NCCloud: A Network-Coding-Based Storage System in a Cloud-of-Clouds

Henry C. H. Chen, Yuchong Hu, Patrick P. C. Lee, and Yang Tang

Abstract—To provide fault tolerance for cloud storage, recent studies propose to stripe data across multiple cloud vendors. However, if a cloud suffers from a permanent failure and loses all its data, we need to repair the lost data with the help of the other surviving clouds to preserve data redundancy. We present a proxy-based storage system for fault-tolerant multiple-cloud storage called NCCloud, which achieves cost-effective repair for a permanent single-cloud failure. NCCloud is built on top of a network-coding-based storage scheme called the functional minimum-storage regenerating (FMSR) codes, which maintain the same fault tolerance and data redundancy as in traditional erasure codes (e.g., RAID-6), but use less repair traffic and hence incur less monetary cost due to data transfer. One key design feature of our FMSR codes is that we relax the encoding requirement of storage nodes during repair, while preserving the benefits of network coding in repair. We implement a proof-of-concept prototype of NCCloud and deploy it atop both local and commercial clouds. We validate that FMSR codes provide significant monetary cost savings in repair over RAID-6 codes, while having comparable response time performance in normal cloud storage operations such as upload/download.

Index Terms—Regenerating codes, network coding, fault tolerance, recovery, implementation, experimentation

1 Introduction

Cloud storage provides an on-demand remote backup solution. However, using a single cloud storage provider raises concerns such as having a single point of failure [7] and vendor lock-ins [1]. As suggested in [1], [7], [10], [11], [61], a plausible solution is to stripe data across different cloud providers. By exploiting the diversity of multiple clouds, we can improve the fault-tolerance of cloud storage.

While striping data with conventional erasure codes performs well when some clouds experience short-term transient failures or foreseeable permanent failures [1], there are real-life cases showing that permanent failures do occur and are not always foreseeable [12], [40], [48], [58]. In view of this, this work focuses on *unexpected permanent* cloud failures. When a cloud fails permanently, it is necessary to activate *repair* to maintain data redundancy and fault tolerance. A repair operation retrieves data from existing surviving clouds over the network and reconstructs the lost data in a new cloud. Today's cloud storage providers charge users for outbound data (see the pricing models in Section 6.1), so moving an enormous amount of data across clouds can introduce significant monetary costs. It is important to reduce the *repair traffic* (i.e., the amount of data being transferred over the network during repair), and hence the monetary cost due to data migration.

To minimize repair traffic, *regenerating codes* [16] have been proposed for storing data redundantly in a distributed storage system (a collection of interconnected storage nodes). Each node could refer to a simple storage device, a storage site, or a cloud storage provider. Regenerating codes are built on the concept of network coding [2], in the sense that nodes perform encoding operations and send encoded data. During repair, each surviving node encodes its stored data chunks and sends the encoded chunks to a new node, which then regenerates the lost data. It is shown that regenerating codes require less repair traffic than traditional erasure codes with the same fault tolerance level [16].

Regenerating codes have been extensively studied in the theoretical context (e.g., [14], [16], [29], [34], [41], [50], [51], [55]–[57]). However, the practical performance of regenerating codes remains uncertain. One key challenge for deploying regenerating codes in practice is that most existing regenerating codes require storage nodes to be equipped with computation capabilities for performing encoding operations during repair. On the other hand, to make regenerating codes *portable* to any cloud storage service, it is desirable to assume only a thin-cloud interface [60], where storage nodes only need to support the standard read/write functionalities. This motivates us to explore, from an applied perspective, how to practically deploy regenerating codes in multiple-cloud storage, if only the thin-cloud interface is assumed.

In this paper, we present the design and implementation of *NCCloud*, a proxy-based storage system designed for providing fault-tolerant storage over multiple cloud storage providers. NCCloud can interconnect different clouds and transparently stripe data across the clouds. On top of NCCloud, we propose the *first* implementable

- *H. Chen, Y. Hu, and P. Lee are with the Chinese University of Hong Kong, Shatin, N.T., Hong Kong (emails: {chchen,pclee}@cse.cuhk.edu.hk, yuchonghu@gmail.com)*
- *Y. Tang is now with Columbia University. His work was done when he was with the Chinese University of Hong Kong (email: ty@cs.columbia.edu)*

- *An earlier 8-page conference version of this paper appeared in USENIX FAST [27]. This article extends the prior work with more in-depth analysis and evaluations on our proposed implementable design of functional minimum storage regenerating (FMSR) codes.*

design for the *functional minimum-storage regenerating (FMSR)* codes[1] [16], [28]. Our FMSR code implementation maintains double-fault tolerance and has the same storage cost as in traditional erasure coding schemes based on RAID-6 codes, but uses less repair traffic when recovering a single-cloud failure. In particular, we eliminate the need to perform encoding operations within storage nodes during repair, while preserving the benefits of network coding in reducing repair traffic. To the best of our knowledge, *this is one of the first studies that puts regenerating codes in a working storage system and evaluates regenerating codes in a practical setting*.

One trade-off of FMSR codes is that they are *non-systematic*, meaning that we store only encoded chunks formed by the linear combination of the original data chunks, and do not keep the original data chunks as in systematic coding schemes. Nevertheless, we mainly design FMSR codes for *long-term archival applications*, in which (i) data backups are rarely read in practice, and (ii) it is common to restore the whole file rather than parts of the file should a lost file needs to be recovered [14][2]. There are many real-life examples in which enterprises and organizations store an enormous amount of archival data (even on the petabyte scale) using cloud storage (e.g., see case studies in [4], [8], [43], [59]). In August 2012, Amazon further introduced Glacier [5], a cloud storage offering optimized for low-cost data archiving and backup (with slow and costly data retrieval) that is being adopted by cloud backup solutions [3], [39]. We believe that FMSR codes provide an alternative option for enterprises and organizations to store data using multiple-cloud storage in a fault-tolerant and cost-effective manner.

While this work is motivated by and established with multiple-cloud storage in mind, we point out that FMSR codes can also find applications in general distributed storage systems where storage nodes are prone to failures and network transmission bandwidth is limited. In this case, minimizing repair traffic is important for reducing the overall repair time.

Our contributions are summarized as follows.

- We present a design of FMSR codes, assuming that double-fault tolerance is used. We show that in multiple-cloud storage, FMSR codes can save the repair cost by 25% compared to RAID-6 codes when four storage nodes are used, and up to 50% as the number of storage nodes further increases. In the meantime, FMSR codes maintain the same amount of storage overhead as RAID-6 codes. Note that FMSR codes can be deployed in a thin-cloud setting as they do not require storage nodes to perform encoding during repair, while still preserving the benefits of network coding in reducing repair traffic. Thus, FMSR codes can be readily deployed in today's cloud storage services.
- We describe the implementation details of how a file object can be stored via FMSR codes. In particular, we propose a two-phase checking scheme, which ensures that double-fault tolerance is maintained in the current and next round of repair. By performing two-phase checking, we ensure that double-fault tolerance is maintained after iterative rounds of repair of node failures. We conduct simulations to validate the importance of two-phase checking.
- We conduct monetary cost analysis to show that FMSR codes effectively reduce the cost of repair when compared to traditional erasure codes, using the price models of today's cloud storage providers.
- We conduct extensive experiments on both local cloud and commercial cloud settings. We show that our FMSR code implementation only adds a small encoding overhead, which can be easily masked by the file transfer time over the Internet. Thus, our work validates the practicality of FMSR codes via NCCloud, and motivates further studies of realizing regenerating codes in large-scale deployments.

The rest of the paper proceeds as follows. In Section 2, we justify the practical importance of repair in multiple-cloud storage. In Section 3, we motivate via examples how FMSR codes reduce repair traffic. In Section 4, we describe our implementable design of FMSR codes, and analyze our proposed two-phase checking scheme for iterative repairs. In Section 5, we present NCCloud, on which FMSR codes are deployed. In Section 6, we evaluate RAID-6 and FMSR codes using NCCloud under both local and commercial cloud settings. Section 7 reviews related work, and Section 8 concludes the paper.

2 Importance of Repair in Multiple-Cloud Storage

In this section, we discuss the importance of repair in cloud storage, especially in disastrous cloud failures that make stored data permanently unrecoverable. We consider two types of failures: transient failure and permanent failure.

Transient failure. A transient failure is expected to be short-term, such that the "failed" cloud will return to normal after some time and no outsourced data is lost. Table 1 shows several real-life examples for the occurrences of transient failures in today's clouds, where the durations of such failures range from several minutes to several days. We highlight that even though Amazon claims that its service is designed for providing 99.99% availability [6], there are arising concerns about this claim and the reliability of other cloud providers after Amazon's outage in April 2011 [12]. We thus expect that transient failures are common, but they will eventually be recovered. If we deploy multiple-cloud storage with enough redundancy, then we can retrieve data from the other surviving clouds during the failure period.

1. The correctness of our FMSR codes is formally proven in our recent work [28].

2. The same argument applies for conventional compression techniques, which reduce storage space at the expense of the decompression overhead when the original data is recovered.

TABLE 1
Examples of transient failures in different cloud services.

