\mathit{RC}(32, 32, d, i)$.

We see that moving to larger repair degree d and to larger piece size (increasing the piece expansion factor i) it is possible to obtain *an impressive reduction of the repair traffic*. Authors in [7] identify two notable cases for the values $i = 0$ and $i = k - 1$. For $i = 0$, the size of the pieces stays constant at the minimum possible size and the codes are called **Minimum Storage Regenerating codes** (MSR). For $i = k - 1$, repair traffic is minimized and the codes are called **Minimum Bandwidth Regenerating codes** (MBR).

3. Random Linear Implementation

Existing works on Regenerating Codes [7], [9] present the theoretic framework that supports the construction of Regenerating Codes and give an intuition for a possible implementation based on random linear codes, without providing details. We propose a precise description of such an implementation and discuss its practical implications.

3.1. Traditional Random Linear Codes

Let us first explain how Random Linear Codes work for the case of traditional erasure codes. The essence of random linear codes is that all the operations are linear operations

over a Galois Field on fixed sized data fragments. Again we describe these operations following the life cycle of a file.

1) **Insertion:** In this phase we have to create $k+h$ pieces of size $|file|/k$. To do that, it is enough to break the file in $n_{file} = k$ equal sized (original) fragments, and compute any of the $k + h$ pieces as a random linear combination of them. The random coefficients used for such combinations are stored along with the pieces.
2) **Maintenance:** As already explained, a repair in traditional erasure codes requires the transfer of the whole piece from d participating peers to the newcomer. The newcomer then builds the new piece performing a random linear combination of the d received pieces. Again the resulting coefficients are stored along with the new piece.
3) **Reconstruction:** The owner of the file downloads k pieces from k other peers and uses these pieces to reconstruct the file. The procedure consists of inverting, if possible, the matrix composed by the coefficients of all the received pieces, and multiplying the inverted matrix by the pieces. The results of such a multiplication are the original fragments, i.e. the original file.

Theory on Random Linear Network Codes [11], [12], [13] says that the probability to successfully invert the matrix upon reconstruction depends only on the size of the Galois Field and that this probability can be made arbitrarily close to 1 by increasing the size of the Galois Field. For all practical purposes a field size equal to 2^{16} is considered sufficient.

3.2. Random Linear Regenerating Codes

In traditional erasure codes, random linear implementation is straightforward, because all the operations are performed on a set of pieces, which means that the size of the piece can be used as the basic unit of information in all the linear combinations and in the decoding.

In Regenerating Codes things are different because they allow that the amount of data stored $|piece|$, is not necessarily equal to the amount of data transmitted by a participant upon a repair $|repair_{up}|$ and that the amount of data downloaded by a newcomer $|repair_{down}|$ is not a multiple of $|piece|$.

In other words the basic unit of information, which is the size of the fragments we break the original file into, cannot be the size of the piece anymore. If we denote this size as $|fragment|$, we can write the constraints it has to fulfill:

$$\begin{aligned} |file| &= n_{file} \cdot |fragment| \\ |piece| &= n_{piece} \cdot |fragment| \\ |repair_{up}| &= n_{repair} \cdot |fragment| \end{aligned} \tag{E3}$$

where n_{file}, n_{piece} and n_{repair} are integers. Using equations in section 2.2, we can compute:

$$\frac{|piece|}{|repair_{up}|} = \frac{p(d,i)}{r(d,i)} = d - k + i + 1$$

and:

$$\frac{|file|}{|repair_{up}|} = \frac{1}{r(d,i)} = \frac{2k(d-k+1) + i(2k-i-1)}{2}$$

Both ratios are integers. This means that we can set $n_{repair} = 1$, which corresponds to setting $|fragment| = |repair_{up}|$, and consequently:

$$\begin{aligned} n_{file} &= \frac{2k(d-k+1)+i(2k-i-1)}{2} \\ n_{piece} &= d - k + i + 1 \end{aligned} \tag{E4}$$

Given these parameters we can describe the operations needed in Random Linear Regenerating Codes:

