Simple Regenerating Codes: Network Coding for Cloud Storage

Dimitris S. Papailiopoulos[†], Jianqiang Luo[‡], Alexandros G. Dimakis[†], Cheng Huang[*], and Jin Li[*]
[†]University of Southern California, Los Angeles, CA 90089, Email:{papailio, dimakis}@usc.edu
[‡]Wayne State University, Detroit, MI 48202, Email: jianqiang@wayne.edu
[*]Microsoft Research, Redmond, WA 98052, Email: {cheng.huang, jinl}@microsoft.com

***Abstract*—Network codes designed specifically for distributed storage systems have the potential to provide dramatically higher storage efficiency for the same availability. One main challenge in the design of such codes is the exact repair problem: if a node storing encoded information fails, in order to maintain the same level of reliability we need to create encoded information at a new node. One of the main open problems in this emerging area has been the design of simple coding schemes that allow exact and low cost repair of failed nodes and have high data rates. In particular, all prior known explicit constructions have data rates bounded by $1/2$.**

In this paper we introduce the first family of distributed storage codes that have simple look-up repair and can achieve arbitrarily high rates. Our constructions are very simple to implement and perform exact repair by simple XORing of packets. We experimentally evaluate the proposed codes in a realistic cloud storage simulator and show significant benefits in both performance and reliability compared to replication and standard Reed-Solomon codes.

I. Introduction

Distributed storage systems have reached such a massive scale that recovery from failures is now part of regular operation rather than a rare exception [23]. Large scale deployments typically need to tolerate multiple failures, both for high availability and to prevent data loss. Erasure coded storage achieves high failure tolerance without requiring a large number of replicas that increase the storage cost [9]. Three application contexts where erasure coding techniques are being currently deployed or under investigation are Cloud storage systems, archival storage, and peer-to-peer storage systems like Cleversafe and Wuala (see e.g. [2], [3], [5], [8], [12])

One central problem in erasure coded distributed storage systems is that of maintaining an encoded representation when failures occur. To maintain the same redundancy when a storage node leaves the system, a *newcomer* node has to join the array, access some existing nodes, and exactly reproduce the contents of the departed node. Repairing a node failure in an erasure coded system requires in-network combinations of coded packets, a concept called network coding. Network coding has been investigated for numerous applications including p2p systems, wireless ad hoc networks and various storage problems (see e.g. [6], [7], [15]).

In this paper we focus on network coding techniques for exact repair of a node failure in an erasure coded storage system [4], [2]. There are several metrics that can be optimized during repair: the total information read from existing disks during repair [11], [12], the total information communicated in the network [14], [16]–[22] (called repair bandwidth [4]), or the total number of disks required for each repair [8], [13].

Currently, the most well-understood metric is that of repair bandwidth. For designing (n, k) erasure codes that have n storage nodes and can tolerate any $n-k$ failures, an information theoretic tradeoff between the repair bandwidth γ and the storage per node α was established in [4], using cut-set bounds on an information flow graph. Explicit code constructions exist for the the two extreme points on this bandwidth-storage tradeoff, see e.g. [2], [5]. Despite this substantial amount of prior work, there are no practical code constructions of efficiently repairable codes with data rates above $1/2$. Further, different performance metrics might be of interest in different applications. It seems that for cloud storage applications the main performance bottleneck is the disk I/O overhead for repair, which is proportional to the number of nodes d involved in rebuilding a failed node.

Our Contribution: In this paper we introduce the first family of distributed storage codes that have simple look-up repair and can achieve arbitrarily high rates. Our constructions are very simple to implement and perform exact repair by simple packet combinations. Specifically, we design simple regenerating codes (SRC) that have high-rate, very small disk-I/O d, and minimal repair computation.

An (n, k, f)-SRC is a code for n storage nodes that can tolerate $n-k$ erasures, where each node stores a fraction $\frac{f+1}{fk}$ of the file size in coded chunks. To repair a single coded chunk we need to access f disks and read 1 chunk from each disk. The regeneration of an entire lost node costs a fraction $\frac{f+1}{k}$ in repair bandwidth and $d = 2f$ disk accesses. Our codes have rate $R = \frac{f}{f+1}\frac{k}{n}$, which can be made arbitrarily close to $\frac{f}{f+1}$, for constant in k erasure resiliency.

We experimentally evaluate the proposed codes in a realistic cloud storage simulator that models node rebuilds in Hadoop. Our simulator was initially validated on a real Hadoop system of 16 machines connected by a 1GB/s network. Our subsequent experiment involves 100 machines and compares the performance of SRC to replication and standard Reed-Solomon codes. We find that SRCs add a new attractive point in the design space of redundancy mechanisms for cloud storage.

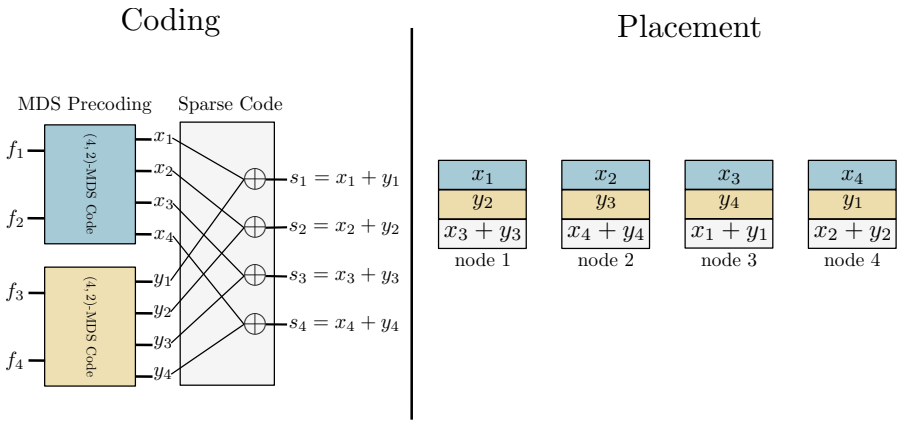

Fig. 1. Example of a $(4, 2, 2)$-SRC. $n = 4$ storage nodes, any $k = 2$ recover the data and XORs of degree $f = 2$ provide simple repair.

II. Simple Regenerating Codes

The first requirement from our storage code is the (n, k) property: a code will be storing information in n storage nodes and should be able *to tolerate any combination of* $n - k$ *failures* without data loss. We refer to codes that have this reliability as "(n, k) *erasure codes*," or codes that have "*the* (n, k) *property*."

