NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds

Yuchong Hu[†], Henry C. H. Chen[†], Patrick P. C. Lee[†], Yang Tang[‡]
[†]*The Chinese University of Hong Kong*, [‡]*Columbia University*
ychu@inc.cuhk.edu.hk, {chchen,pclee}@cse.cuhk.edu.hk, ty@cs.columbia.edu

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, then we need to repair the lost data from other surviving clouds to preserve data redundancy. We present a proxy-based system for multiple-cloud storage called NCCloud, which aims to achieve cost-effective repair for a permanent single-cloud failure. NCCloud is built on top of network-coding-based storage schemes called regenerating codes. Specifically, we propose an implementable design for the functional minimum-storage regenerating code (F-MSR), which maintains the same data redundancy level and same storage requirement as in traditional erasure codes (e.g., RAID-6), but uses less repair traffic. We implement a proof-of-concept prototype of NCCloud and deploy it atop local and commercial clouds. We validate the cost effectiveness of F-MSR in storage repair over RAID-6, and show that both schemes have comparable response time performance in normal cloud storage operations.

1 Introduction

Cloud storage provides an on-demand remote backup solution. However, using a single cloud storage vendor raises concerns such as having a single point of failure [3] and vendor lock-ins [1]. As suggested in [1, 3], a plausible solution is to stripe data across different cloud vendors. While striping data with conventional erasure codes performs well when some clouds experience *short-term* failures or *foreseeable* permanent failures [1], there are real-life cases showing that permanent failures do occur and are not always foreseeable [23, 14, 20].

This work focuses on *unexpected* cloud failures. When a cloud fails permanently, it is important to activate *storage repair* to maintain the level of data redundancy. A repair operation reads data from existing surviving clouds and reconstructs the lost data in a new cloud. It is desirable to reduce the repair traffic, and hence the monetary cost, due to data migration.

Recent studies (e.g., [6, 8, 16, 25]) propose *regenerating codes* for distributed storage. Regenerating codes are built on the concept of network coding [2]. They aim to intelligently mix data blocks that are stored in existing storage nodes, and then regenerate data at a new storage node. It is shown that regenerating codes reduce the data repair traffic over traditional erasure codes subject to the same fault-tolerance level. Despite the favorable property, regenerating codes are mainly studied in the theoretical context. It remains uncertain regarding the practical performance of regenerating codes, especially with the encoding overhead incurred in regenerating codes.

In this paper, we propose *NCCloud*, a proxy-based system designed for multiple-cloud storage. We propose the *first* implementable design for the *functional minimum-storage regenerating code (F-MSR)* [8], and in particular, we eliminate the need of performing encoding operations within storage nodes as in existing theoretical studies. Our F-MSR implementation maintains double-fault tolerance and has the same storage cost as in traditional erasure coding schemes based on RAID-6, but uses less repair traffic when recovering a single-cloud failure. On the other hand, unlike most erasure coding schemes that are *systematic* (i.e., original data chunks are kept), F-MSR is *non-systematic* and stores only linearly combined code chunks. Nevertheless, F-MSR is suited to rarely-read long-term archival applications [6].

We show that in a practical deployment setting, F-MSR can save the repair cost by 25% compared to RAID-6 for a four-cloud setting, and up to 50% as the number of clouds further increases. In addition, we conduct extensive evaluations on both local cloud and commercial cloud settings. We show that our F-MSR 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 F-MSR via NCCloud, and motivates further studies of realizing regenerating codes in large-scale deployments.