1) **Insertion:** We break the file in n_{file} equal sized original fragments, and compute any of the $k + h$ pieces as n_{piece} random linear combinations of them. The random coefficients used for such combinations are stored along with the piece. They form a (n_{piece}, n_{file}) matrix[6].
2) **Maintenance:** A repair involves d existing peers, which send data to the newcomer. The data sent by any of the d peers correspond to the results of one random linear combination of the n_{piece} fragments contained in the stored piece, as depicted in figure Fig. 2(a). The newcomer receives thus d fragments and the corresponding coefficients and obtains its new piece as n_{piece} random linear combinations of them, as depicted in Fig. 2(b). Note that in the particular case of $d = n_{piece}$ the newcomer does not need to perform linear combinations of the received fragments, since they constitute already the new piece.
3) **Reconstruction:** The owner of the file downloads k pieces from k peers, which correspond to $n_{piece} \cdot k$ fragments, along with the coefficients which form a $(n_{piece} \cdot k, n_{file})$ matrix. It tries to find n_{file} independent rows in the coefficient matrix, then it inverts the resulting square submatrix and finally multiplies this matrix by the concerned fragments. An important remark is that if the file owner downloads k pieces, it potentially downloads an amount of data quite bigger than the file size. In [7] it is claimed that this can represent a significant drawback for Regenerating Codes. In our implementation, we eliminated this shortcoming: we download only the coefficients, we extract a full-rank square submatrix, we invert it, and finally we download *only the* n_{file} *fragments corresponding to the invertible submatrix* that was extracted. In this way we download always an amount of data equal to the file size, without paying any extra-cost.

6. In our notation a (n, m) matrix is a matrix with n rows and m columns.

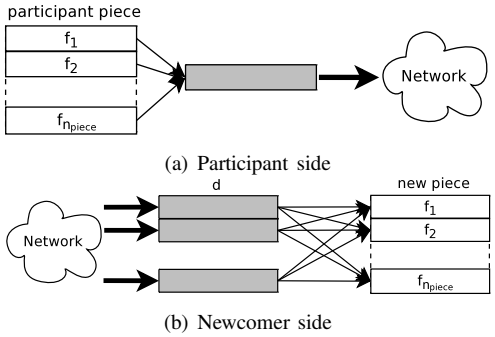


Figure 2. Repair scheme on the participant side and on the newcomer side. Every arrow indicates a participation to a random linear combination.

4. Analytical Evaluation

In this section we perform an analytical evaluation of the Random Linear Regenerating Codes. To do this, we give a formal description of the linear operations performed. All the data we handle can be interpreted as a sequence of values, called elements, in a given Galois Field. Usually the size of such field is chosen to be equal to 2^q, since this speeds up the computation. In this case every value is a sequence of q bits, a common choice is $q = 16$, which corresponds to an element size of 2 bytes. Every fragment is thus represented by a vector of $l_{frag} = (|fragment|/q)$ elements. The whole file is thus represented by a (n_{file}, l_{frag}) matrix, denoted as $F_{n_{file},l_{frag}}$. A set of n encoded fragments is represented as a (n, l_{frag}) matrix $E_{n,l_{frag}}$[7], this matrix can be always represented as a set of linear combinations of the original fragments:

$$f_{n,l_{frag}} = C_{n,n_{file}} F_{n_{file} \times l_{frag}}$$

where $C_{n,n_{file}}$ are elements in the field and represent the coefficients associated with the set of fragments.

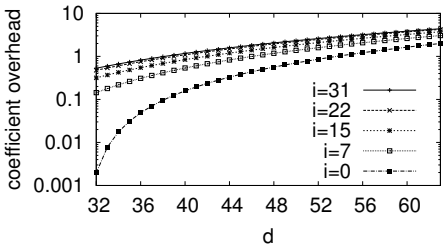


Figure 3. Coefficient overhead (in log-scale) of $RC(32, 32, d, i)$ for a 1 MByte file.