One well-known class of erasure codes that have this property is the family of maximum distance separable (MDS) codes [5], [10]. In short, an MDS code is a way to take a data object of size M, split it into chunks of size M/k and create n chunks *of the same size* that have the (n, k) property. It can be seen that MDS codes achieve the (n, k) property with the minimum storage overhead possible: any k storage nodes jointly store M bits of useful information, which is the minimum possible to guarantee recovery.

Our second requirement is efficient exact repair [5]. When one node fails or becomes unavailable, the stored information should be easily reconstructable using other surviving nodes. Simple regenerating codes achieve the (n, k) property and simple repair simultaneously by separating the two problems. Large MDS codes are used to provide reliability against any $n-k$ failures while very simple XORs applied *over the MDS coded packets* provide efficient exact repair when single node failures happen.

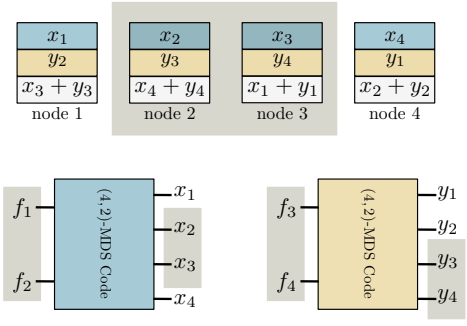

Fig. 2. File reconstruction of a $(4, 2, 2)$-SRC.

We give a first overview of our construction through a simple example in Fig. 1, which shows an $(n = 4, k = 2, f = 2)$-SRC. The original data object is split in 4 chunks f_1, f_2, f_3, f_4. We first encode $[f_1\ f_2]$ in $[x_1\ x_2\ x_3\ x_4]$ and $[f_3\ f_4]$ in $[y_1\ y_2\ y_3\ y_4]$ using any standard $(4, 2)$ MDS code. This can be easily done by multiplication of the data with the 2×4 generator matrix $\mathbf{G}$ of the MDS code to form $[x_1\ x_2\ x_3\ x_4] = [f_1\ f_2]\mathbf{G}$ and $[y_1\ y_2\ y_3\ y_4] = [f_3\ f_4]\mathbf{G}$. Then we generate a parity out of each "level" of coded chunks, i.e., $s_i = x_i + y_i$, which results in an aggregate of 12 chunks. We circularly place these chunks in 4 nodes, each storing 3, as shown in Fig. 1.

It is easy to check that this code has the (n, k) property and in Fig. 2 we show an example by failing nodes 1 and 4. Any two nodes contain two x_i and two y_i chunks which through the outer MDS codes can be used to recover the original data object. We note that the parity chunks are not used in this process, which shows the sub-optimality of our construction.

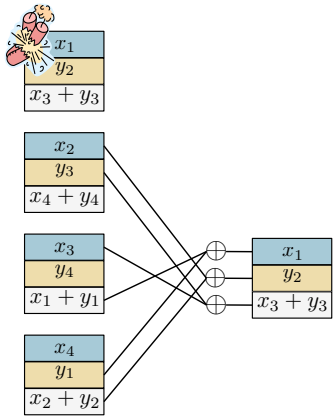

Fig. 3. The repair of node 1 in a $(4, 2, 2)$-SRC

In Fig. 3, we give an example of a single node repair of the $(4, 2, 2)$-SRC. We assume that node 1 is lost and a newcomer joins the system. To reconstruct x_1, the newcomer has to download y_1 and s_1 from nodes 3 and 4. This simple repair scheme is possible due to the way that we placed the chunks in the 4 storage nodes: each node stores 3 chunks with

different index. The newcomer reconstructs each lost chunk by downloading, accessing, and XORing 2 other chunks. In this process the outer MDS codes are not used.

In short our codes combine outer MDS codes and simple parities to provide fault tolerance and efficient repair respectively. Due to this separation of duties, our codes are suboptimal. However, as we show subsequently this optimality loss corresponds to asymptotically negligible loss in storage efficiency and only a logarithmic factor overhead compared to the optimal information theoretic storage bounds.

A. The $f = 2$ Case: degree 2 parities

We now present our general SRC construction for the $f = 2$ case.

B. Code Construction, Erasure Resiliency, and Rate

Let a file $\mathbf{f}$, of size $M = 2k$, that we cut into 2 parts, say

$$\mathbf{f} = \left[\mathbf{f}^{(1)} \;\; \mathbf{f}^{(2)}\right], \tag{1}$$

where $\mathbf{f}^{(i)} \in \mathbb{F}^{1\times k}$, for $i \in [2]$, where $[N] = \{1, \ldots, N\}$ and $\mathbb{F}$ is the finite field over which all operations are performed. Our coding process, is a two-step one: first we independently encode each of the file parts using an outer MDS code and generate simple parity sum out of them. Then we store the coded chunks and the parity sum chunks in a specific way in n storage components. This encode and place scheme enables easy repair of lost chunks and arbitrary erasure tolerance.

We start with an (n, k) MDS code that we use to encode *independently* each of the 2 file parts of size k, $\mathbf{f}^{(1)}$ and $\mathbf{f}^{(2)}$, into two coded vectors, $\mathbf{x}$ and $\mathbf{y}$, of length n. This encoding process is given by

$$\mathbf{x} = \mathbf{f}^{(1)}\mathbf{G} \text{ and } \mathbf{y} = \mathbf{f}^{(2)}\mathbf{G}, \tag{2}$$

where $\mathbf{G} \in \mathbb{F}^{k\times n}$ is the outer MDS code generator matrix. We pose no requirements on that MDS code, in the sense that any (n, k) MDS design will work for our purposes. The maximum distance of the code ensures that any k encoded chunks of $\mathbf{x}$ can reconstruct $\mathbf{f}^{(1)}$; the same goes for any k chunks from $\mathbf{y}$, i.e., we can use them to reconstruct $\mathbf{f}^{(1)}$. We continue by generating a parity sum vector by adding the two coded vectors $\mathbf{x}$ and $\mathbf{y}$

$$\mathbf{s} = \mathbf{x} + \mathbf{y}, \tag{3}$$

where $s_l = x_l + y_l$; we note that the index l of the parity sum s_l is the same as the subscript of the 2 coded chunks that generate it. This process yields $3n$ chunks: $2n$ coded chunks in the vectors $\mathbf{x}$ and $\mathbf{y}$, and n parity sum chunks, i.e., the vector $\mathbf{s} = \mathbf{x} + \mathbf{y}$.