2 Motivation of F-MSR

We consider a distributed, multiple-cloud storage setting from a client's perspective, such that we stripe data over multiple cloud vendors. We propose a proxy-based design [1] 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). That is, the

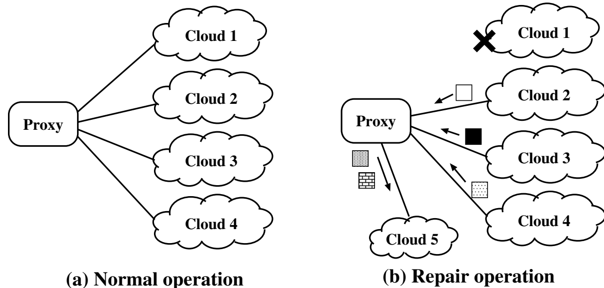


Figure 1: Proxy-based design for multiple-cloud storage: (a) normal operation, and (b) repair operation when Cloud node 1 fails. During repair, the proxy regenerates data for the new cloud.

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 *maximum distance separable (MDS)* codes. Given a file object, we divide it into equal-size *native chunks*, which in a non-coded system, would be stored on k clouds. With coding, the native chunks are encoded by linear combinations to form *code chunks*. The native and code chunks are distributed over $n > k$ clouds. When an MDS code is used, the original file object may be reconstructed from the chunks contained in *any* k of the n clouds. Thus, it tolerates the failure of any $n - k$ clouds. We call this feature the *MDS property*. The extra feature of F-MSR is that reconstructing a single native or code chunk may be achieved by reading up to 50% less data from the surviving clouds than reconstructing the whole file.

This paper considers a multiple-cloud setting that is double-fault tolerant (e.g., RAID-6) and provides data availability toward at most two cloud failures (e.g., a few days of outages [7]). That is, we set $k = n - 2$. We expect that such a fault tolerance level suffices in practice. Given that a permanent failure is less frequent but possible, our primary objective is to minimize the cost of storage repair for a permanent single-cloud failure, due to the migration of data over the clouds.

We define the *repair traffic* as the amount of outbound data being read from other surviving clouds during the single-cloud failure recovery. Our goal is 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 in many cloud vendors (see Table 1 in Section 5).

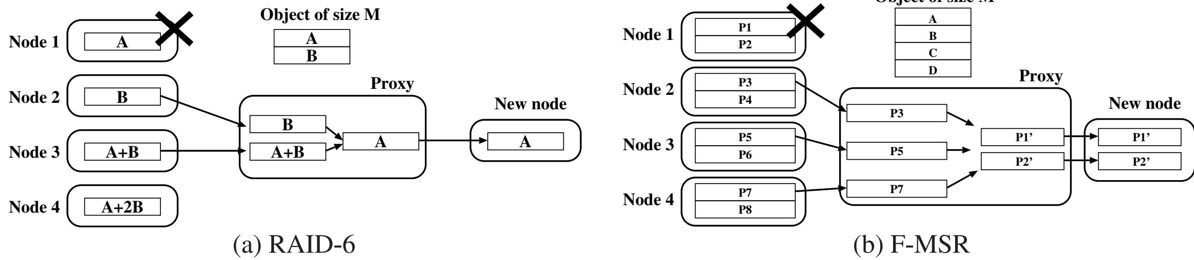
We now show how F-MSR saves the repair traffic via an example. Suppose that we store a file of size M on four clouds, each viewed as a *logical storage node*. Let us first consider RAID-6, which is double-fault tolerant. Here, we consider the RAID-6 implementation based on Reed-Solomon codes [26], as 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.

We now consider the double-fault tolerant implementation of F-MSR in a proxy-based setting, as shown in Figure 2(b). F-MSR divides the file into four native chunks, and constructs 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 F-MSR, the storage size is $2M$ (as in RAID-6), but the repair traffic is $0.75M$, which is 25% of saving.

To generalize F-MSR for n storage nodes, we divide a file of size M into $2(n-2)$ native chunks, and generate $4(n-2)$ 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 $n-1$ nodes, so the repair traffic is $\frac{M(n-1)}{2(n-2)}$. In contrast, for RAID-6, the total storage size is also $\frac{Mn}{n-2}$, while the repair traffic is M. When n is large, F-MSR can save the repair traffic by close to 50%.