7. In our notation $F_{n,m}$ corresponds to a set of original fragments, while $E_{n,m}$ to a set of encoded fragments.

4.1. Impact of coefficients

The first question we address is the impact of the coefficients in the storage and in the communication costs. Since with every fragment, we associate a set of n_{file} coefficients, the relative impact of the coefficients is given by the ratio:

$$r_{coeff} = \frac{n_{file} q}{|fragment|} = \frac{n_{file}^2}{|file|} \cdot q$$

this ratio can be interpreted as the overhead due to coefficients: for every bit of data we need r_{coeff} bits of coefficients. Note that this ratio is inversely proportional with the size of the file we store, this means, as one could expect, that the bigger the file the smaller is the coefficient overhead. More importantly, the overhead increases with the square of n_{file}, which increases significantly as we increase the parameters d and i in Regenerating Codes (see eq. E4).

To understand the impact of this additional cost, let us consider the class of regenerating codes $RC(32, 32, d, i)$ and let us assume that the field size is $q = 16$, which corresponds to an element size of 2 bytes. In Fig. 3 we plot the values of the coefficient overhead when the original file size is $|file|$=1 MByte for all the possible values of d and i.

For such a small file size, the coefficient overhead is not negligible: in the 'most expensive' configuration *for 1 bit of data, more than 4 bits of coefficients are needed*, which is clearly unacceptable. By increasing the file size, this overhead decreases[8]. The implication of figure 3 is that when using Regenerating Codes, system designers need to choose a minimum size for storage objects that is significantly bigger than for traditional erasure codes.

4.2. Computational Complexity

One of the main concerns in the employment of coding in real systems is the computational effort that they require. In this section we propose a formal analysis of Random Linear Regenerating Codes.

All the operations are performed in Galois Fields. Therefore, we need to make sure to control the cost of the operations by choosing the right field size. If we set the field size equal to 2^q, with $q = 16$ all the operations are performed on unsigned short integers (2 bytes). In this case

- Additions and subtractions correspond to an XOR operation between two elements.
- Multiplication and division are performed in the log-space. For example: $a \cdot b$ becomes $\exp(\log a + \log b)$. $\log$ and $\exp$ for all the possible values in the field are computed *offline* and stored, which requires 256 KB of memory for $q = 16$. The operations $\log$ and $\exp$ can then be implemented as value lookups in a vector,

8. The actual overhead is given by the values shown in figure 3 divided by the file size in MBytes

which allows to implement divisions and multiplication in 3 lookups and 1 addition.

All the operations we perform in Regenerating Codes can be reduced to: (1) Linear Combinations and (2) Matrix inversions. Let us analyze them in details:

1) A linear combination of n vectors of length l consists in $n \cdot l$ additions and $n \cdot l$ multiplications for a total of $5nl$ operations.
2) The inversion of a square (n, n) matrix consists in n^3 additions and n^3 multiplications that can be implemented $5n^3$ operations. Actually for Regenerating Codes the situation is slightly different: we have a (m, n) matrix, $m \geq n$ from which we need to *extract* n rows that are linearly independent, which will result in a (n, n) submatrix that can then be inverted. Extraction and inversion are done in parallel and the cost will vary accordingly to the particular matrix between the bounds $5n^3$ and $5mn^2$.

Now we have all the basic tools to compute the complexity of Regenerating Codes along the lifetime of a file:

1) **Insertion:** In this phase we perform $(k+h) \cdot n_{piece}$ linear combinations of n_{file} fragments for a total number of operations equal to:

$$CPU(encoding) = 5(k+h) \cdot n_{file} \cdot n_{piece} \cdot l_{frag}$$

Using the definitions of the different parts we obtain:

$$CPU(encoding) = \frac{5}{2}(k+h) \cdot n_{piece} \cdot |file| \quad \text{(E5)}$$

2) **Maintenance:** As already explained, in a repair, part of the work is done on the participating peers and another part is done on the newcomer. On every participating peer we perform one linear combination of n_{piece} fragments, which corresponds to:

$$CPU(repair)_{up} = 5 \cdot n_{piece} \cdot l_{frag}$$

doing some manipulations we obtain that the number of operations is proportional to the size of the piece expressed in bytes:

$$CPU(repair)_{up} = \frac{5}{2} \cdot |piece| \quad \text{(E6)}$$

On the newcomer we perform n_{piece} linear combinations of d fragments, which corresponds to:

$$CPU(repair)_{down} = 5 \cdot d \cdot n_{piece} \cdot l_{frag} = d \cdot CPU(repair)_{up} \quad \text{(E7)}$$

Note that every fragment is also associated with a set of coefficients. This means that every time that a new fragment is generated as a linear combination of other existing fragments, this operation must be performed also on the correspondent coefficients, in order to obtain the coefficients associated with the new fragment. In terms of computation cost, this can be taken into account assuming that the fragment size is virtually increased by the size of coefficients, which is given by the overhead in section 4.1.