Cloud service	Failure reason	Duration	Date
Google Gmail	Software bug [24]	4 days	Feb 27-Mar 2,2011
Google Search	Programming error [38]	40 mins	Jan 31,2009
Amazon S3	Gossip protocol blowup [9]	6-8 hours	July 20,2008
Microsoft Azure	Malfunction in Windows Azure [36]	22 hours	Mar 13-14,2008

Permanent failure. A permanent failure is long-term, in the sense that the outsourced data on a failed cloud will become permanently unavailable. Clearly, a permanent failure is more disastrous than a transient one. Although we expect that a permanent failure is unlikely to happen, there are several situations where permanent cloud failures are still possible:

- *Data center outages in disasters.* AFCOM [48] found that many data centers are ill-prepared for disasters. For example, 50% of the respondents have no plans to repair damages after a disaster. It was reported [48] that the earthquake and tsunami in northeastern Japan in March 11, 2011 knocked out several data centers there.
- *Data loss and corruption.* There are real-life cases where a cloud may accidentally lose data [12], [40], [58]. In the case of Ma.gnolia [40], half a terabyte of data, including its backups, are all lost and unrecoverable.
- *Malicious attacks.* To provide security guarantees for outsourced data, one solution is to have the client application encrypt the data before putting the data on the cloud. On the other hand, if the outsourced data is corrupted (e.g., by virus or malware), then even though the content of the data is encrypted and remains confidential to outsiders, the data itself is no longer useful. AFCOM [48] found that about 65 percent of data centers have no plan or procedure to deal with cyber-criminals.

Unlike transient failures where the cloud is assumed to be able to return to normal, permanent failures will make the hosted data in the failed cloud no longer accessible, so we must *repair* and reconstruct the lost data in a different cloud or storage site in order to maintain the required degree of fault tolerance. In our definition of repair, we mean to retrieve data only from the other surviving clouds, and reconstruct the data in a new cloud or another storage site.

3 Motivation of FMSR Codes

We consider a distributed, multiple-cloud storage setting from a client's perspective, where data is striped over multiple cloud providers. We propose a proxy-based design [1], [30] that interconnects multiple cloud repositories, as shown in Figure 1(a). The proxy serves as an interface between client applications and the clouds. If a cloud experiences a permanent failure, the proxy activates the repair operation, as shown in Figure 1(b).

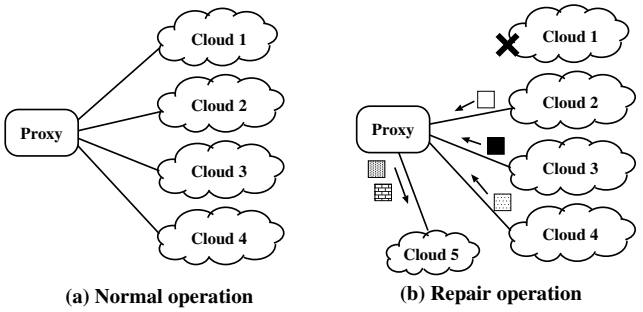


Fig. 1. Proxy-based design for multiple-cloud storage: (a) normal operation, and (b) repair operation when Cloud node 1 fails. During repair, the proxy regenerates data for the new cloud.

That is, the proxy reads the essential data pieces from other surviving clouds, reconstructs new data pieces, and writes these new pieces to a new cloud. Note that this repair operation does not involve direct interactions among the clouds.

We consider fault-tolerant storage based on a type of *maximum distance separable (MDS)* codes. Given a file object of size M, we divide it into equal-size *native chunks*, which are linearly combined to form *code chunks*. When an (n, k)-MDS code is used, the native/code chunks are then distributed over n (larger than k) nodes, each storing chunks of a total size M/k, such that the original file object may be reconstructed from the chunks contained in *any* k of the n nodes. Thus, it tolerates the failures of any $n - k$ nodes. We call this fault tolerance feature the *MDS property*. The extra feature of FMSR codes is that reconstructing the chunks stored in a failed node can be achieved by downloading less data from the surviving nodes than reconstructing the whole file.

This paper considers a multiple-cloud setting with two levels of reliability: fault tolerance and recovery. First, we assume that the multiple-cloud storage is double-fault tolerant (e.g., as in conventional RAID-6 codes) and provides data availability under the transient unavailability of at most two clouds. That is, we set $k = n - 2$. Thus, clients can always access their data as long as no more than two clouds experience transient failures (see examples in Table 1) or any possible connectivity problems. We expect that such a fault tolerance level suffices in practice. Second, we consider single-fault recovery in multiple-cloud storage, given that a permanent cloud failure is less frequent but possible. Our primary objective is to minimize the cost of storage repair (due to

the migration of data over the clouds) for a permanent single-cloud failure. In this work, we focus on comparing two codes: traditional RAID-6 codes and our FMSR codes with double-fault tolerance[3].

We define the *repair traffic* as the amount of outbound data being downloaded from the other surviving clouds during the single-cloud failure recovery. We seek to minimize the repair traffic for cost-effective repair. Here, we do not consider the inbound traffic (i.e., the data being written to a cloud), as it is free of charge for many cloud providers (see Table 3 in Section 6).

We now study the repair traffic involved in different coding schemes via examples. Suppose that we store a file of size M on four clouds, each viewed as a *logical storage node*. Let us first consider conventional RAID-6 codes, which are double-fault tolerant. Here, we consider a RAID-6 code implementation based on the Reed-Solomon code [52], as shown in Figure 2(a). We divide the file into two native chunks (i.e., A and B) of size $M/2$ each. We add two code chunks formed by the linear combinations of the native chunks. Suppose now that Node 1 is down. Then the proxy must download the same number of chunks as the original file from two other nodes (e.g., B and $A+B$ from Nodes 2 and 3, respectively). It then reconstructs and stores the lost chunk A on the new node. The total storage size is $2M$, while the repair traffic is M.

Regenerating codes have been proposed to reduce the repair traffic. One class of regenerating codes is called the *exact minimum-storage regenerating (EMSR)* codes [57]. EMSR codes keep the same storage size as in RAID-6 codes, while having the storage nodes send *encoded* chunks to the proxy so as to reduce the repair traffic. Figure 2(b) illustrates the double-fault tolerant implementation of EMSR codes. We divide a file into four chunks, and allocate the native and code chunks as shown in the figure. Suppose Node 1 is down. To repair it, each surviving node sends the XOR summation of the data chunks to the proxy, which then reconstructs the lost chunks. We can see that in EMSR codes, the storage size is $2M$ (same as RAID-6 codes), while the repair traffic is $0.75M$, which is 25% of saving (compared with RAID-6 codes). EMSR codes leverage the notion of network coding [2], as the nodes generate encoded chunks during repair.

We now consider the double-fault tolerant implementation of FMSR codes as shown in Figure 2(c). We divide the file into four native chunks, and construct eight distinct code chunks $P_1, \cdots, P_8$ formed by different linear combinations of the native chunks. Each code chunk has the same size $M/4$ as a native chunk. Any two nodes can be used to recover the original four native chunks. Suppose Node 1 is down. The proxy collects one code chunk from each surviving node, so it downloads three code chunks of size $M/4$ each. Then the proxy regenerates two code chunks P_1' and P_2' formed by different linear combinations of the three code chunks. Note that P_1' and P_2' are still linear combinations of the native chunks. The proxy then writes P_1' and P_2' to the new node. In FMSR codes, the storage size is $2M$ (as in RAID-6 codes), yet the repair traffic is $0.75M$, which is the same as in EMSR codes. A key property of our FMSR codes is that nodes do not perform encoding during repair.

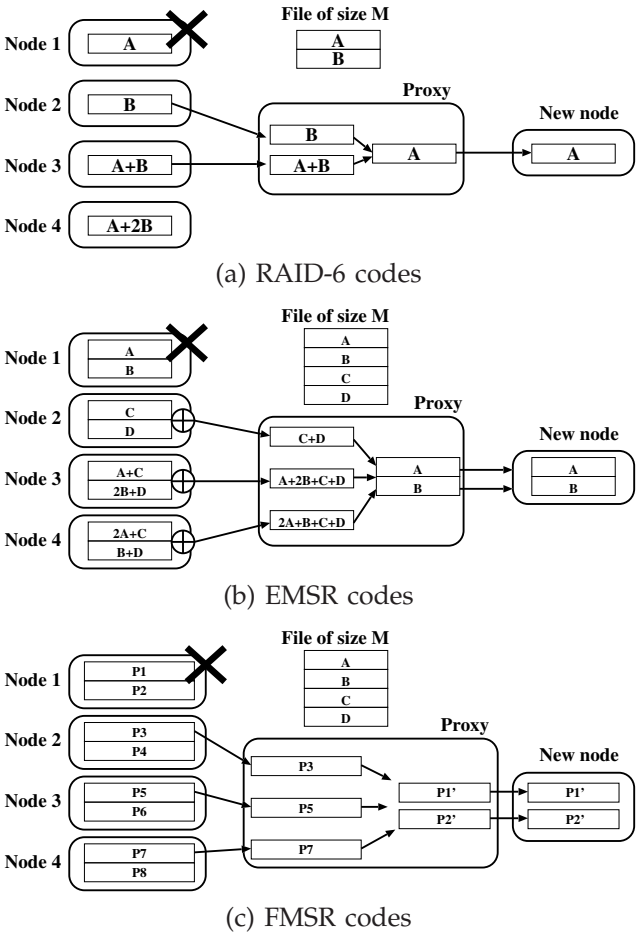


Fig. 2. Examples of repair operations in different codes with $n = 4$ and $k = 2$. All arithmetic operations are performed over the Galois Field GF(2^8).

To generalize double-fault tolerant FMSR codes for n storage nodes, we divide a file of size M into $2(n-2)$ native chunks, and use them to generate $2n$ code chunks. Then each node will store two code chunks of size $\frac{M}{2(n-2)}$ each. Thus, the total storage size is $\frac{Mn}{n-2}$. To repair a failed node, we download one chunk from each of the other $n-1$ nodes, so the repair traffic is $\frac{M(n-1)}{2(n-2)}$. In contrast, for RAID-6 codes, the total storage size is also $\frac{Mn}{n-2}$, while the repair traffic is M. When n is large, FMSR codes can save the repair traffic by *close to 50%*.