Note that F-MSR keeps only code chunks rather than native chunks. To access a single chunk of a file, we need to download and decode the entire file for the particular chunk. Nevertheless, F-MSR is acceptable for long-term archival applications, whose read frequency is typically low [6]. Also, to restore backups, it is natural to retrieve the entire file rather than a particular chunk.

This paper considers the baseline RAID-6 implementation using Reed-Solomon codes. Its repair method is to reconstruct the whole file, and is applicable for *all* erasure codes in general. Recent studies [18, 28, 29] 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 [28, 29]. Although such approaches are suboptimal (recall that F-MSR can save up to 50% of repair traffic in RAID-6), their use of efficient XOR operations can be of practical interest.



(a) RAID-6 (b) F-MSR

Figure 2: Examples of repair operations in RAID-6 and F-MSR with n = 4 and k = 2.

3 F-MSR Implementation

In this section, we present a systematic approach for implementing F-MSR. We specify three operations for F-MSR on a particular file object: (1) file upload; (2) file download; (3) repair. A key difference of our implementation from prior theoretical studies is that we do not require storage nodes to have encoding capabilities, so our implementation can be compatible with today's cloud storage. Another key design issue is that instead of simply generating random linear combinations for code chunks (as assumed in [8]), we also guarantee that the generated linear combinations always satisfy the MDS property to ensure data availability, even after iterative repairs. Here, we implement F-MSR as an MDS code for general (n,k). We assume that each cloud repository corresponds to a logical storage node.

3.1 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 α_{ij} (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 ith row vector of $\mathbf{EM}$. We compute each P_i by the scalar product of ECV$_i$ and the native chunk vector (F_i), i.e., $P_i = \sum_{j=1}^{k(n-k)} \alpha_{ij} F_j$ 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 4). There are many ways of constructing $\mathbf{EM}$, as long as it satisfies the MDS property and the repair MDS property (see Section 3.3). Note that the implementation details of the arithmetic operations in Galois Fields are extensively discussed in [15].

3.2 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 F-MSR as a standard Reed-Solomon code, and our technique of creating an inverse matrix to decode the original data has been described in the tutorial [22].

3.3 Iterative Repairs

We now consider the repair of F-MSR for a file F for a permanent single-node failure. Given that F-MSR 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 rth 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 rth 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 property*). We now describe the rth 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 heuristic. 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 random ECV from each of the $n-1$ surviving nodes. Note that each ECV in $\mathbf{EM}$ corre-

sponds to a code chunk. We randomly 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 a $(n-k)\times(n-1)$ *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 [24].

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 given by:

$$i^{th} \text{ row vector of } \mathbf{EM}' = \begin{cases} \text{ECV}_i, & i \text{ is a surviving node,} \\ \text{ECV}'_i, & i \text{ is a new node.} \end{cases}$$

Step 5: Given $\mathbf{EM}'$, check if both the MDS and repair MDS 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 repair MDS property, we check that for *any* failed node (out of n nodes), we can collect *any* one out of $n-k$ chunks from 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 repair MDS property is at most $n(n-k)^{n-1}\binom{n}{k}$. If n is small, then the enumeration complexities for both MDS and repair MDS 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 claim that in addition to checking the MDS property, checking the repair MDS property is essential for iterative repairs. We conduct simulations to justify that checking the repair MDS property can make iterative repairs sustainable. In our simulations, we consider multiple rounds of permanent node failures for different values of n. Specifically, in each round, we randomly pick a node to permanently fail and trigger a repair. We say a repair is *bad* if the loop of Steps 2 to 5 is repeated over 10 times. We observe that without checking the repair MDS property, we see a bad repair very quickly, say after no more than 7 and 2 rounds for n = 8 and n = 12, respectively. On the other hand, checking the repair MDS property makes iterative repairs sustainable for hundreds of rounds for different values of n, and we do not yet find any bad repair after extensive simulations.

4 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 F-MSR).