3) **Reconstruction:** We can split the reconstruction in two phases: (1) we need to extract n_{file} linear independent rows from a $k \cdot n_{piece} \times n_{file}$ matrix, and then invert the obtained submatrix (2) We multiply this submatrix by the correspondent encoded fragments. According to these two phases, the cost of reconstruction can be split in two components as well:

$$CPU(reconstruction) = CPU(inversion) + CPU(decoding)$$

As explained before the cost of the inversion is bounded by two limits:

$$5 \cdot n_{file}^3 < CPU(inversion) < 5 \cdot k \cdot n_{piece} \cdot n_{file}^2 \quad \text{(E8)}$$

The decoding, then, corresponds to n_{file} linear combinations of n_{file} fragments, which leads to:

$$CPU(decoding) = 5 \cdot n_{file}^2 \cdot l_{frag} = \frac{5}{2} \cdot n_{file} \cdot |file|$$

Note that all the costs, except from the inversion cost, are linearly dependent to the file size $|file|$ (This holds also for repair, since $|piece|$ is in turn proportional to $|file|$).

5. Experimental Evaluation

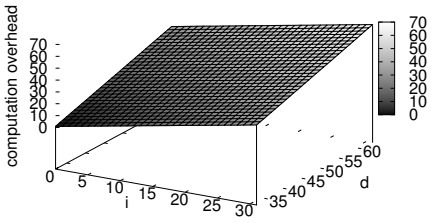
In this section we evaluate the resource requirements of Regenerating Codes. For this purpose, we wrote an optimized C implementation of Random Linear Regenerating Codes that we execute on an Intel Core 2 Duo CPU at 2.66GHz.

We execute all the operations performed in the life cycle of a stored file, as described in section 4, and measure the time needed to perform these operations. All the experiments have been done for a file of 1 MByte in size and the Regenerating Code parameters are fixed to $k = 32$, $h = 32$, and can take all possible values for i and d.

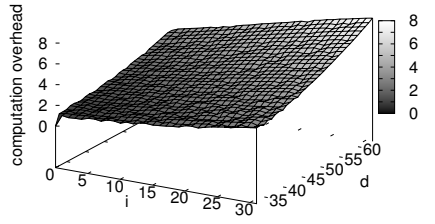
5.1. Computational Cost

To have a basis for comparing different configurations of Regenerating Codes, we first show the results obtained for a traditional erasure code, (i.e. a Regenerating Code with $RC(32, 32, 32, 0)$) when a file of 1 MByte is stored. Let $t_{d,i}$ denote the time needed by a particular operation for a Regenerating Code $RC(32, 32, d, i)$. The following table shows the time $t_{32,0}$ needed for each operation:

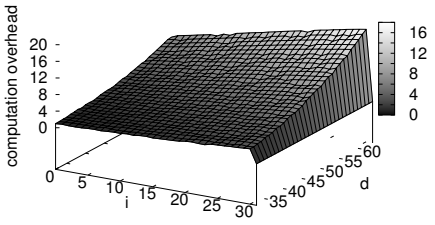
	$t_{32,0}[sec]$
Encoding	0.52
Participant Repair	0
Newcomer Repair	0.01
Matrix Inversion	0.002
Decoding	0.25



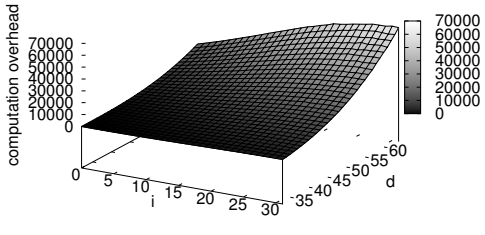
(a) Encoding



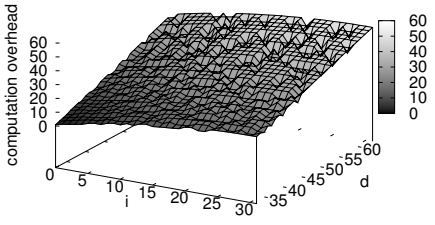
(b) Repair: Participant side.