Note that FMSR codes are *non-systematic*, as they keep only code chunks but not native chunks. To access a single chunk of a file, we need to download and decode

3. If we assume single-fault tolerance (i.e., $k = n - 1$) and single-fault recovery, then by the theoretical results of [16], we can show that traditional RAID-5 codes [45] have the same data redundancy and same repair traffic as FMSR codes.

the entire file for that particular chunk. This is opposed to systematic codes (as in traditional RAID storage), in which native chunks are kept. Nevertheless, FMSR codes are acceptable for long-term archival applications, where the read frequency is typically low. Also, to restore backups, it is natural to retrieve the entire file rather than a particular chunk [14].

This paper considers the baseline RAID-6 implementation using Reed-Solomon codes. Its repair method involves reconstructing the whole file first, and is applicable for *all* erasure codes in general. Recent studies [35], [62], [63] show that data reads can be minimized specifically for XOR-based erasure codes. For example, in RAID-6, data reads can be reduced by 25% compared to reconstructing the whole file [62], [63]. Although such approaches achieve less saving than FMSR codes, which can save up to 50% of repair traffic, their use of efficient XOR operations can be of practical interest.

4 FMSR Code Implementation

We now present the details for implementing FMSR codes in multiple-cloud storage. We specify three operations for FMSR codes on a particular file object: (1) file upload; (2) file download; (3) repair. Each cloud repository is viewed as a logical storage node. Our implementation assumes a thin-cloud interface [60], such that the storage nodes (i.e., cloud repositories) only need to support basic read/write operations. Thus, we expect that our FMSR code implementation is compatible with today's cloud storage services.

One property of FMSR codes is that we do not require lost chunks to be exactly reconstructed, but instead in each repair, we regenerate code chunks that are not necessarily identical to those originally stored in the failed node, as long as the MDS property holds. We propose a *two-phase checking* scheme, which ensures that the code chunks on all nodes always satisfy the MDS property, and hence data availability, even after iterative repairs. In this section, we analyze the importance of the two-phase checking scheme.

4.1 Basic operations

4.1.1 File Upload

To upload a file F, we first divide it into $k(n-k)$ equal-size native chunks, denoted by $(F_i)_{i=1,2,\cdots,k(n-k)}$. We then encode these $k(n-k)$ native chunks into $n(n-k)$ code chunks, denoted by $(P_i)_{i=1,2,\cdots,n(n-k)}$. Each P_i is formed by a linear combination of the $k(n-k)$ native chunks. Specifically, we let $\mathbf{EM} = [\alpha_{i,j}]$ be an $n(n-k) \times k(n-k)$ encoding matrix for some coefficients $\alpha_{i,j}$ (where $i = 1, \ldots, n(n-k)$ and $j = 1, \ldots, k(n-k)$) in the Galois field GF(2^8). We call a row vector of $\mathbf{EM}$ an *encoding coefficient vector (ECV)*, which contains $k(n-k)$ elements. We let ECV_i denote the i^{th} row vector of $\mathbf{EM}$. We compute each P_i by the product of ECV_i and all the native chunks $F_1, F_2, \cdots, F_{k(n-k)}$, i.e., $P_i = \sum_{j=1}^{k(n-k)} \alpha_{i,j} F_j$ for $i = 1, 2, \cdots, n(n-k)$, where all arithmetic operations are performed over GF(2^8). The code chunks are then evenly stored in the n storage nodes, each having $(n-k)$ chunks. Also, we store the whole $\mathbf{EM}$ in a metadata object that is then replicated to all storage nodes (see Section 5). There are many ways of constructing $\mathbf{EM}$, as long as it passes our two-phase checking (see Section 4.1.3). Note that the implementation details of the arithmetic operations in Galois Fields are extensively discussed in [25].

4.1.2 File Download

To download a file, we first download the corresponding metadata object that contains the ECVs. Then we select any k of the n storage nodes, and download the $k(n-k)$ code chunks from the k nodes. The ECVs of the $k(n-k)$ code chunks can form a $k(n-k) \times k(n-k)$ square matrix. If the MDS property is maintained, then by definition, the inverse of the square matrix must exist. Thus, we multiply the inverse of the square matrix with the code chunks and obtain the original $k(n-k)$ native chunks. The idea is that we treat FMSR codes as standard Reed-Solomon codes, and our technique of creating an inverse matrix to decode the original data has been described in the tutorial [46].

4.1.3 Iterative Repairs

We now consider the repair of FMSR codes for a file F for a permanent single-node failure. Given that FMSR codes regenerates different chunks in each repair, one challenge is to ensure that the MDS property still holds even after *iterative repairs*. This is in contrast to regenerating the *exact* lost chunks as in RAID-6, which guarantees the invariance of the stored chunks. Here, we propose a *two-phase* checking heuristic as follows. Suppose that the $(r-1)^{th}$ repair is successful, and we now consider how to operate the r^{th} repair for a single permanent node failure (where $r \geq 1$). We first check if the new set of chunks in all storage nodes satisfies the MDS property after the r^{th} repair. In addition, we also check if another new set of chunks in all storage nodes still satisfies the MDS property after the $(r+1)^{th}$ repair, should another single permanent node failure occur (we call this the *repair MDS (rMDS) property*). We now describe the r^{th} repair as follows.

Step 1: Download the encoding matrix from a surviving node. Recall that the encoding matrix $\mathbf{EM}$ specifies the ECVs for constructing all code chunks via linear combinations of native chunks. We use these ECVs for our later two-phase checking. Since we embed $\mathbf{EM}$ in a metadata object that is replicated, we can simply download the metadata object from one of the surviving nodes.

Step 2: Select one ECV from each of the $n-1$ surviving nodes. Each ECV in $\mathbf{EM}$ corresponds to a code chunk. We pick one ECV from each of the $n-1$ surviving nodes. We call these ECVs to be ECV_{i_1}, ECV_{i_2}, $\cdots$, $\text{ECV}_{i_{n-1}}$.

Step 3: Generate a repair matrix. We construct an $(n-k) \times (n-1)$ *repair matrix* $\mathbf{RM} = [\gamma_{i,j}]$, where each element $\gamma_{i,j}$ (where $i = 1, \ldots, n-k$ and $j = 1, \ldots, n-1$) is randomly

selected in GF(2^8). Note that the idea of generating a random matrix for reliable storage is consistent with that in [49].

Step 4: Compute the ECVs for the new code chunks and reproduce a new encoding matrix. We multiply $\mathbf{RM}$ with the ECVs selected in Step 2 to construct $n-k$ new ECVs, denoted by $\text{ECV}'_i = \sum_{j=1}^{n-1} \gamma_{i,j}\text{ECV}_{i_j}$ for $i = 1, 2, \cdots, n-k$. Then we reproduce a new encoding matrix, denoted by $\mathbf{EM}'$, which is formed by substituting the ECVs of $\mathbf{EM}$ of the failed node with the corresponding new ECVs.

Step 5: Given $\mathbf{EM}'$, check if both the MDS and rMDS properties are satisfied. Intuitively, we verify the MDS property by enumerating all $\binom{n}{k}$ subsets of k nodes to see if each of their corresponding encoding matrices forms a full rank. For the rMDS property, we check that for *any* possible node failure (one out of n nodes), we can collect one out of $n-k$ chunks from each of the other $n-1$ surviving nodes and reconstruct the chunks in the new node, such that the MDS property is maintained. The number of checks performed for the rMDS property is at most $n(n-k)^{n-1}\binom{n}{k}$. If n is small, then the enumeration complexities for both MDS and rMDS properties are manageable. If either one phase fails, then we return to Step 2 and repeat. We emphasize that Steps 1 to 5 only deal with the ECVs, so their overhead does not depend on the chunk size.

Step 6: Download the actual chunk data and regenerate new chunk data. If the two-phase checking in Step 5 succeeds, then we proceed to download the $n-1$ chunks that correspond to the selected ECVs in Step 2 from the $n-1$ surviving storage nodes to NCCloud. Also, using the new ECVs computed in Step 4, we regenerate new chunks and upload them from NCCloud to a new node.

Remark: We can reduce the complexity of two-phase checking with the proposed FMSR code construction in our recent work [28]. The proposed construction specifies the ECVs to be selected in Step 2 deterministically, and tests their correctness (i.e., satisfying both MDS and rMDS properties) by checking against a set of inequalities in Step 5. This reduces the complexity of each iteration as well as the number of iterations (i.e., number of times that Steps 2-5 are repeated) in generating a valid $\mathbf{EM}'$. Our current implementation of NCCloud includes the proposed construction. We refer readers to [28] for details of the proposed construction.

4.2 Analysis

We now argue that checking the rMDS property in each repair is necessary in order to maintain the MDS property after iterative repairs. We show via a counter-example that if a repair checks only the MDS property but without checking the rMDS property, then the MDS property will be lost in the next repair. We also show by simulations that our two-phase checking can sustain many iterations of repairs in more general cases.

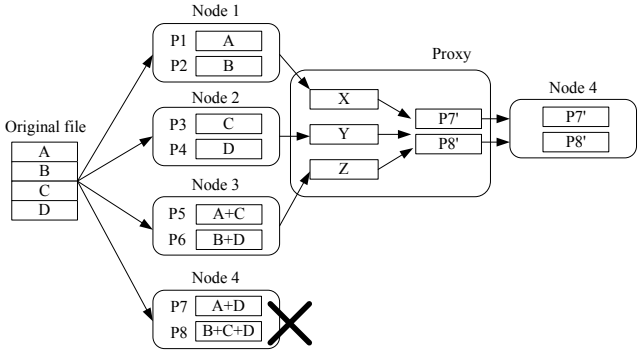


Fig. 3. Counter-example: code chunks that satisfy the MDS property but not the rMDS property.