We proceed by placing these $3n$ chunks in n storage nodes in the following way: each storage node will be storing 3 chunks, one from $\mathbf{x}$, one from $\mathbf{y}$, and one from the parity vector $\mathbf{s}$. We require that these 3 chunks do not share a subscript. This subscript requirement can be guaranteed by the following circular placement of chunks in the i-th node

$$\begin{bmatrix} x_i \\ y_{i\oplus 1} \\ s_{i\oplus 2} \end{bmatrix}, \tag{4}$$

where $i \in [n]$ and $\oplus$ denotes modulus addition on the ring $\{1, \ldots, n\}$ (for example $n \oplus 1 = 1$). The above circular chunk placement results in the following coded array of n storage nodes

node 1	node 2	...	node $n-2$	node $n-1$	node n
x_1	x_2	...	x_{n-2}	x_{n-1}	x_n
y_2	y_3	...	y_{n-1}	y_n	y_1
s_3	s_4	...	s_n	s_1	s_2

We can observe that for $n \geq 2$, indeed the 3 chunks of each node do not share a subscript.

C. Erasure Resiliency and Effective Coding Rate

In this section, we present the erasure resiliency and coding rate of the $(n, k, 2)$-SRC and prove the following theorem. Due to lack of space we do not present some proofs in full length and we give sketches instead. The extended version of the paper with full proofs can be found online at [1].

Theorem 1: The $(n, k, 2)$-SRC can tolerate any possible combination $n-k$ erasures and has effective coding rate $\frac{2}{3} \cdot \frac{k}{n}$.

Proof Sketch: The (n, k) property of the SRC is inherited by the underlying MDS outer codes: we can always retrieve the file by connecting to any subset of k nodes of the storage array. Any subset of k nodes contain k chunks of each of the two file parts $\mathbf{f}^{(1)}$ and $\mathbf{f}^{(2)}$, which can be retrieved by inverting the corresponding $k \times k$ submatrices of the MDS generator matrix $\mathbf{G}$ Hence, the (n, k) property of the two identical outer MDS pre-codes renders gives the $(n, k, 2)$-SRC its (n, k) property.

We proceed by calculating the coding rate (space efficiency) R of the $(n, k, 2)$-SRC, by considering the ratio of the total amount of useful stored information, to the total amount of data that is stored. That is, the ratio of the initial file size to the expedited storage

$$R = \frac{\text{file size}}{\text{storage spent}} = \frac{2 \cdot k}{3 \cdot n}. \tag{5}$$

□

Hence, the $(n, k, 2)$-SRC is an erasure code with rate upper bounded by $\frac{2}{3}$: for fixed erasure tollerance, $n - k = m$, the SRC can have rate arbitrarily close to $\frac{2}{3}$, that is,

$$\frac{2}{3}\frac{k}{k+m} \overset{k\to\infty}{\longrightarrow} \frac{2}{3}. \tag{6}$$

The $(n, k, 2)$ SRC construction that is presented in this section can be generalize to constructions where the rate can be made arbitrarily high. This is done by increasing the amount of chunks stored per node and the degree of the parity sums from 2 to f. These constructions are presented in Section III.

D. Repairing Lost Chunks

For the general $(n, k, 2)$-SRC, when a single node is lost, or a single chunk of that lost node is requested to be accessed, the repair process is initiated. To sustain high data availability in the presence of chunk and node erasures, the repair process has to be fast and simple: it should be low cost with respect to information read, communicated, and with respect to the number of total disk accesses. The circular placement of chunks in the SRC enables easy repair of single lost chunks, or single node failures, with respect to the aforementioned metrics. This is due to the fact that each chunk that is lost shares an index with 2 more chunks stored in 2 distinct nodes. By contacting these 2 remaining nodes, we can repair the lost chunk by a simple XOR operation. For the repair of a single chunk or a single node, we have the following theorem.

Theorem 2: The repair of a single chunk of the $(n, k, 2)$-SRC costs 2 in repair bandwidth and chunk reads, that is a fraction $\frac{1}{k}$ of the file size, and 2 disk accesses. Moreover, the repair of a single node failure costs 6 in repair bandwidth and chunk reads, that is a fraction $\frac{3}{k}$ of the file size, and 4 in disk accesses.

Proof: Let for example node $i \in [n]$ fail, that is, chunks x_i, $y_{i\oplus 1}$, and $s_{i\oplus 2}$ are lost. Then, a newcomer joins the storage array and wishes to regenerate the lost information. To reconstruct $x_i^{(1)}$, the newcomer connects to the two chunks available in the storage system that share the same subscript i, i.e., it connects to the node that contains the parity s_i and to the node that contains the chunk y_i. The newcomer can then restore the lost chunk x_i simply by subtracting y_i from the parity s_i. This repair process is summarized in the following 3 steps.

Step	Repair chunk $x_i^{(1)}$:
1	Access Disk $i \ominus 1$ and download y_i
2	Access Disk $i \ominus 2$ and download s_i
3	restore $x_i^{(1)} := s_i - x_i$

where $\ominus$ is subtraction on the ring $\{1, \ldots, n\}$ (for example $1 \ominus 1 = n$). We follow the same manner to repair $y_{i\oplus 1}$:

Step	Repair chunk $y_{i\oplus 1}$:
1	Access Disk $i \oplus 1$ and download $x_{i\oplus 1}$
2	Access Disk $i \ominus 1$ and download $s_{i\oplus 1}$
3	restore $y_{i\oplus 1} := s_{i\oplus 1} - x_{i\oplus 1}$

The parity repair is also similar, we need to access the 2 nodes that contain the coded chunks $x_{i\oplus 2}$, and $y_{i\oplus 2}$ and sum them:

Step	Repair chunk $s_{i\oplus 2}$:
1	Access Disk $i \oplus 2$ and download $x_{i\oplus 2}$
2	Access Disk $i \oplus 1$ and download $y_{i\oplus 2}$
3	restore $s_{i\oplus 2} := x_{i\oplus 2} + y_{i\oplus 2}$

From the above, we observe that the repair of a single chunk contained in a storage node requires 2 disk accesses, 2 chunk reads, and 2 downloads. Moreover, to repair a single node failure an aggregate of 6 chunk reads and 6 downloads is required. The set of disks that are accessed to repair all chunks of nodes i is $\{i \ominus 2, i \ominus 1, i \oplus 1, i \oplus 2\}$, for $i \in [n]$, Hence, the number of disk accesses is $\min(n-1, 4)$, and $n-1$ is true when $i \ominus 2 = i \oplus 2$, as is the case in our $(4, 2, 2)$ example in Figures 1-3. □

Remark 1: We would like to note that a repair would only fail, i.e., one of the packets that are used to regenerate lost information can not be retrieved only if $n \leq 2$.