NCCloud is mainly implemented in Python, while the storage schemes are implemented in C for better efficiency. The file system layer is built on FUSE [12]. The coding layer implements both RAID-6 and F-MSR. RAID-6 is built on zfec [30], and our F-MSR implementation mimics the optimizations made in zfec for a fair comparison.

Recall that F-MSR generates multiple chunks to be stored on the same repository. To save the request cost overhead (see Table 1), multiple chunks destined for the same repository are aggregated before upload. Thus, F-MSR keeps only one (aggregated) chunk per file object on each cloud, as in RAID-6. To retrieve a specific chunk, we calculate its offset within the combined chunk and issue a range GET request.

5 Evaluation

We now use our NCCloud prototype to evaluate RAID-6 (based on Reed-Solomon codes) and F-MSR in multiple-cloud storage. In particular, we focus on the setting $n = 4$ and $k = 2$. We expect that using $n = 4$ clouds may suffice for practical deployment. Based on this setting, we allow data retrieval with at most two cloud failures.

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

5.1 Cost Analysis

Table 1 shows the monthly price plans for three major vendors as of September 2011. For Amazon S3, we take the cost from the first chargeable usage tier (i.e., storage

	S3	RS	Azure
Storage (per GB)	$0.14	$0.15	$0.15
Data transfer in (per GB)	free	free	free
Data transfer out (per GB)	$0.12	$0.18	$0.15
PUT,POST (per 10K requests)	$0.10	free	$0.01
GET (per 10K requests)	$0.01	free	$0.01

Table 1: Monthly price plans (in US dollars) for Amazon S3 (US Standard), Rackspace Cloud Files and Windows Azure Storage, as of September, 2011.

usage within 1TB/month; data transferred out more than 1GB/month but less than 10TB/month).

From the analysis in Section 2, we can save 25% of the download traffic during storage repair when n = 4. The storage size and the number of chunks being generated per file object are the same in both RAID-6 and F-MSR (assuming that we aggregate chunks in F-MSR as described in Section 4). However, in the analysis, we have ignored two practical considerations: the size of metadata (Section 4) 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 F-MSR metadata size within 160B, regardless of the file size. NCCloud aims at long-term backups (see Section 2), and can be integrated with other backup applications. Existing backup applications (e.g., [27, 11]) typically aggregate small files into a larger data chunk in order to save the processing overhead. For example, the default setting for Cumulus [27] creates chunks of around 4MB each. The metadata size is thus usually negligible.

Number of requests: From Table 1, we see that some cloud vendors nowadays charge for requests. RAID-6 and F-MSR differ in the number of requests when retrieving data during repair. Suppose that we store a file of size 4MB with $n = 4$ and $k = 2$. During repair, RAID-6 and F-MSR retrieve two and three chunks, respectively (see Figure 2). The cost overhead due to the `GET` request for RAID-6 is at most 0.427%, and that for F-MSR is at most 0.854%, a mere 0.427% increase.

5.2 Response Time Analysis

We deploy our NCCloud prototype in real environments. We then evaluate the response time performance of different operations in two scenarios. The first part analyzes in detail the time taken by different NCCloud operations, and is done on a local cloud storage in order to lessen the effects of network fluctuations. The second part evaluates how NCCloud actually performs in commercial clouds. All results are averaged over 40 runs.

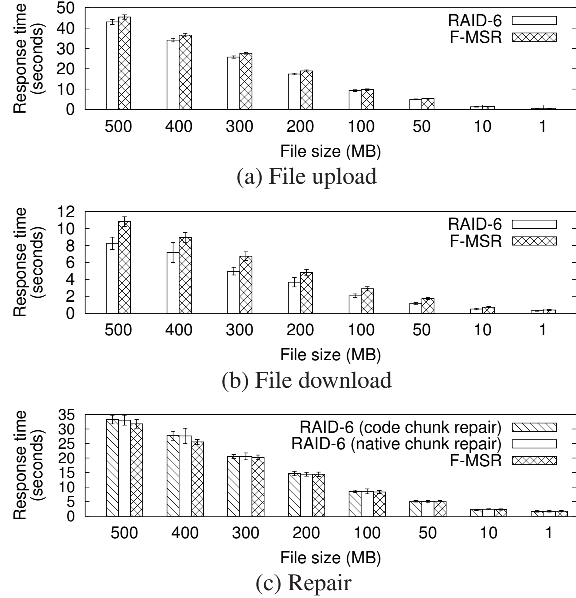


Figure 3: Response times of main NCCloud operations.