4.2.1 A Counter-Example

We consider a counter-example shown in Figure 3 to illustrate the importance of the rMDS property. We use the same notation as described in Figure 2(c) with $n = 4$ and $k = 2$. Suppose that the code chunks $P_1, \ldots, P_8$ are linearly combined from native chunks A, B, C, and D as shown in Figure 3, and such linear combinations are in fact the same as in the shortened EVENODD code in [62]. It is easy to verify that the code chunks $P_1, \ldots, P_8$ satisfy the MDS property, i.e., the four chunks from any two nodes can be used to reconstruct the native chunks A, B, C and D. However, we do not check if they satisfy the rMDS property (actually they do not, as we shall see later).

Next, consider that Node 4 fails. Based on FMSR codes, the repair selects one chunk from each of Nodes 1, 2, and 3 (denote them by chunks X, Y, and Z) and use them to regenerate the new code chunks P'_7 and P'_8, which will be stored in a new node (which we still denote by Node 4). There are $2^3 = 8$ possible selections of $\{X, Y, Z\}$. Let us consider one possible selection $\{P_1, P_3, P_5\}$. Then the new code chunks become

$$P'_7 = \gamma_{1,1}P_1 + \gamma_{1,2}P_3 + \gamma_{1,3}P_5,$$
$$P'_8 = \gamma_{2,2}P_1 + \gamma_{2,2}P_3 + \gamma_{2,3}P_5,$$

where $\gamma_{i,j}$ ($i = 1, \ldots, n-k$ and $j = 1, \ldots, n-1$) are some random coefficients used to generate the new code chunks. Then we have

$$P'_7 = (\gamma_{1,1} + \gamma_{1,3})A + (\gamma_{1,2} + \gamma_{1,3})C,$$
$$P'_8 = (\gamma_{2,1} + \gamma_{2,3})A + (\gamma_{2,2} + \gamma_{2,3})C.$$

Since $P_1 = A$ and $P_2 = B$, we cannot reconstruct the native chunk D from P_1, P_2, P'_7, P'_8. The MDS property is lost because the chunks of Nodes 1 and 4 cannot be used to reconstruct the native chunks. Thus, the repair fails with this selection of chunks.

We can apply similar reasoning to other possible selections of chunks. Table 2 lists all eight possible selections of chunks, along with the set of chunks (and nodes) that cannot be used to rebuild the original file. This shows that none of the selections succeeds in maintaining the

TABLE 2
Eight possible selections of chunks from surviving nodes for generating P_7' and P_8', along with the corresponding set of chunks that will fail to reconstruct the file.

X, Y, Z	Set of chunks that cannot rebuild the file
P_1, P_3, P_5	P_1, P_2, P_7', P_8' (Nodes 1 and 4)
P_2, P_3, P_5	P_1, P_2, P_7', P_8' (Nodes 1 and 4)
P_1, P_4, P_5	P_3, P_4, P_7', P_8' (Nodes 2 and 4)
P_2, P_4, P_5	P_5, P_6, P_7', P_8' (Nodes 3 and 4)
P_1, P_3, P_6	P_5, P_6, P_7', P_8' (Nodes 3 and 4)
P_2, P_3, P_6	P_3, P_4, P_7', P_8' (Nodes 2 and 4)
P_1, P_4, P_6	P_1, P_2, P_7', P_8' (Nodes 1 and 4)
P_2, P_4, P_6	P_1, P_2, P_7', P_8' (Nodes 1 and 4)

MDS property after the repair. This counter-example shows how checking the MDS property only but not the rMDS property can lead to a failed repair.

4.2.2 Simulations

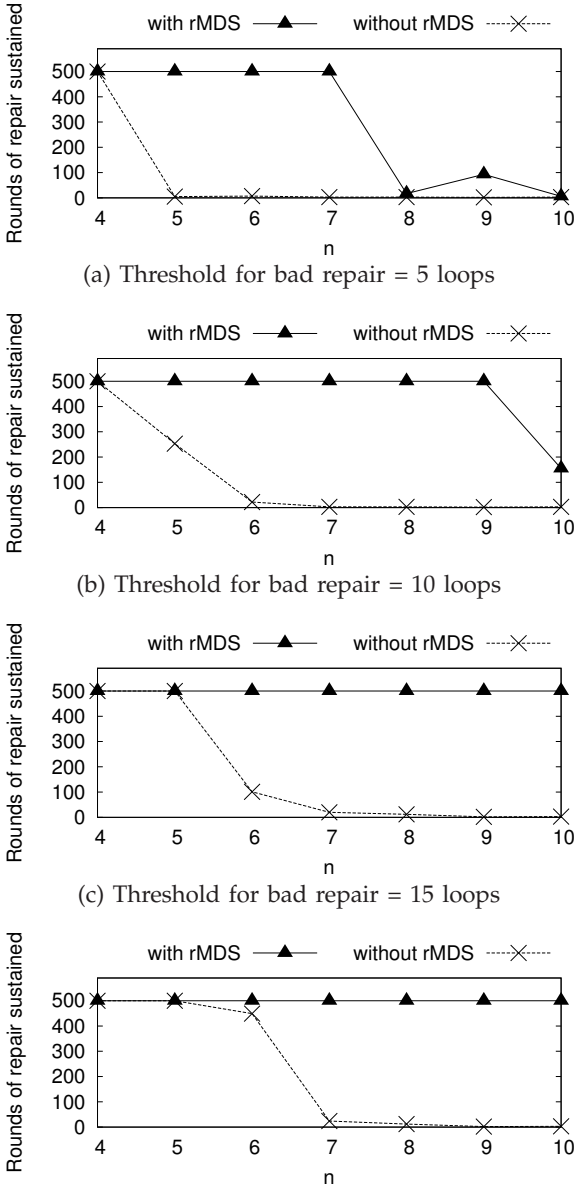
We conduct simulations to justify that checking the rMDS property can make iterative repairs sustainable. We also evaluate via simulations the overhead of our two-phase checking (Steps 2 to 5 of repair). Our simulations are carried out on a 2.4GHz CPU core.

First, we consider multiple rounds of node repairs for different values of n, and argue that in addition to checking the MDS property, checking the rMDS property is essential for iterative repairs. Specifically, in each round, we randomly pick a node to fail, and then repair the failed node. We say a repair is *bad* if the loop of Steps 2 to 5 in our two-phase checking is repeated over a threshold number of times but no suitable encoding matrix has yet been obtained. In our simulations, we vary the threshold of the number of loops for determining a bad repair. We carry out a maximum of 500 rounds of repair, and stop once we encounter a bad repair. We do not include the construction of [28] in this part of simulations to study the effects of the baseline MDS and rMDS checks.

Figure 4 shows the number of rounds of repair that can be sustained when the rMDS property is checked or is not checked. It shows that checking the rMDS property enables us to sustain more rounds of repair before seeing a bad repair. For example, suppose that we set the threshold to be 20 loops. Then we can sustain 500 rounds of repair for different values of n (number of nodes) by checking the rMDS property, but we encounter a bad repair quickly (e.g., in 3 rounds of repair for $n = 10$) if we do not check the rMDS property.

Next we evaluate via simulations the time overhead of the two-phase checking, with the proposed FMSR code construction [28] that reduces the complexity. In each round of repair, we randomly pick a node to fail and carry out the repair operation. We carry out the two-phase checking (i.e., Steps 2 to 5), and measure the time required to generate an encoding matrix that satisfies both the MDS and rMDS properties.

Figure 5 plots the cumulative time of two-phase checking for 50 rounds of repair (in log scale) for $n = 4$ to



(a) Threshold for bad repair = 5 loops

(b) Threshold for bad repair = 10 loops

(c) Threshold for bad repair = 15 loops

(d) Threshold for bad repair = 20 loops

Fig. 4. Number of rounds of repair sustainable without seeing a bad repair.

$n = 16$. The checking spends negligible time compared to the actual repairs of even a 1MB file (see Section 6.2.2). For example, when $n = 10$, it takes only 0.02s to carry out 50 consecutive repairs (around 0.0004s per repair); even when $n = 16$, it takes only 0.1s to carry out 50 consecutive repairs (around 0.002s per repair). Note that the range of n we consider follows the stripe sizes used in many practical storage systems [47]. To further reduce the overhead, we can pre-compute the new encoding coefficients for any possible node failure *offline* while the system is running normally, and keep the results to prepare for the next repair.

4.2.3 Reliability Analysis

Following prior studies that evaluate the reliability of various erasure codes and replication (e.g., [20], [31],

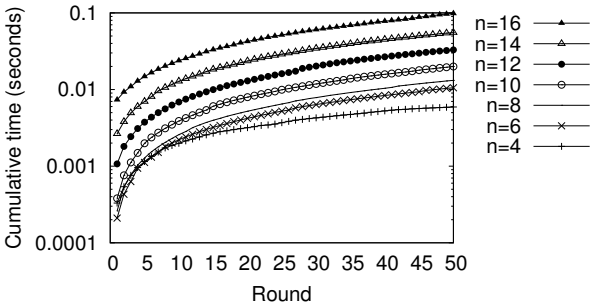


Fig. 5. The cumulative time required by the checking phase (plotted in log scale) in 50 consecutive rounds of repair from $n = 4$ to $n = 16$.