In the following section, we introduce the general code construction of the (n, k, f)-SRC, where we consider its rate, reliability, repair properties, and analyze its asymptotics.

III. SRC: The General Construction

In this section we generalize the $f = 2$ construction, to the (n, k, f)-SRC. For the general (n, k, f)-SRC, we use f parallel and identical MDS outer pre-codes and generate a single parity vector from f encoded parts. We circularly place the generated chunks in n storage nodes. The (n, k, f)-SRC is an (n, k) erasure code with rate $R = \frac{f}{f+1}\frac{k}{n}$, i.e., the SRC always attains a $\frac{f}{f+1}$ fraction of the space efficiency of an (n, k) MDS code, for the same reliability, but with simple and low cost node repair. We perform single node repairs in the same manner as the $f = 2$ case: to repair a chunk, we access f nodes and perform a simple addition. For any f, the communication overhead to repair a single chunk is a fraction $\frac{1}{k}$ of tha file size and the number of chunk reads and disk accesses is f, which can be constant and not necessarily a function of k. The repair of a single node failure costs $(f+1)\frac{M}{k}$ in repair bandwidth and we prove that the total number of disk accesses needed for a single node failure is exactly $2 \cdot f$. We proceed by introducing the general code construction and showing its properties.

A. Encoding, Erasure Resilience, and Rate

Let a file $\mathbf{f}$, of size $M = fk$, that is subpacketized in f parts,

$$\mathbf{f} = \left[\mathbf{f}^{(1)} \ldots \mathbf{f}^{(f)}\right], \tag{7}$$

with each $\mathbf{f}^{(i)}$, $i \in [f]$, having size k. We encode each of the f file parts independently, into vectors $\mathbf{x}^{(i)}$ of length n, using an outer (n, k) MDS code. That is, we have

$$\mathbf{x}^{(1)} = \mathbf{f}^{(1)}\mathbf{G}, \ \ldots, \ \mathbf{x}^{(f)} = \mathbf{f}^{(f)}\mathbf{G} \tag{8}$$

where $\mathbf{G}$ is the $n \times k$ MDS generator matrix.

Remark 2: The outer MDS code can be any scalar or array (n, k) MDS code, i.e., we pose no requirements on its design or finite field size.

We generate a single parity sum vector from all the coded vectors

$$\mathbf{s} = \sum_{i=1}^{f} \mathbf{x}^{(i)}. \tag{9}$$

This process yields a total of fn coded chunks in the $\mathbf{x}^{(i)}$ vectors and n parity chunks in $\mathbf{s}$, i.e., we have an aggregate of $(f+1)n$ chunks available to place in n nodes.

We will circularly place these $(f+1)n$ chunks in n storage nodes, with each node storing f coded chunks and 1 parity sum chunk, hence each node expends

$$\alpha_{\text{SRC}} = f + 1 = \frac{f+1}{f}\frac{M}{k} \tag{10}$$

in storage capacity. The placement will again obey the property that enables easy repair: no two chunks within a storage node should share the same subscript. To ensure successful repair we also require that $f \leq n$. Below we state the circular placement of chunks in the i-th node, for $i \in [n]$

$$\begin{bmatrix} x_i^{(1)} \\ x_{i\oplus 1}^{(2)} \\ \vdots \\ x_{i\oplus(f-1)}^{(f)} \\ s_{i\oplus f} \end{bmatrix}, \tag{11}$$

which results in the following array of n storage nodes

node 1	node 2	...	node $n-1$	node n
$x_1^{(1)}$	$x_2^{(1)}$	...	$x_{n-1}^{(1)}$	$x_n^{(1)}$
$x_2^{(2)}$	$x_3^{(2)}$	...	$x_n^{(2)}$	$x_1^{(2)}$
$x_3^{(3)}$	$x_4^{(3)}$	...	$x_1^{(3)}$	$x_2^{(3)}$
$\vdots$	$\vdots$	...	$\vdots$	$\vdots$
$x_f^{(f)}$	$x_{f\oplus 1}^{(f)}$	...	$x_{f\oplus(n-2)}^{(f)}$	$x_{f\oplus(n-1)}^{(f)}$
$s_{f\oplus 1}$	$s_{f\oplus 2}$	...	$s_{f\oplus(n-1)}$	$s_{f\oplus n}$

Then, we have the following theorem.

Theorem 3: The (n, k, f)-SRC can tolerate any combination of $n-k$ node erasures and has coding rate $\frac{f}{f+1} \cdot \frac{k}{n}$.

Proof Sketch: The f MDS pre-codes guarantee perfect file reconstruction posterior to any $n-k$ erasures. The file can always be reconstructed by connecting to any k nodes: any collection of k nodes contain fk distinct coded chunks, k of each file part. Each of these k-tuples of coded chunks can give back the information chunks of a single file part due to the f outer MDS codes.

The effective coding rate of the (n, k, f)-SRC is equal to the ratio of the initial file size to the expedited storage, that is

$$R_{\text{SRC}} = \frac{\text{file size}}{\text{storage spent}} = \frac{f \cdot k}{(f+1) \cdot n}. \tag{12}$$

□

By the above theorem we can claim that the rate of the SRC is a fraction $\frac{f}{f+1}$ of the coding rate of an (n, k) MDS code, hence is upper bounded by

$$\frac{f}{f+1}\frac{k}{k+m} \overset{k\to\infty}{\longrightarrow} \frac{f}{f+1}. \tag{13}$$

□

In Fig. 4, we show how the effective coding rate of a $(20, 16, f)$ SRC scales as a function of f, and compare it with that of a $(20, 16)$ MDS code. Both codes can tolerate 4 failures. We observe that as f increases the coding rate of the SRC approaches that of the MDS code.