(c) Repair: Newcomer side.



(d) Reconstruction: Matrix Inversion.



(e) Reconstruction: Decoding.

Figure 4. Computation overhead for $RC(32, 32, d, i)$.

Note that the participant repair has a computation time of zero because in traditional erasure codes repairs do not require any computation at the participant side, which simply sends to the newcomer the entire piece.

Let us now introduce the results obtained for the general case of Regenerating Codes $RC(32, 32, d, i)$. To understand the computational overhead of these codes, we consider the ratio between the time $t_{d,i}$ and the time $t_{32,0}$ measured for traditional erasure codes. We call this ratio **computation overhead** r_{cpu}:

$$r_{cpu} = \frac{t_{d,i}}{t_{32,0}}$$

The computation overhead tells us how much a given Regenerating Code is slower than a traditional erasure code. Following the life cycle of a file we have:

1) **Insertion:** We show in Fig. 4(a) the computation overhead of the initial encoding of the file. We see that the overhead grows linearly with i and d. This is consistent with eq. E5, which says that the cost is proportional to n_{piece}, which is in turn linear with d and i as we can see from eq. E4.
2) **Maintenance:** Fig. 4(b) shows the computation overhead on the participant side[9], in this case the computation overhead grows slightly more than linearly with d and i, since as we know from eq. E6 it is proportional to the piece size, which in turn has the behavior shown in Fig. 1(a). Fig. 4(c) shows the computation overhead on the newcomer side. From eq. E7, this cost is proportional to d times the cost on the participant side, which is confirmed by the roughly quadratic relation with d shown by the plot. Note that for $i = k - 1$ the overhead falls to zero, since for this configuration the newcomer does not need to combine the received blocks, but simply stores them (c.f. section 3.2).
3) **Reconstruction:** The reconstruction requires the inversion of the matrix coefficients and then the decoding of the fragments. Fig. 4(d) shows the computation overhead for the inversion, which as we know from eq. E8 grows roughly as n_{file}^3. Inversion can be computationally very expensive, in particular for large values of d and i. Fig. 4(e) shows the computation overhead of the decoding, whose shape closely resembles the one for encoding (see Fig. 4(a)), which is expected since both perform analogous operations.

5.2. Bottleneck Network Bandwidth

As outlined in section 2.1, a redundancy scheme introduces three different costs, namely computation, storage

9. Note that this cost is equal to zero in traditional erasure codes, for this reason the normalization is done by the smallest value larger than zero which occurs for $d = 33$ and $i = 0$ and is equal in terms of computation time to 0.0003 sec.

d	i	Bottleneck Network Bandwidth					Communication	Storage
		Encoding	Repair		Reconstruction		$\|repair_{down}\|$	$\|storage\|$
			Participant	Newcomer	Matrix Inversion	Decoding		
32	0	31.2 Mbps	∞	777.3 Mbps	7.8 Mbps	24.6 Mbps	1MB	2 MB
63	30	655 Kbps	11.0 Mbps	10.2 Mbps	383 Kbps	482 Kbps	42.47 KB	2.61 MB
32	30	1.9 Mbps	21.6 Mbps	21.6 Mbps	1.6 Mbps	1.3 Mbps	62.18 KB	3.76 MB
40	1	3.1 Mbps	70.5 Mbps	76.8 Mbps	1.5 Mbps	2.5 Mbps	128.40 KB	2.006 MB

Table 1. Resource requirements of $RC(32, 32, d, i)$ for a 1 MByte file.

and communication. So far we have only considered computation. However, what we are really interested in is to evaluate which resource (computation or communication) is the overall performance bottleneck of the system.