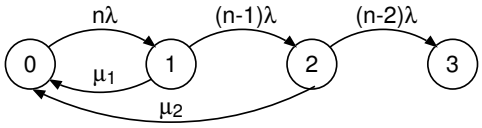
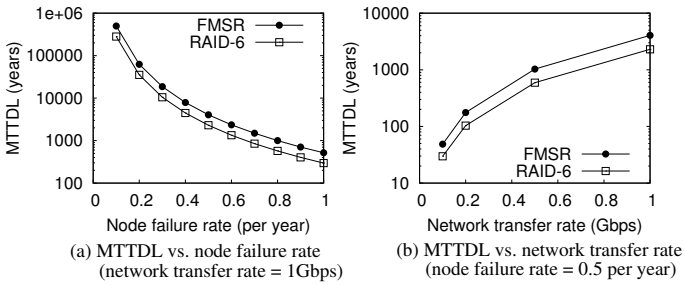


Fig. 6. Markov model for double-fault tolerant codes.

[53]), we compare the reliability of FMSR codes and traditional RAID-6 codes with respect to different failure rates using the mean-time-to-data-loss (MTTDL) metric, defined as the expected time elapsed until the original data becomes unrecoverable. While MTTDL is ineffective to quantify the real reliability [26], it remains a widely adopted reliability metric in the storage community and we use it only for the comparative study of different coding schemes with different repair performance.

MTTDL is solved via the Markov model. Figure 6 shows the Markov model for double-fault tolerant codes (i.e., $k = n - 2$), in which state i (where $i = 0, 1, 2, 3$) denotes the number of failed nodes in a storage system. State 3 means that there are more than two failed nodes and the data is permanently lost. We compute MTTDL as the expected time to move from state 0 (i.e., all nodes are normal) to state 3.

We make assumptions in our analysis. For simplicity, we assume that node failures and repairs are independent events that follow an exponential distribution. The assumption is imperfect in general [54], but it makes our analysis tractable and has been used in prior studies [20], [31], [53]. Let λ be the node failure rate (i.e., $1/\lambda$ is the expected time to failure of a node). Thus, the transition rate from state i to state $i+1$ is $(n-i)\lambda$, where $i = 0, 1, 2$. Also, let μ_1 and μ_2 be the repair rates for single-node and double-node failures, respectively. We assume that the network transfer between the *surviving* nodes and the proxy is the major bottleneck (see Section 3 for our formulation) and determines the resulting repair rates. Let S be the size of the data stored in each node (i.e., the total amount of original data stored is $(n-2)S$) and B be the network capacity between the surviving nodes and the proxy. Now, consider the repair of a single-node failure. As shown in Section 3, for FMSR codes, the repair traffic is $\frac{(n-1)S}{2}$ and hence $\mu_1 = \frac{2B}{(n-1)S}$; for traditional RAID-6 codes, the repair traffic is $(n-2)S$ and hence $\mu_1 = \frac{B}{(n-2)S}$. For the repair of a double-node failure, both



(a) MTTDL vs. node failure rate (network transfer rate = 1Gbps)

(b) MTTDL vs. network transfer rate (node failure rate = 0.5 per year)

Fig. 7. MTTDLs of FMSR codes and RAID-6 codes (plotted in log scale) when $n = 10$ and $k = 8$.

FMSR codes and traditional RAID-6 codes resort to the conventional approach and reconstruct the lost data by downloading the amount of original data (i.e., $(n-2)S$) from the remaining $k = n-2$ surviving nodes. Both have $\mu_2 = \frac{B}{(n-2)S}$.

We now evaluate the MTTDLs of FMSR codes and traditional RAID-6 codes for some specific parameters. Suppose that we fix $n = 10$, $k = 8$, and $S = 1$TB. Figure 7(a) shows the MTTDLs for different values of λ from 0.1 to 1 (in units per year) when $B = 1$Gbps, while Figure 7(b) shows the MTTDLs for different values of B from 0.1 to 1 (in units of Gbps) when $\lambda = 0.5$ per year. Under our settings, the MTTDL of FMSR codes is 50% to 80% longer than traditional RAID-6 codes due to a higher repair rate for a single-node failure. For example, with $\lambda = 0.5$ per year and $B = 1$Gbps, the MTTDL of FMSR codes is 76% longer.

4.3 Discussions

We now point out several open issues of the existing design of FMSR codes, and we pose them as future work.

Generalization of FMSR codes. We currently consider only an FMSR code implementation with double-fault tolerance (i.e., $k = n - 2$). Its correctness is also proven in our recent work [28]. While double-fault tolerance is the default setting of today's enterprise storage systems (e.g., 3-way replication in GFS [22]), it is unclear how to generalize FMSR codes for different (n, k) values. In addition, while single-node failures are the most common failure patterns in practical cloud storage systems [31], it is interesting to study how to generalize FMSR codes to support efficient repairs of concurrent node failures.

Study of different reliability metrics. In Section 4.2.3, we compare the reliability of FMSR codes and conventional RAID-6 codes for different failure rates using the MTTDL metric. An open issue is to model the failure rate of a cloud repository. In future work, we also plan to conduct further reliability analysis using more effective metrics [26].

Degraded reads. When reading the original data in failure mode, we perform degraded reads, in which we reconstruct the lost data of a failed node from the data available on the other surviving nodes. In FMSR codes, we always download the same amount of original data

by connecting to any k nodes (see Section 4.1.2); while in traditional RAID-6 codes, the original amount of data is retrieved to recover the lost data. Thus, FMSR codes and traditional RAID-6 codes retrieve the same amount of data in degraded reads, while FMSR codes have higher computational overhead in decoding (see Section 6.2.1). Recent studies [31], [35], [53] improve the degraded read performance for erasure-coded data. On the other hand, we do not consider degraded reads in this work since FMSR codes are designed for long-term archives that are rarely read.

5 NCCloud Design and Implementation

We implement NCCloud as a proxy that bridges user applications and multiple clouds. Its design is built on three layers. The *file system layer* presents NCCloud as a mounted drive, which can thus be easily interfaced with general user applications. The *coding layer* deals with the encoding and decoding functions. The *storage layer* deals with read/write requests with different clouds.

Each file is associated with a *metadata* object, which is replicated at each repository. The metadata object holds the file details and the coding information (e.g., encoding coefficients for FMSR codes).

NCCloud is mainly implemented in Python, while the coding schemes are implemented in C for better efficiency. The file system layer is built on FUSE [21]. The coding layer implements both RAID-6 and FMSR codes. Our RAID-6 code implementation is based on the Reed-Solomon code [52] (as shown in Figure 2(a)) for baseline evaluation. We use zfec [65] to implement the RAID-6 codes, and we utilize the optimizations made in zfec to implement FMSR codes for fair comparison.

Recall that FMSR codes generate multiple chunks to be stored on the same repository. To save the request cost overhead (see Table 3), multiple chunks destined for the same repository are aggregated before upload. Thus, FMSR codes keep only one (aggregated) chunk per file object on each cloud, as in RAID-6 codes. To retrieve a specific chunk, we calculate its offset within the combined chunk and issue a range GET request.

We make NCCloud deployable in one or multiple machines. In the latter case, we use ZooKeeper [32] to implement a distributed file-based shared lock to avoid simultaneous updates on the same file. We conduct preliminary evaluations in a LAN environment and observe that the overhead due to ZooKeeper is minimal. Here, we focus on deploying NCCloud on a single machine, and we mount NCCloud as a local file system.

6 Evaluation

We use our NCCloud prototype to evaluate RAID-6 codes (based on Reed-Solomon codes) and FMSR codes in multiple-cloud storage. In particular, we focus on the setting $k = n - 2$ for different values of n, and hence allow data retrieval with at most two cloud failures.

TABLE 3
Monthly price plans (in US dollars) for Amazon S3 (US Standard), Rackspace Cloud Files and Windows Azure Storage (North America and Europe), as of May, 2013.

	S3	RS	Azure
Storage (per GB)	$0.095	$0.10	$0.095
Data transfer in (per GB)	free	free	free
Data transfer out (per GB)	$0.12	$0.12	$0.12
PUT,POST (per 10K requests)	$0.05	free	$0.001
GET (per 10K requests)	$0.004	free	$0.001

The goal of our experiments is to explore the practicality of using FMSR codes in multiple-cloud storage. Our evaluation consists of two parts. We first compare the monetary costs of using RAID-6 and FMSR codes based on the price plans of today's cloud providers. We also empirically evaluate the response time performance of our NCCloud prototype atop a local cloud and also a commercial cloud provider.

Summary of evaluation results. We summarize our findings here. Our emphasis is on the monetary cost advantage of FMSR codes over RAID-6 codes, while still maintaining acceptable response time performance. In terms of monetary costs, we show that in normal operations, both RAID-6 and FMSR codes incur similar storage costs, while in the repair operation, FMSR codes save a significant amount of transfer costs over RAID-6 codes. In terms of response time, we demonstrate that both FMSR and RAID-6 codes have comparable response time performance (within 5%) when deployed on a commercial cloud (Azure). The resulting response time is mainly determined by the transmission performance of the Internet.

6.1 Cost Analysis

6.1.1 Repair Cost Saving

We first analyze the saving of monetary costs in repair in practice. Table 3 shows the monthly price plans for three major providers as of May 2013. We take the cost from the first chargeable usage tier (i.e., storage usage within 1TB/month; data transferred out more than 1GB/month but less than 10TB/month).