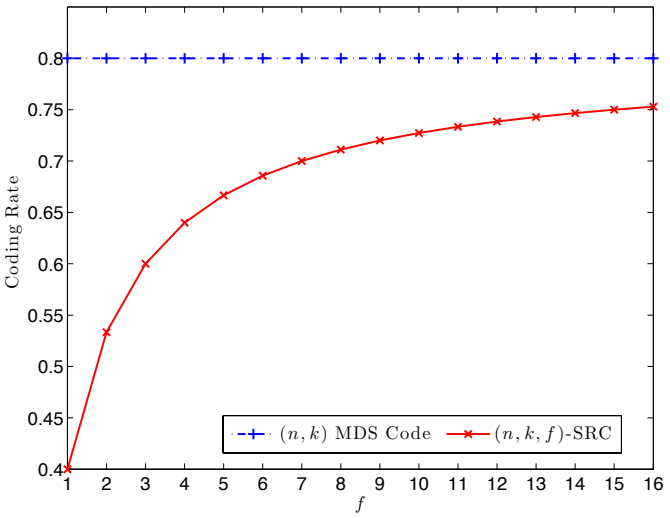

Fig. 4. Rate comparison of a $(20, 16, f)$-SRC and a $(20, 16)$-MDS Code.

B. Repairing Lost Elements

In this subsection we prove the repair properties of the SRC, which are summarized in the following theorem.

Theorem 4: The repair of a single chunk of the (n, k, f)-SRC, where each node stores $\alpha_{\text{SRC}} = \frac{f+1}{f} \cdot \frac{M}{k}$, costs $\frac{M}{k}$ in repair bandwidth and f in chunk reads, and disk accesses. The repair of a single node failure costs

$$\gamma_{\text{SRC}} = (f+1)\frac{M}{k} \tag{14}$$

in repair bandwidth, $f(f+1) = (f+1)\frac{M}{k}$ in chunk reads, and

$$d_{\text{SRC}} = \min(2f, n-1) \tag{15}$$

in disk accesses.

Proof: Let node $i \in [n]$ fail. A newcomer node can reconstruct the lost chunk $x_{i\oplus(l-1)}^{(l)}$ by accessing all f nodes in the set

$$\mathcal{S}_i(l) = \{i \ominus (f-1+l), i \ominus (f-2+l), \ldots, i \ominus l\} \backslash i. \tag{16}$$

and downloading the chunk of each node that has the same subscript $i \oplus (l-1)$ as the lost chunk. For example to reconstruct $x_i^{(1)}$ we need to perform the following steps:

Step	Repair chunk $x_i^{(1)}$:
1	Access Disk $i \ominus 1$ and download $x_i^{(2)}$
2	Access Disk $i \ominus 2$ and download $x_i^{(3)}$
$\vdots$	$\vdots$
f-1	Access Disk $i \ominus (f-1)$ and download $x_i^{(f)}$
f	Access Disk $i \ominus f$ and download s_i
f+1	restore $x_i := s_i - \sum_{l=2}^{f} x_i^{(l)}$

Hence, repairing a single coded chunk requires $f = \frac{M}{k}$ chunk downloads, reads, f and disk accesses. To reconstruct the parity sum chunk $s_{i\oplus f}$, we need to connect to the f nodes that contain the chunks $x_{i\oplus f}^{(l)}$, $l \in [f]$ which generate it.

	(n,k)-MDS	$(n,k,d=n-1)$-MSR	$(n,k,d=k)$-MBR	$(n,k,d=n-1)$-MBR	(n,k,f)-SRC
Storage per node (α)	M/k	M/k	$2\frac{k}{k+1}M/k$	$\frac{2(n-1)}{2(n-1)-k+1}M/k$	$\frac{f+1}{f}M/k$
Repair Bandwidth (γ)	M	$\frac{n-1}{n-k}M/k$	$2\frac{k}{k+1}M/k$	$\frac{2(n-1)}{2(n-1)-k+1}M/k$	$(f+1)M/k$
Disk Accesses (d)	k	$n-1$	k	$n-1$	$2\cdot f$
Rate (R)	k/n	k/n	$\frac{1}{2}\cdot\frac{k+1}{n}\leq\frac{1}{2}$	$\leq\frac{1}{2}$	$\frac{f}{f+1}k/n$

Fig. 5. (n,k,f)-SRC Performance Comparison

To repair a single node failure we need to communicate and read $(f+1)f=(f+1)\frac{M}{k}$ symbols. The total number of disk accesses for a single node repair is given by the number of distinct indices in the set

$$\mathcal{S}_i=\bigcup_{l=1}^{f+1}\mathcal{S}_i(l). \tag{17}$$

To enumerate the distinct indices in $\mathcal{S}_i$, we first count the number of distinct indices between sets $\mathcal{S}_i(l)$ and $\mathcal{S}_i(l+1)$ for all $l\in[f]$. We observe that

$$\mathcal{S}_i(l)\cup\mathcal{S}_i(l+1)=\{i\ominus(f-1+l),i\ominus(f-2+l),\ldots,i\ominus(l-1)\}\backslash i$$

and

$$|\mathcal{S}_i(l)\cup\mathcal{S}_i(l+1)|=f+1, \tag{18}$$

that is, for any two "consecutive" chunk repairs, we need to access $f+1$ storage nodes. Starting with f disk accesses for the first chunk repair, each additional chunk repair requires an additional disk access, with respect to what has already been accessed. The total number of disks accessed is

$$\begin{aligned}d_{\text{SRC}}&=(\#\text{ of disks accesses for chunk } x_i^{(1)})\\&\quad+(\#\text{ of disks accesses for chunk } x_{i\oplus1}^{(2)})\\&\quad\vdots\\&\quad+(\#\text{ of disks accesses for chunk } s_{i\oplus f})\\&=f+\underbrace{1+1+\ldots+1}_{f\text{additional disks accesses}}=2\cdot f\end{aligned} \tag{19}$$

Therefore, to repair a single node failure an aggregate of $2f$ disk accesses is required, when $2f\leq n-1$. If $2f>n-1$ then the number of total disk accesses is $n-1$. □

In Fig. 5, we give a comparison table between MDS, MSR, MBR, and Simple Regenerating Codes, with respect to 1) storage capacity per node α, 2) repair bandwidth per single node repair γ, 3) number of disk accesses per single node repair d, and 4) effective coding rate R. We consider MSR and MBR codes that connect to $d=\{k,n-1\}$ remaining nodes for a single node failure. Observe that the number of disk acceses in the SRC is a design parameter that can be set to a constant by appropriately choosing f, which can be orders less than k.