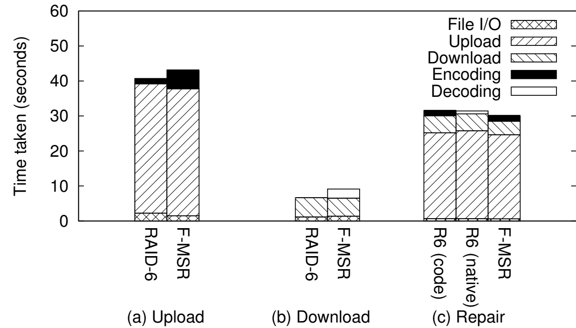
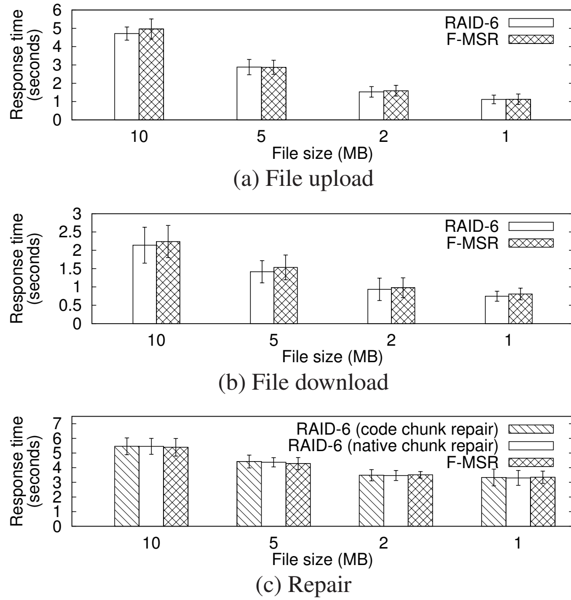


Figure 4: Breakdown of response time when dealing with 500MB file.

5.2.1 On a Local Cloud

The experiment on a local cloud is carried out on an object-based storage platform based on OpenStack Swift 1.4.2 [21]. NCCloud is mounted on a machine with Intel Xeon E5620 and 16GB RAM. This machine is connected to an OpenStack Swift platform attached with a number of storage servers, each with Intel Core i5-2400 and 8GB RAM. We create $(n+1) = 5$ containers on Swift, so each container resembles a cloud repository (one of them is a spare node used in repair).

In this experiment, we test the response time of three basic operations of NCCloud: (a) file upload; (b) file download; (c) repair. 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-existing 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.



(a) File upload

(b) File download

(c) Repair

Figure 5: Response times of NCCloud on Azure.

Figure 3 shows the response times of all three operations (with 95% confidence intervals plotted), and Figure 4 shows five key constituents of the response time when dealing with a 500MB file. Figure 3 shows that RAID-6 has less response time in file upload and download. With the help of Figure 4, we pinpoint the overhead of F-MSR over RAID-6. Due to having the same MDS property, RAID-6 and F-MSR exhibit similar data transfer time during upload/download. However, F-MSR displays a noticeable encoding/decoding overhead over RAID-6. When uploading a 500MB file, RAID-6 takes 1.490s to encode while F-MSR takes 5.365s; when downloading a 500MB file, no decoding is needed in the case of RAID-6 as the native chunks are available, but F-MSR takes 2.594s to decode.

On the other hand, F-MSR has slightly less response time in repair. The main advantage of F-MSR is that it needs to download less data during repair. In repairing a 500MB file, F-MSR spends 3.887s in download, while the native-chunk repair of RAID-6 spends 4.832s.