From the analysis in Section 3, we can save 25-50% of the download traffic during storage repair. The storage size and the number of chunks being generated per file object are the same in both RAID-6 and FMSR codes (assuming that we aggregate chunks in FMSR codes as described in Section 5). However, in the analysis, we have ignored two practical considerations: the size of metadata (Section 5) and the number of requests issued during repair. We now argue that they are negligible and that the simplified calculations based only on file size suffice for real-life applications.

Metadata size: Our implementation currently keeps the metadata size of FMSR codes within 160 bytes when $n = 4$ and $k = 2$, regardless of the file size. For a large n, say when $n = 12$ and $k = 10$, the metadata size

TABLE 4
Tiered monthly price plans (in US dollars) for both Amazon S3 (US Standard) and Windows Azure Storage (North America and Europe), as of May 2013.

Storage (per GB)	Data transfer out (per GB)
\$0.095 (First 1TB/month)	\$0.12 (First 10TB/month)
\$0.08 (Next 49TB/month)	\$0.09 (Next 40TB/month)
\$0.07 (Next 450TB/month)	\$0.07 (Next 100TB/month)
\$0.065 (Next 500TB/month)	\$0.05 (Over 150TB/month)
\$0.06 (Next 4000TB/month)	

is still within 900 bytes. NCCloud aims at long-term backups (see Section 3), and can be integrated with other backup applications. Existing backup applications (e.g., [19], [60]) typically aggregate small files into a larger data chunk in order to save the processing overhead. For example, the default setting for Cumulus [60] creates chunks of around 4MB each. Thus, the metadata size overhead can be made negligible. Since both RAID-6 and FMSR codes store the same amount of file data, they incur very similar storage costs in normal usage (assuming that the metadata costs are negligible).

Number of requests: From Table 3, some cloud providers charge for requests. RAID-6 and FMSR codes differ in the number of requests when retrieving data during repair. Suppose that we store a file object of size 4MB with $n = 4$ and $k = 2$. During repair, RAID-6 and FMSR codes retrieve two and three chunks, respectively (see Figure 2). The cost overhead due to the `GET` requests for RAID-6 codes is at most 0.171%, and that for FMSR codes is at most 0.341%, a mere 0.17% increase.

6.1.2 Case Study

We now explore the implications of our cost analysis using an enterprise use case. Our study builds on the case of Backupify, a cloud backup solution provider founded in 2008 and reported to store multiple terabytes to petabytes of backups on Amazon S3 and Glacier [3]. To simplify our analysis, let us assume that Backupify currently stores 1PB worth of backups in the cloud. Also, the data is stored over 10 cloud repositories with $n = 10$ and $k = 8$, giving a redundancy overhead of 25%. As we argue above, both RAID-6 and FMSR codes incur similar storage and data transfer costs, while FMSR codes incur less repair cost than RAID-6 codes. In particular, the percentage of cost saving of FMSR codes is $1 - \frac{n-1}{2(n-2)}$ (see Section 3), or equivalently, 43.75%. In the following, we consider two cost models.

Regular-cost storage model. When storing terabytes of data, cloud storage providers typically use a tiered pricing scheme that offers lower rates for higher usage. Table 4 shows a simplified tiered pricing scheme used by both Amazon S3 (US Standard) and Windows Azure (defaults for North America and Europe). We will use this tiered scheme for our cost calculation.

In our case, we have 1.25PB of data stored, and will be paying \$86,851 monthly storage cost for both RAID-6 and FMSR codes. If a cloud repository fails permanently and we run the repair operation, then RAID-6 codes will download 1PB of data, while FMSR codes will download only 0.5625PB of data. Thus, the corresponding repair cost for RAID-6 codes is \$56,832, while that for FMSR codes is \$33,894, with a saving of \$22,938.

Low-cost storage model. We point out that the repair cost can significantly exceed the monthly storage cost if an alternative low-cost storage model is used. Take Amazon Glacier [5] for example. It charges a flat rate of \$0.01 per GB of stored data, which is much cheaper than S3, while using the same data transfer pricing as S3 (see Table 4). The downside of using Glacier over S3 is that the restore operation takes much longer time and is more expensive. Glacier also charges a restore fee of \$0.01 per GB when restoring more than 5% of stored data per month.

Under this cost model, the monthly storage cost drops to only \$13,107 for both RAID-6 and FMSR codes. However, the repair cost for RAID-6 codes is \$66,662, while that for FMSR codes is \$39,137, with a saving of \$27,525.

Although we cannot pinpoint the failure rate of a cloud storage repository to accurately gauge the annual saving brought by the reduction in repair cost, we note that varying degrees of permanent data loss do occur in cloud storage in the last few years since its popular adoption by the masses (e.g., [12], [40], [58], [64]). If we estimate that full repairs have to be made every two years on average, this would translate to an annual saving of over \$10,000 for our case.

To conclude, we can see that although cloud failures are rare, the monetary benefits brought by FMSR codes in unexpected repair events can be significant. Another practical consideration that is not shown here is data accumulation. Our case study assumes a constant amount of data stored, but in reality, the amount of data may grow with time, such as when customers generate new data daily or when there are more customers using the cloud storage service. Such data accumulation leads to a larger archive size as time goes by, and will make our monetary advantage in repair cost more prominent.

6.2 Response Time Analysis

We deploy our NCCloud prototype in real environments. We evaluate the response time performance of three basic operations, namely file upload, file download, and repair, in two scenarios. The first part analyzes in detail the time taken by different NCCloud operations. It is done on a local cloud storage testbed in order to lessen the effects of network fluctuations. The second part evaluates how NCCloud actually performs in a commercial cloud environment. All results are averaged over 40 runs. We assume that repair coefficients are generated offline (see Section 4.2.2), so the time taken by two-phase checking is not accounted for in the repair operation. Nevertheless, we believe that this has limited impact on our results since the checks takes up little time compared to the overall repair operation, as shown in Section 4.2.2.

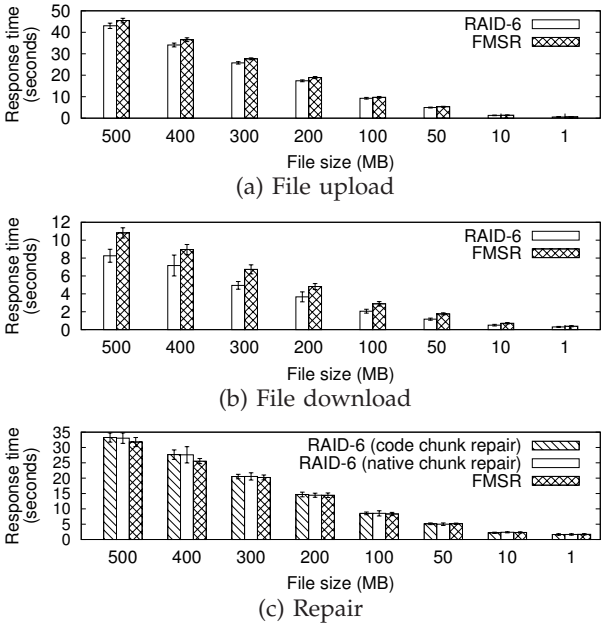


Fig. 8. Response times of NCCloud operations when $n = 4$ and $k = 2$.

6.2.1 On a Local Cloud

The experiments on local cloud are carried out on an object-based storage platform based on OpenStack Swift 1.4.2 [42]. NCCloud is mounted on a machine with an Intel Xeon E5620 2.4GHz CPU and 16GB RAM. The machine is connected to an OpenStack Swift platform attached to a number of storage servers, each with Intel Core i5-2400 and 8GB RAM. We create $(n+1)$ containers on Swift, so each container resembles a cloud repository (one of them is a spare node used in repair). We carry out two experiments on the local cloud. The first experiment compares RAID-6 and FMSR codes when $n = 4$ and $k = 2$ with varying file sizes. The second experiment compares RAID-6 and FMSR codes under different values for n and k with a fixed file size.

In the first experiment, we test the response times of the file upload, file download, and repair operations of NCCloud with $n = 4$ and $k = 2$. We use eight randomly generated files from 1MB to 500MB as the data set. We set the path of a chosen repository to a non-existent location to simulate a node failure in repair. Note that there are two types of repair for RAID-6, depending on whether the failed node contains a native chunk or a code chunk. Figure 8 plots the response times of all three operations (with 95% confidence intervals plotted) versus the file size.

In the second experiment, we fix the file size at 500MB and test the response times of the three operations again under four different sets of configurations for n and k: $n = 4$, $k = 2$; $n = 6$, $k = 4$; $n = 8$, $k = 6$; and $n = 10$, $k = 8$. Figure 9 shows the response time results, each broken down into several key constituents.

Figures 8 and 9 show that RAID-6 codes have less response time than FMSR codes in file upload and download, regardless of n and k. With the help of Figure 9, we pinpoint the overhead of FMSR codes over RAID-6. Due to having the same MDS property, RAID-6 and FMSR codes exhibit similar data transfer time during upload/download. However, FMSR codes display a noticeable encoding/decoding overhead over RAID-6 codes. For example in the case of $n = 4$ and $k = 2$, when uploading a 500MB file, RAID-6 codes take 1.53s to encode while FMSR codes take 5.48s; when downloading a 500MB file, no decoding is needed in the case of RAID-6 codes as the native chunks are available, but FMSR codes take 2.71s to decode. The differences increase with n and k.

On the other hand, FMSR codes have slightly less response time in repair. The main advantage of FMSR codes is that FMSR codes download less data during repair. For example, in repairing a 500MB file with $n = 4$ and $k = 2$, FMSR codes spend 4.02s in download, while the native-chunk repair of RAID-6 codes spends 5.04s.