Remark 3: Regenerating Codes [4] have the property that a single node failure can be repaired by any subset of d remaining nodes, and $k\leq d\leq n-1$ is fixed by the specific code design. In sharp contrast, SRCs are look-up repair codes: for a single node failure, only a specific d_{SRC} subset of the remaining nodes can reconstruct the file and d_{SRC} can be a constant, or a function of k that potentially grows much slower than $\Theta(k)$.

C. Asymptotics of the SRC and links to MDS codes

In this subsection, we consider the asymptotics of the SRC. What happens if we fix $R=\frac{k}{n}$ and let the degree of parities f grow as a function of k? Let for example

$$f=\log(k). \tag{20}$$

Then, the repair of a single node costs $\gamma_{\text{SRC}}=(\log(k)+1)M/k$, with $d_{\text{SRC}}=2f=2\log(k)$. In comparison, a single node failure of an (n,k) MSR code costs $\gamma_{\text{MSR}}=\frac{n-1}{n-k}M/k$. If we let k and n grow and fix $R=\frac{k}{n}$ we obtain

$$\frac{\gamma_{\text{SRC}}}{\gamma_{\text{MSR}}}=\frac{\log(k)+1}{\frac{n-1}{n-k}}=\frac{\log(k)+1}{\frac{1/Rk-1}{(1/R-1)k}}=\Theta(\log(k)). \tag{21}$$

The effective coding rate of the SRC is given by

$$\frac{f}{f+1}\frac{k}{n}=\frac{\log(k)}{\log(k)+1}\frac{k}{n}\overset{k\to\infty}{\longrightarrow}R. \tag{22}$$

Therfore, compared to repair optimal MDS codes, i.e. MSR codes, SRCs with $f=\log(k)$ sacrifice asymptotically negligible coding rate and have a logarithmic overhead compared to minimum bandwidth node repair, when at the same time they attain very easy repair based on simple XORs, with logarithmic in k number of disk accesses.

IV. Simulations

In addition to our theoretical analysis, we evaluate SRCs in a realistic cloud storage simulator. We only tested SRCs with $f=2$ in this paper. This case allows the most efficient repair but at somewhat high storage overhead. We leave the exploration of other choices of f and the involved tradeoffs as future work.

A. Simulator Introduction

We first present the architecture of the cloud storage system that our simulator is modeling. The architecture contains one master server and a great number of data storage servers, similar to that of GFS [23] and Hadoop [24]. As a cloud storage system may store up to tens of petabytes of data, we expect numerous failures and hence fault tolerance and high availability are critical. To offer high data reliability, the master server needs to monitor the health status of each storage server and detect failures promptly.

In the systems of interest, data is partitioned and stored as a number of fixed-size chunks, which in Hadoop can be 64MB or 128MB. Chunks form the smallest accessible data units and in our system are set to be 64MB. To tolerate storage server failures, replication or erasure codes are employed to generate redundant chunks. Then, several chunks are grouped and form a redundancy set [25]. If one chunk is lost, it can be reconstructed from other surviving chunks. To repair the chunks due to a failure event, the master server will initiate the repair process and schedule repair jobs.

We implemented a discrete-event simulator of a cloud storage system using a similar architecture and data repair mechanism as Hadoop. To provide accurate simulation results, our simulator models most entities of the involved components such as machines and chunks. When performing repair jobs, the simulator keeps track of the details of each repair process which gives us a detailed performance analysis.

B. Simulator Validation

We first calibrated our simulator to accurately model the data repair behavior of Hadoop. During the validation, we ran one experiment on a real Hadoop system. This system contains 16 machines, which are connected by a 1Gb/s network. Each machine has about 410GB data, namely approximately 6400 chunks. Then, we manually failed one machine, and let Hadoop repair the lost data. After the repair was completed, we analyzed the log file of Hadoop and derived repair time of each chunk. Next, we ran a similar experiment in our simulator. We also collected the repair time of each chunk from the simulation. We present the CDF of the repair time of both experiments in Fig. 6.

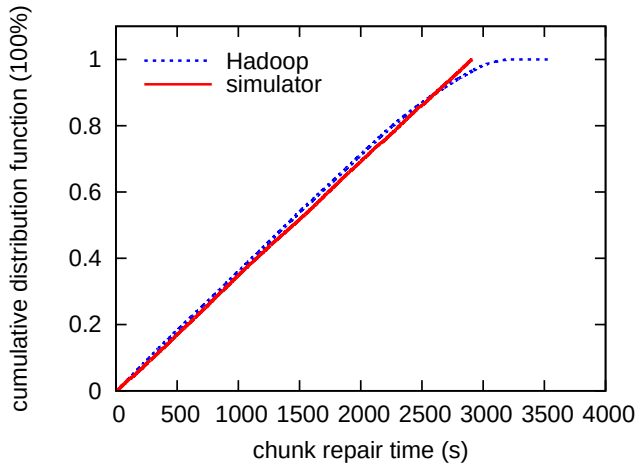

Fig. 6. CDF of repair time

Fig. 6 shows that the repair result of the simulation matches the results of the real Hadoop system very well, particularly when the percentile is below 95. Therefore, we conclude that the simulator can precisely simulate the data repair process of Hadoop.

C. Storage Cost Analysis

Now we observe how storage overhead varies when we grow (n, k). We compare three codes: 3-way replication, Reed-Solomon (RS) codes, and SRC. To make the storage overhead easily understood, we define the cost of storing one byte as the metric of how many bytes are stored for each useful byte. Obviously, high cost results in high storage overhead. As 3-way replication is a popularly used approach, we use it as the base line for comparison. The result is presented in Fig. 7.

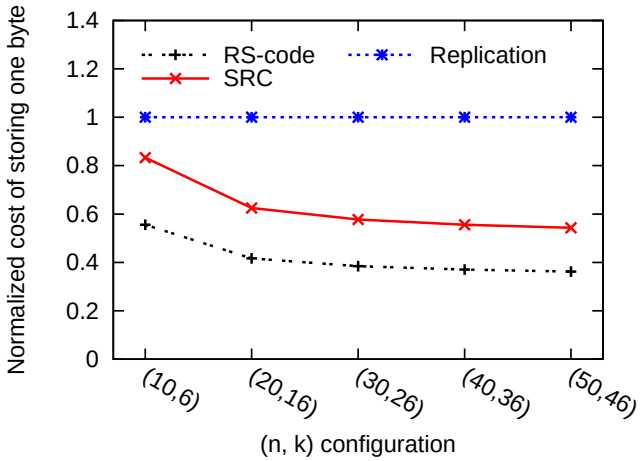

Fig. 7. Storage cost comparison

Fig. 7 shows that when $n - k$ is fixed, the normalized cost of both the RS-code and the SRC decreases as n grows. When (n, k) grows to $(50, 46)$, the normalized cost of SRC is 0.54, and that of RS-code is 0.36. In other words, $(50, 46, 2)$ SRCs need approximately half the storage of 3-way replication. It is worth noting that the cost of SRCs will further reduce if we use larger values of f, but at the cost of slower repair.