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

5.2.2 On a Commercial Cloud

The following experiment is carried out on a machine with Intel Xeon E5530 and 16GB RAM running 64-bit Ubuntu 9.10. We repeat the three operations in Section 5.2.1 on four randomly generated files from 1MB to 10MB atop Windows Azure Storage. On Azure, we create $(n+1) = 5$ containers to mimic different cloud repositories. The same operation for both RAID-6 and F-MSR are run interleaved to lessen the effect of network fluctuation on the comparison due to different times of the day. Figure 5 shows the results for different file sizes with 95% confidence intervals plotted. Note that although we have used only Azure in this experiment, actual usage of NCCloud should stripe data over different vendors and locations for better availability guarantees.

From Figure 5, we do not see distinct response time differences between RAID-6 and F-MSR in all operations. Furthermore, on the same machine, F-MSR takes 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 F-MSR does not have significant performance overhead over our baseline RAID-6 implementation.

6 Related Work

There are several systems proposed for multiple-cloud storage. HAIL [5] provides integrity and availability guarantees for stored data. DEPSKY [4] addresses Byzantine Fault Tolerance by combining encryption and erasure coding 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 be failed and moves the data to the new cloud. Unlike RACS, NCCloud excludes the failed cloud in repair. All the above systems are based on erasure codes, while NCCloud considers regenerating codes with an emphasis on storage repair.

Regenerating codes (see survey [9]) exploit the optimal trade-off between storage cost and repair traffic. Existing studies mainly focus on theoretical analysis. Several studies (e.g., [10, 13, 19]) empirically evaluate random linear codes for peer-to-peer storage. However, their evaluations are mainly based on simulations. NCFS [17] implements regenerating codes, but does not consider MSR codes that are based on linear combinations. Here, we consider the F-MSR implementation, and perform empirical experiments in multiple-cloud storage.

7 Conclusions

We present NCCloud, a multiple-cloud storage file system that practically addresses the reliability of today's cloud storage. NCCloud not only achieves fault tolerance of storage, but also allows cost-effective repair when a cloud permanently fails. NCCloud implements a practical version of the functional minimum storage regenerating code (F-MSR), which regenerates new chunks during

repair subject to the required degree of data redundancy. Our NCCloud prototype shows the effectiveness of F-MSR in accessing data, in terms of monetary costs and response times. The source code of NCCloud is available at **http://ansrlab.cse.cuhk.edu.hk/software/nccloud**.

8 Acknowledgments

We would like to thank our shepherd, James Plank, and the anonymous reviewers for their valuable comments. This work was supported by grant AoE/E-02/08 from the University Grants Committee of Hong Kong.

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] 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.

[4] 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.

[5] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In *Proc. of ACM CCS*, 2009.

[6] 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.

[7] CNNMoney. Amazon's cloud is back, but still hazy. `http://money.cnn.com/2011/04/25/technology/amazon_cloud/index.htm`.

[8] 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.

[9] 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.

[10] A. Duminuco and E. Biersack. A Practical Study of Regenerating Codes for Peer-to-Peer Backup Systems. In *Proc. of IEEE ICDCS*, 2009.

[11] B. Escoto and K. Loafman. Duplicity. `http://duplicity.nongnu.org/`.

[12] FUSE. `http://fuse.sourceforge.net/`.

[13] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proc. of INFOCOM*, 2005.

[14] GmailBlog. Gmail back soon for everyone. `http://gmailblog.blogspot.com/2011/02/gmail-back-soon-for-everyone.html`.

[15] 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.

[16] 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.

[17] 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.

[18] O. Khan, R. Burns, J. Plank, and C. Huang. In Search of I/O-Optimal Recovery from Disk Failures. In *USENIX HotStorage*, 2011.

[19] 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.

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

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

[22] 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.

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

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

[25] 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.

[26] 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.

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

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

[29] 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.

[30] zfec. `http://pypi.python.org/pypi/zfec`.