Although RAID-6 codes generally have less response time than FMSR codes in a local cloud environment, we expect that the encoding/decoding overhead of FMSR codes can be easily masked by network fluctuations over the Internet, as will be shown next.

6.2.2 On a Commercial Cloud

The following experiment is carried out on a machine with an Intel Xeon E5530 2.4GHz CPU and 16GB RAM. The machine is running 64-bit Ubuntu 9.10. We focus on the setting $n = 4$ and $k = 2$, and repeat the three operations in Section 6.2.1 on four randomly generated files from 1MB to 10MB atop Windows Azure Storage [13]. On Azure, we create $(n+1) = 5$ containers to mimic different cloud repositories. The same operation for both RAID-6 and FMSR codes are run interleaved to lessen the effect of network fluctuation on the comparison due to different times of the day. It is important to note that although we have used only Azure in this experiment, the actual usage of NCCloud should stripe data over different providers and locations for better availability guarantees.

Figure 10 shows the results for different file sizes with 95% confidence intervals plotted. From the figure, we do not see distinct response time differences between RAID-6 and FMSR codes in all operations. Furthermore, on the same machine, FMSR codes take around 0.150s to encode and 0.064s to decode a 10MB file (not shown in the figures). These constitute roughly 3% of the total upload and download times (4.962s and 2.240s respectively). Given that the 95% confidence intervals for the upload and download times are 0.550s and 0.438s respectively, network fluctuation plays a bigger role in determining the response time. Overall, we demonstrate that FMSR codes do not have significant performance overhead over our baseline RAID-6 code implementation.

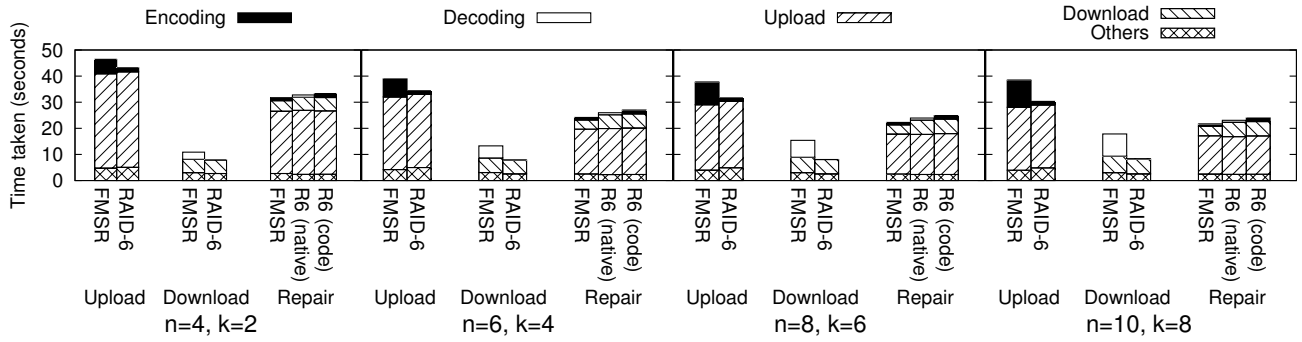


Fig. 9. Breakdown of the response time.

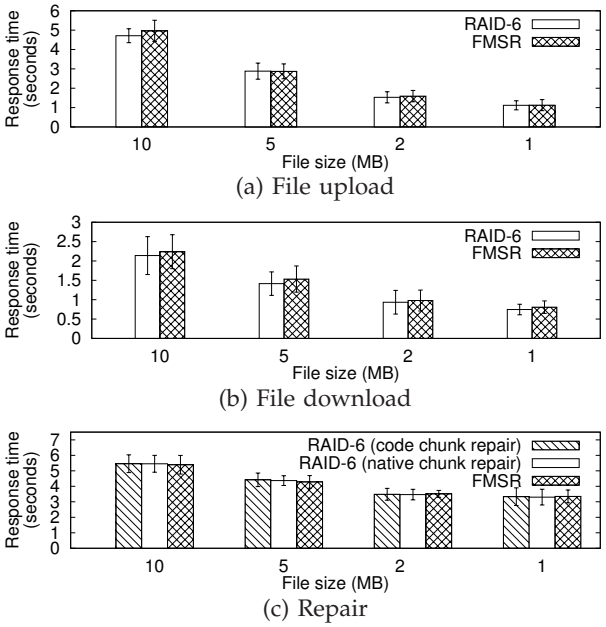


Fig. 10. Response times of NCCloud on Azure.

7 Related Work

We review the related work in multiple-cloud storage and failure recovery.

Multiple-cloud storage. There are several systems proposed for multiple-cloud storage. HAIL [11] provides integrity and availability guarantees for stored data. RACS [1] uses erasure coding to mitigate vendor lock-ins when switching cloud vendors. It retrieves data from the cloud that is about to fail and moves the data to the new cloud. Unlike RACS, NCCloud excludes the failed cloud in repair. Vukolić [61] advocates using multiple independent clouds to provide Byzantine fault tolerance. DEPSKY [10] addresses Byzantine fault tolerance by combining encryption and erasure coding for stored data. All the above systems are built on erasure codes to provide fault tolerance, while NCCloud takes one step further and considers regenerating codes with an emphasis on both fault tolerance and storage repair.

Minimizing I/Os. Several studies propose efficient single-node failure recovery schemes that minimize the amount of data read (or I/Os) for XOR-based erasure codes. For example, authors of [62], [63] propose optimal recovery for specific RAID-6 codes and reduce the amount of data read by up to around 25% (compared to conventional repair that downloads the amount of original data) for any number of nodes. Note that our FMSR codes can achieve 25% saving when the number of nodes is four, and up to 50% saving if the number of nodes increases. Authors of [35] propose an enumeration-based approach to search for an optimal recovery solution for arbitrary XOR-based erasure codes. Efficient recovery is recently addressed in commercial cloud storage systems. For example, new constructions of non-MDS erasure codes designed for efficient recovery are proposed for Azure [31] and Facebook [53]. The codes used in [31], [53] trade storage overhead for performance, and are mainly designed for data-intensive computing. Our work targets the cloud backup applications.

Minimizing repair traffic. Regenerating codes [16] stem from the concept of network coding [2] and minimize the repair traffic among storage nodes. They exploit the optimal trade-off between storage cost and repair traffic, and there are two optimal points. One optimal point refers to the *minimum storage regenerating (MSR)* codes, which minimize the repair bandwidth subject to the condition that each node stores the minimum amount of data as in Reed-Solomon codes. Another optimal point is the *minimum bandwidth regenerating (MBR)* codes, which allow each node to store more data to further minimize the repair bandwidth. The construction of MBR codes is found in [51], while that of MSR codes based on interference alignment is found in [50], [57]. In this work, we focus on the MSR codes.

On top of regenerating codes, several studies (e.g., [29], [34], [55], [56]) address cooperative recovery for multiple failures. Their idea is to have new nodes exchange reconstructed data to minimize the overall repair traffic. Our work focuses on single-failure recovery, which accounts for the majority of failures in cloud storage systems [31]. Some studies (e.g., [14], [41]) address the security issues for regenerating-coded data, while the security aspect of FMSR codes is addressed in our prior work [15]. We refer readers to the survey paper [17] for the state-of-the-art research in regenerating codes.

Existing MSR codes (e.g., [50], [57]) require nodes to perform encoding operations during repair. Our FMSR code implementation eliminates the encoding requirement of nodes, while maintaining the recovery perfor-

mance of MSR codes. The trade-off is that the codes are non-systematic (see Section 3).

Empirical studies on regenerating codes. Existing studies on regenerating codes mainly focus on theoretical analysis. Several studies (e.g., [18], [23], [37]) empirically evaluate random linear codes for peer-to-peer storage. Authors of [44] propose simple regenerating codes to minimize the number of surviving nodes to contact during recovery with a trade-off of incurring a higher storage cost, and evaluate the codes on a cloud storage simulator. Authors of [33] evaluate the encoding/decoding performance of regenerating codes. Existing studies do not implement a storage system and evaluate the actual read/write performance with regenerating codes as in our work. NCFS [30] implements regenerating codes, but does not consider MSR codes that are based on linear combinations. Here, we consider the FMSR code implementation, and perform empirical experiments in multiple-cloud storage.

Follow-up studies on FMSR codes. We have some follow-up studies after the conference version [27]. We extend NCCloud to support integrity checking of FMSR-coded data against Byzantine attacks [15]. We also theoretically prove that our two-phase checking can preserve the MDS property of the stored data after iterative repairs [28]. In this work, we focus on the practical deployment of regenerating codes. We propose an implementable design of regenerating codes and conduct empirical studies in practical cloud storage environment.

8 Conclusions

We present NCCloud, a proxy-based, multiple-cloud storage system that practically addresses the reliability of today's cloud backup storage. NCCloud not only provides fault tolerance in storage, but also allows cost-effective repair when a cloud permanently fails. NCCloud implements a practical version of the functional minimum storage regenerating (FMSR) codes, which regenerates new parity chunks during repair subject to the required degree of data redundancy. Our FMSR code implementation eliminates the encoding requirement of storage nodes (or cloud) during repair, while ensuring that the new set of stored chunks after each round of repair preserves the required fault tolerance. Our NCCloud prototype shows the effectiveness of FMSR codes in the cloud backup usage, in terms of monetary costs and response times. The source code of NCCloud is available at **http://ansrlab.cse.cuhk.edu.hk/software/nccloud**.