D. Repair Performance

In this experiment, we measure the throughput of repairing one failed data server. The experiment involves a total of 100 machines, each storing 410GB of data. We fail at random one machine and start the data repair process. After the repair is finished, we measure the elapsed time and calculate the repair throughput. The results are shown in Fig. 8. Note that the throughput of using 3-way replication is constant across different (n, k) since there is no such dependency on these parameters.

From Fig. 8 we can make two observations. First, 3-way replication has the best repair performance followed by SRC, while the RS-code offers the worst performance. This is not surprising due to the amount of data that has to be accessed for the repair. Second, the repair performance of SRC remains constant on various (n, k), but the performance of RS-code becomes much worse as n grows. This is one of the major benefits of SRC, i.e., the repair performance can be independent from (n, k). Furthermore, the repair throughput of SRC is about 500MB/s, approximately 64% of the 3-way replication's performance.

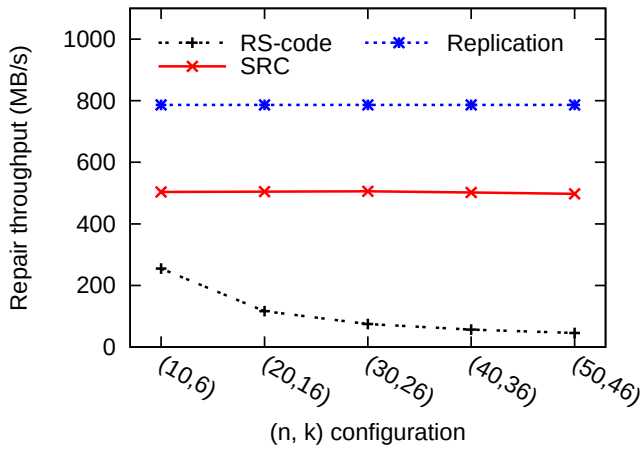

Fig. 8. Repair performance comparison

E. Degraded Read Performance

In a real system, repair can take place in two situations. One situation is when we need to repair a failed data storage server. Another situation is when we wish to read a piece of data, but it is stored in a storage server that is currently unavailable. The two situations differ in whether the repaired data is stored or not. The first situation is a regular repair operation, which writes the repaired data back to the system. The second situation repairs the data in the main memory and then simply drops it after serving the read request. We call the latter degraded read. The degraded read performance is important, since clients can notice performance degradations when servers have temporary or permanent failures.

We use a similar experimental environment to what we presented in section IV-D. The only difference is after a chunk is repaired, we do not write it back. The performance results are presented in Fig. 9.

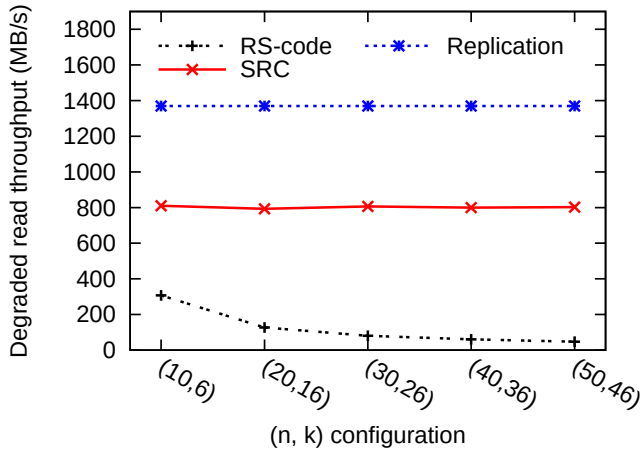

Fig. 9. Degraded read performance comparison

We can also make two observations from Fig. 9. First, for all three codes, the performance trend of degraded read performance is similar to that of repair performance, shown in Fig. 8. Second, for a code with the same (n, k), the degraded read performance is higher than that of repair performance, due to less accessed data. Again, SRC achieves approximately 60% degraded read performance of 3-way replication.

F. Data Reliability Analysis

Now we analyze the data reliability of an SRC cloud storage system. We use a simple Markov model [26] to estimate the reliability. For simplicity, failures happen only to disks and we assume no failure correlations. We note that we expect correlated failures to further benefit SRCs over replication since they spread the data to more nodes and hence achieve better diversity protection under correlated failure scenarios. This, however, remains to be verified in a more thorough experimental study of coded cloud storage systems.

We assume that the mean time to failure (MTTF) of a disk is 5 years and the system stores 1PB data. To be conservative, the repair time is 15 minutes when using 3-way replication and 30 minutes for SRC, which is in accordance to Fig. 8. In the case of RS-code, the repair time depends on k of (n, k). With these parameters, we first measure the reliability of one redundancy set, and then use it to derive the reliability of the entire system. The estimated MTTF of the entire storage system is presented in Fig. 10.

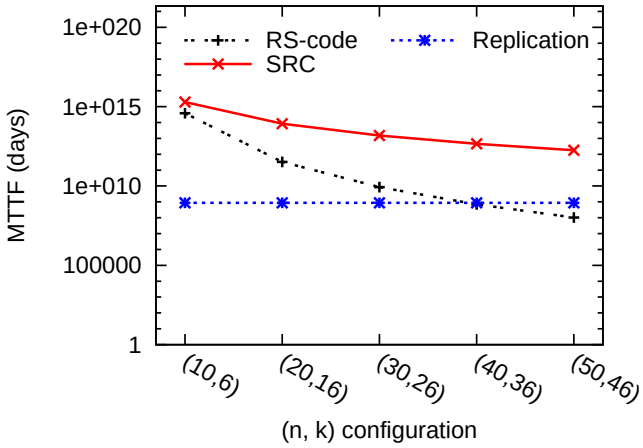

Fig. 10. MTTF comparison

Fig. 10 shows that the data reliability of the 3-way replication is in the order of 10^9. This is consistent with the results in [26]. We can observe that the reliability of SRCs is much higher than 3-way replication. Even for the high rate (low storage overhead) $(50, 46)$ case, SRCs are several orders of magnitude more reliable than 3-way replication. This is benefited from the high repair speed of SRCs. RS codes show a significantly different trend. Although the reliability of $(10, 6)$ and $(20, 16)$ are higher than 3-way replication, the reliability of the RS-code reduces greatly when (n, k) grows. This happens because their repair performance rapidly decreases as k grows.