In a distributed storage system the data handled must be *transferred over the network*. Let us assume that the transfer operation is pipelined with the coding, which means in the case of insertion that each fragment is transmitted as soon as it is produced by the initial encoding step. If the transfer takes longer than the computation, then the *bottleneck is communication*, and the use of a computationally more efficient code will not make the insertion operation faster. This means that whether or not computation has an impact on the overall performance of the system depends on the available network bandwidth of the participating peers. For this purpose we want to know the minimum network bandwidth of a peer, for which the computation represents the bottleneck for the overall performance. We call this bandwidth **bottleneck network bandwidth**, which is denoted as *bnb*.

The bottleneck network bandwidth can be computed as the bandwidth for which the transfer time is equal to the computation time. If $t_{d,i}$ denotes the time needed to perform an operation for $RC(32, 32, d, i)$ and $|data|_{d,i}$ denotes the amount of data handled by that operation that need to be transmitted over the network. We have:

$$bnb_{d,i} = \frac{|data|_{d,i}}{t_{d,i}}$$

From the above definition it is clear that the bottleneck network bandwidth also gives the *amount of data that can be processed by the coding/decoding operation*.

The values of $|data|_{d,i}$ for the different operations are computed as follows:

- **Encoding:** This operation produces the $(k + h)$ initial pieces. The amount of data produced that is sent over the network is given by the size of these pieces: $|data| = (k + h) \cdot |piece|$.
- **Participant Repair:** This operation produces a single fragment plus the corresponding coefficients. The amount of data that is sent over the network is: $|data| = (1 + r_{coeff}) \cdot |fragment|$.
- **Newcomer Repair:** This operation produces a new piece and his coefficients from d received fragments and their coefficients. The amount of data that is *received* from the network is given by the size of d fragments plus the corresponding coefficients: $|data| = (1 + r_{coeff}) \cdot d \cdot |fragment|$.
- **Inversion:** This operation extracts n_{file} independent rows form the received $(k \cdot n_{piece}, n_{file})$ matrix (which describes the k pieces used for reconstruction), and inverts the submatrix obtained. This means that the amount of data that is received for this operation is given by the size of the coefficients of the k pieces: $|data| = k \cdot r_{coeff} \cdot |piece|$.
- **Decoding:** This operation produces the original file by multiplying the matrix obtained from the inversion by the correspondent encoded fragments. The amount of data that is received for this operations is given by the size of n_{file} fragments, i.e. the file size: $|data| = |file|$.

Table 1 shows the bottleneck network bandwidth for all the operations in the life cycle of a file for different values of d, i. The last two columns show the volume of repair traffic $|repair_{down}|$ and the total amount of data stored in the system $|storage|$.

The first row with $d = 32, i = 0$ presents the results for a traditional erasure code, which minimizes the *storage* requirement at the expense of a very large volume of repair traffic ($|repair_{down}| = |file|$). In row two we consider a code with $d = 63$ and $i = 30$, which *minimizes the repair traffic*. However, as we know from figure 4 this particular code has the highest computational costs, which result in bottleneck network bandwidth values that can be as low as a few hundred Kbps.

However, if we remember the results presented in Fig. 1(b), which shows the savings in repair traffic for Regenerating Codes, we know that most of the savings are already achieved by quite small values of d, i.e. where $d = k$ or where d is slightly larger than k. For this reason, the next two rows of table 1 consider Regenerating Codes with values of $d = 32$ and $d = 40$ that illustrate how we can trade off storage requirement and repair traffic:

- If we have plenty of storage space, we can use a big value for the piece expansion index i: For $d = 32, i = 30$ the storage space required as compared to the one required by traditional erasure codes almost doubles. However, the reduction in repair traffic as compared to traditional erasure codes is still almost as good as for the Regenerating Code with $d = 63, i = 30$, which minimizes the repair traffic.
- On the other hand if storage space matters, we can choose a code with a small i, and a d slightly larger

than k, which still preserves most of reduction in repair traffic. Results for $d = 40, i = 1$ are shown in the fourth row of table 1. If we compare the results to the best one achievable for each resource (see first two rows), we see that we achieve a close to minimal storage requirement (2.006 MB vs. 2.0 MB), and a repair traffic (128.40 KB) that is almost one order of magnitude less than for traditional erasure codes.