Acknowledgments

This work is supported by grants from the University Grants Committee of Hong Kong (AoE/E-02/08 and ECS CUHK419212) and seed grants from the CUHK MoE-Microsoft Key Laboratory of Human-centric Computing and Interface Technologies.

References

[1] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proc. of ACM SoCC*, 2010.
[2] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Trans. on Information Theory*, 46(4):1204–1216, Jul 2000.
[3] Amazon. AWS Case Study: Backupify. http://aws.amazon.com/solutions/case-studies/backupify/.
[4] Amazon. Case Studies. https://aws.amazon.com/solutions/case-studies/#backup.
[5] Amazon Glacier. http://aws.amazon.com/glacier/.
[6] Amazon S3. http://aws.amazon.com/s3.
[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.
[8] Asigra. Case Studies. http://www.asigra.com/product/case-studies/.
[9] AWS Service Health Dashboard. Amazon s3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.
[10] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DEPSKY: Dependable and Secure Storage in a Cloud-of-Clouds. In *Proc. of ACM EuroSys*, 2011.
[11] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proc. of ACM CCS*, 2009.
[12] Business Insider. Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data. http://www.businessinsider.com/amazon-lost-data-2011-4/, Apr 2011.
[13] B. Calder et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, 2011.
[14] B. Chen, R. Curtmola, G. Ateniese, and R. Burns. Remote Data Checking for Network Coding-Based Distributed Storage Systems. In *Proc. of ACM CCSW*, 2010.
[15] H. C. H. Chen and P. P. C. Lee. Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage. In *Proc. of IEEE SRDS*, 2012.
[16] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, Sep 2010.
[17] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A Survey on Network Codes for Distributed Storage. *Proc. of the IEEE*, 99(3):476–489, Mar 2011.
[18] A. Duminuco and E. Biersack. A Practical Study of Regenerating Codes for Peer-to-Peer Backup Systems. In *Proc. of IEEE ICDCS*, 2009.
[19] B. Escoto and K. Loafman. Duplicity. http://duplicity.nongnu.org/.
[20] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, 2010.
[21] FUSE. http://fuse.sourceforge.net/.
[22] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. of ACM SOSP*, 2003.
[23] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proc. of INFOCOM*, 2005.
[24] GmailBlog. Gmail back soon for everyone. http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html.
[25] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz. Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications. In *Proc. of IEEE MASCOTS*, 2008.
[26] K. M. Greenan, J. S. Plank, and J. J. Wylie. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In *Proc. of USENIX HotStorage*, 2010.
[27] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proc. of FAST*, 2012.
[28] Y. Hu, P. P. C. Lee, and K. W. Shum. Analysis and Construction of Functional Regenerating Codes with Uncoded Repair for Distributed Storage Systems. In *Proc. of IEEE INFOCOM*, Apr 2013.
[29] Y. Hu, Y. Xu, X. Wang, C. Zhan, and P. Li. Cooperative recovery of distributed storage systems from multiple losses with network coding. *IEEE JSAC*, 28(2):268–276, Feb 2010.
[30] Y. Hu, C.-M. Yu, Y.-K. Li, P. P. C. Lee, and J. C. S. Lui. NCFS: On the Practicality and Extensibility of a Network-Coding-Based Distributed File System. In *Proc. of NetCod*, 2011.

[31] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proc. of USENIX ATC*, 2010.

[33] S. Jiekak, A.-M. Kermarrec, N. L. Scouarnec, G. Straub, and A. Van Kempen. Regenerating Codes: A System Perspective. *CoRR*, abs/1204.5028, 2012.

[34] A. Kermarrec, N. Le Scouarnec, and G. Straub. Repairing Multiple Failures with Coordinated and Adaptive Regenerating Codes. In *Proc. of NetCod*, Jun 2011.

[35] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, 2012.

[36] N. Kolakowski. Microsoft's cloud azure service suffers outage. http://www.eweekeurope.co.uk/news/news-solutions-applications/microsofts-cloud-azure-service-suffers-outage-395.

[37] M. Martaló, M. Picone, M. Amoretti, G. Ferrari, and R. Raheli. Randomized Network Coding in Distributed Storage Systems with Layered Overlay. In *Information Theory and Application Workshop*, 2011.

[38] M. Mayer. This site may harm yoyur computer on every search results. http://googleblog.blogspot.com/2009/01/this-site-may-harm-your-computer-on.html.

[39] MSPmentor. CloudBerry Labs Unveils Support for Low-Cost Amazon Glacier. http://mspmentor.net/managed-services/cloudberry-labs-unveils-support-low-cost-amazon-glacier/, Jan 2013.

[40] E. Naone. Are We Safeguarding Social Data? http://www.technologyreview.com/blog/editors/22924/, Feb 2009.

[41] F. Oggier and A. Datta. Byzantine Fault Tolerance of Regenerating Codes. In *Proc. of P2P*, 2011.

[42] OpenStack Object Storage. http://www.openstack.org/projects/storage/.

[43] Panzura. US Department of Justice Case Study. http://panzura.com/us-department-of-justice-case-study/.

[44] D. Papailiopoulos, J. Luo, A. Dimakis, C. Huang, and J. Li. Simple Regenerating Codes: Network Coding for Cloud Storage. In *Proc. of IEEE INFOCOM*, Mar 2012.

[45] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of ACM SIGMOD*, 1988.

[46] J. S. Plank. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems. *Software - Practice & Experience*, 27(9):995–1012, Sep 1997.

[47] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In *Proc. of USENIX FAST*, 2009.

[48] C. Preimesberger. Many data centers unprepared for disasters: Industry group. http://www.eweek.com/c/a/IT-Management/Many-Data-Centers-Unprepared-for-Disasters-Industry-Group-772367/, Mar 2011.

[49] M. O. Rabin. Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM*, 36(2):335–348, Apr 1989.

[50] K. Rashmi, N. Shah, and P. Kumar. Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction. *IEEE Trans. on Information Theory*, 57(8):5227–5239, Aug 2011.

[51] K. V. Rashmi, N. B. Shah, P. V. Kumar, and K. Ramchandran. Explicit Construction of Optimal Exact Regenerating Codes for Distributed Storage. In *Proc. of Allerton Conference*, 2009.

[52] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[53] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proc. of VLDB Endowment*, 2013.

[54] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. of USENIX FAST*, Feb 2007.

[55] K. Shum. Cooperative Regenerating Codes for Distributed Storage Systems. In *Proc. of IEEE Int. Conf. on Communications (ICC)*, Jun 2011.

[56] K. Shum and Y. Hu. Exact Minimum-Repair-Bandwidth Cooperative Regenerating Codes for Distributed Storage Systems. In *Proc. of IEEE Int. Symp. on Information Theory (ISIT)*, Jul 2011.

[57] C. Suh and K. Ramchandran. Exact-Repair MDS Code Construction using Interference Alignment. *IEEE Trans. on Information Theory*, 57(3):1425–1442, Mar 2011.

[58] TechCrunch. Online Backup Company Carbonite Loses Customers' Data, Blames And Sues Suppliers. http://techcrunch.com/2009/03/23/online-backup-company-carbonite-loses-customers-data-blames-and-sues-suppliers/, Mar 2009.

[59] TechTarget. Cloud case studies: Data storage pros offer first-hand experiences. http://searchcloudstorage.techtarget.com/feature/Cloud-case-studies-Data-storage-pros-offer-first-hand-experiences/.

[60] M. Vrable, S. Savage, and G. Voelker. Cumulus: Filesystem backup to the cloud. In *Proc. of USENIX FAST*, 2009.

[61] M. Vukolić. The Byzantine Empire in the Intercloud. *ACM SIGACT News*, 41:105–111, Sep 2010.

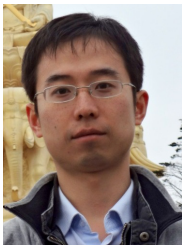
[62] Z. Wang, A. Dimakis, and J. Bruck. Rebuilding for Array Codes in Distributed Storage Systems. In *IEEE GLOBECOM Workshops*, 2010.

[63] L. Xiang, Y. Xu, J. Lui, Q. Chang, Y. Pan, and R. Li. A Hybrid Approach to Failed Disk Recovery Using RAID-6 Codes: Algorithms and Performance Evaluation. *ACM Trans. on Storage*, 7(3):11, 2011.

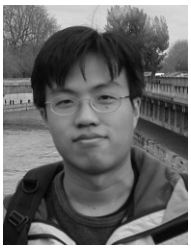
[64] ZDNet. AWS cloud accidentally deletes customer data. http://www.zdnet.com/aws-cloud-accidentally-deletes-customer-data-3040093665/, Aug 2011.

[65] zfec. http://pypi.python.org/pypi/zfec.

Henry C. H. Chen received his B.Eng. in Computer Engineering and M.Phil. in Computer Science and Engineering from the Chinese University of Hong Kong in 2010 and 2012 respectively. He is now a research assistant at the Chinese University of Hong Kong. His research interests are in security and applied cryptography.



Yuchong Hu received the B.S. degree in Computer Science and Technology from the School for the Gifted Young, University of Science & Technology of China, Anhui, China, in 2005. He received the Ph.D. degree in Computer Science and Technology from the School of Computer Science, University of Science & Technology of China, in 2010. He was a postdoctoral fellow at the Institute of Network Coding, the Chinese University of Hong Kong. His research interests include network coding and distributed storage.



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an assistant professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in cloud storage, distributed systems and networks, and security/resilience.

Yang Tang received the B.Eng. degree in Computer Science and Technology from Tsinghua University in 2009 and the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2011. He is currently working toward the Ph.D. degree in Computer Science at Columbia University. His research interests are in reliability/security of software systems and program analysis/verification.