V. Conclusions

We introduced a novel family of distributed storage codes that are formed by combining MDS codes and simple locally decodable parities for efficient repair and high fault tolerance. We theoretically show that our codes have the (n, k) reliability, have asymptotically optimal storage and are within a logarithmic factor from optimality in repair bandwidth. One very significant benefit is that the number of nodes that need to be contacted for repair can be made a small constant, independent of n, k. Further, SRCs can be easily implemented by combining any prior MDS code implementation with XORing of coded chunks and the appropriate chunk placement into nodes.

We presented a comparison of the proposed codes with replication and Reed-Solomon codes using a cloud storage simulator. We have interest on relatively large values of (n, k) because when we keep $n - k$ constant, larger values of k impose lower storage overhead (higher code rates). Standard Reed-Solomon codes cannot operate in this regime since their repair cost increases linearly in k. On the contrary, SRCs require only a constant number of nodes involved in each repair and can therefore achieve very good storage overhead with good performance. As an example, if we compare a $(50, 46, 2)$ SRC with 3-way replication we find that the SRC requires approximately half the storage but has approximately 60% worse degraded read performance. The main strength of the SRC in this comparison, however, is that it provides approximately four more zeros of data reliability compared to replication. The comparison with Reed-Solomon leads almost certainly to a win of SRCs when slightly more storage is allowed.

In conclusion we think that SRCs add new feasible points in the tradeoff space of distributed storage codes. They deliver comparable performance to 3-way replication and significantly higher data reliability at a lower storage cost. Our preliminary investigation therefore suggests that SRCs should be attractive for real cloud storage systems.

References

[1] Extended version of the paper available online at `http://tinyurl.com/3ke8ese`

[2] The Coding for Distributed Storage wiki `http://tinyurl.com/storagecoding`

[3] J. Li, S. Yang, X. Wang, B. Li. "Tree-structured data regeneration in distributed storage systems with regenerating codes," in *Proc. IEEE Infocom 2010*, April 2010.

[4] A. G. Dimakis, P. G. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," in *IEEE Trans. on Inform. Theory*, vol. 56, pp. 4539 – 4551, Sep. 2010.

[5] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh, "A survey on network codes for distributed storage," in *IEEE Proceedings*, vol. 99, pp. 476 – 489, Mar. 2011.

[6] A. Asterjadhi, E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi. "Towards network coding-based protocols for data broadcasting in wireless ad hoc networks." in *IEEE Transactions on Wireless Commun.*, vol. 9, pp. 662 – 673, Feb. 2010

[7] C. Gkantsidis, J. Miller, and P. Rodriguez, "Comprehensive view of a live network coding P2P system," in IMC, Association for Computing Machinery, Inc., Oct 2006

[8] O. Khan, R. Burns, J. Plank, and C. Huang, "In search of I/O-optimal recovery from disk failures," to appear in *Hot Storage 2011, 3rd Workshop on Hot Topics in Storage and File Systems*, Portland, OR, Jun., 2011.

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

[10] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in raid architectures," in *IEEE Transactions on Computers*, 1995.

[11] Z. Wang, A. G. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies (ACTEMT)*, 2010.

[12] L. Xiang, Y. Xu, J.C.S. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems" in *Proc. ACM SIGMETRICS (2010) international conference on Measurement and modeling of computer systems*

[13] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *Proc. IEEE Infocom 2011*, Shanghai, China, Apr. 2011.

[14] K.V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran "Explicit construction of optimal exact regenerating codes for distributed storage," In *Allerton Conf. on Control, Comp., and Comm.*, Urbana-Champaign, IL, September 2009.

[15] M. Sardari, R. Restrepo, F. Fekri, and E. Soljanin "Memory allocation in distributed storage networks," Proceedings of the IEEE International Symposium on Information Theory (ISIT) 2010

[16] C. Suh and K. Ramchandran, "Exact regeneration codes for distributed storage repair using interference alignment," in *Proc. 2010 IEEE Int. Symp. on Inform. Theory (ISIT)*, Seoul, Korea, Jun. 2010.

[17] K. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction," submitted to IEEE Transactions on Information Theory. Preprint available at at http://arxiv.org/pdf/1005.4178.

[18] S. El Rouayheb and K. Ramchandran, "Fractional repetition codes for repair in distributed storage systems," in Proc. of *48th Allerton Conf. on Commun., Control and Comp.*, Monticello, IL, September 2010.,"

[19] I. Tamo, Z. Wang, and J. Bruck, "MDS array codes with optimal rebuilding," to appear in *2011 IEEE Symposium on Information Theory (ISIT)*. Preprint available at http://arxiv.org/abs/1103.3737

[20] V. R. Cadambe, C. Huang, S. A. Jafar, and J. Li, "Optimal repair of MDS codes in distributed storage via subspace interference alignment," *arxiv pre-print 2011*. Preprint available at http://arxiv.org/abs/1106.1250.

[21] K. W. Shum and Y. Hu, "Exact minimum-repair-bandwidth cooperative regenerating codes for distributed storage systems," to appear in *2011 IEEE Symposium on Information Theory (ISIT)*. Preprint available at http://arxiv.org/abs/1102.1609.

[22] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe, "Repair optimal erasure codes through Hadamard designs," Preprint available at http://arxiv.org/abs/1106.1634v1.

[23] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP '03: Proc. of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[24] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *MSST '10: Proc. of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, 2010.

[25] Q. Xin, E. L. Miller, D. D. E. Long, S. A. Brandt, T. Schwarz, and W. Litwin, "Reliability mechanisms for very large storage systems," in *MSST '03: Proc. of the 20th IEEE Symposium on Massive Storage Systems and Technologies*, 2003.

[26] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. T. L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *OSDI '10: Proc. of the 9th Usenix Symposium on Operating Systems Design and Implementation*, 2010.