From the results presented so far, we can conclude that Regenerating Codes, as compared to traditional erasure codes, can provide substantial reductions in repair traffic, at almost no extra cost in terms of storage space required. However, this gain comes at the price of a much higher computational cost as can be seen when looking at the encoding and reconstruction performance, which is nearly one order of magnitude lower than for traditional erasure codes. With the current implementation, we can encode/decode in the order of 1 GByte of data per hour.

This performance may be too low for a large data center. However, we feel that Regenerating Codes are best suitable for those systems that do not insert or retrieve very large amounts of data and that need to do a significant amount of repairs: An example is given by peer-to-peer data backup systems where the data maintenance due to the high node churn, is far more frequent than data insertion or retrieval.

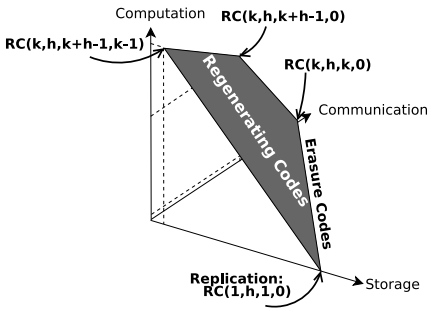


Figure 5. Illustration of the trade-offs provided by Regenerating Codes.

6. Conclusion

Regenerating Codes can be seen as a generalization of previously known redundancy schemes based on replication and erasure codes. They allow to trade off not only communication and storage requirements, but also computational costs. We schematically depict this trade-off in figure 5.

We proposed a practical implementation of Regenerating Codes, based on Random Linear Codes. We presented and evaluated its performance trade-offs. We saw that the important savings provided in terms of repair traffic do not come for free, as Regenerating Codes have much lower coding and decoding rates.

However, we feel that Regenerating Codes have a lot of potential in environments where repairs are frequent and the available bandwidth to carry repair traffic is limited, as is for instance the case in Internet-wide peer-to-peer backup systems.

As future work, we plan to deploy Random Linear Regenerating Codes in a real P2P storage system. We want to compare the performance of Regenerating Codes to other existing solutions, in particular traditional erasure codes and Hierarchical Codes[8], under different conditions with respect to data volume and available bandwidth.

References

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer, "Farsite: Federated, available and reliable storage for an incompletely trusted environment," in *OSDI*, 2002.

[2] F. Dabek *et al.*, "Wide-area cooperative storage with CFS," in *SOSP*, 2001.

[3] A. Haeberlen, A. Mislove, and P. Druschel, "Glacier: Highly durable, decentralized storage despite massive correlated failures," in *NSDI*, 2005.

[4] H. Weatherspoon, "Design and evaluation of distributedwide-area on-line archival storage systems," Ph.D. dissertation, University of California, Berkeley, 2006.

[5] R. Rodrigues and B. Liskov, "High availability in DHTs: Erasure coding vs.replication," in *IPTPS*, 2005.

[6] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *IPTPS*, 2002.

[7] A. G. Dimakis, B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *Computer Research Repository (CoRR)*, vol. arXiv:0803.0632v1 http://arxiv.org/abs/0803.0632, Mar. 2008.

[8] A. Duminuco and E. Biersack, "Hierarchical codes: How to make erasure codes attractive for peer-to-peer storage systems," in *IEEE P2P*, 2008.

[9] Y. Wu, A. Dimakis, and K. Ramchandran, "Deterministic regenerating codes for distributed storage," in *Annual Allerton Conference*, 2007.

[10] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software – Practice & Experience*, vol. 27, no. 9, pp. 995–1012, September 1997.

[11] S. Acedacnski, S. Deb, M. Medard, and R. Koetter, "How good is random linear coding based distributed networked storage?" in *NETCOD*, 2005.

[12] S.-Y. R. Li, R. W. Yeung, and N. Cai, "Linear network coding," *IEEE Transactions on Information Theory*, vol. 49, no. 2, February 2003.

[13] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, "Network information flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, July 2000.