Efficient Proof of Retrievability Technique for Cloud Data Storage

Parth Shah [1], Nilax Patel [2], Dr. Amit Ganatra [3]

Department of Information Technology [1], U. & P. U. Patel Department of Computer Engineering [3]

Chandubhai S. Patel Institute of Technology,

CHARUSAT, Changa, India

U. & P. U. Patel Department of Computer Engineering [3]

K J Institute of Engineering and Technology, Savli, Vadodara [2]

parthshah.ce@charusat.ac.in [1], nilaxpatel8291@gmail.com [2], amitganatra.ce@charusat.ac.in [3]

**Abstract**— Recent advances have given rise to the popularity and success of cloud computing. Cloud storage enables users to remotely store their data and enjoy the on-demand high quality cloud applications without the burden of local hardware and software management. It moves the application software and databases to the centralized large data centres, where the management of the data and services may be error prone and not be fully trustworthy. This unique paradigm brings about many new security challenges. To protect outsourced data in cloud storage against corruptions, enabling integrity protection, fault tolerance, and efficient recovery for cloud storage becomes critical. Therefore, we study the problem of remotely checking the integrity of regenerating-coded data against corruptions under a real-life cloud storage setting. In this paper Data Integrity Protection (DIP) scheme is implemented atop the Functional Minimum Storage Regenerating Code (FMSR) code. We have also parallelized the DIP encoding operations in multi-threaded mode to reduce the running time of our DIP scheme in FMSR-DIP protocol [3]. FMSR-DIP which is a proof-of-concept prototype aimed at providing data integrity protection atop today’s cloud storage. FMSR-DIP augments the FMSR code with a data checking capability that allows stored data to be sampled for checking in a flexible manner, without adding to its download traffic requirements during file downloads or repairs. Finally we compared different techniques: Replication, Erasure codes (Reed Solomon Code) and regenerating code for various file operations and the result are compared.

Keywords— cloud computing, security, Proof of Retrievably, regenerating code, FMSR-DIP.

i. Introduction

Cloud computing defined as “A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamic and scalable services are delivered on demand to external customers over the Internet.” Cloud storage offers an on-demand data storage outsourcing service model, and is gaining popularity due to its elasticity, low maintenance and infrastructure cost.

One major use of cloud storage is long-term archival, which represents a workload that is written once and rarely read. While the stored data is rarely read, it remains necessary to ensure its integrity for disaster recovery, unauthorized access by honest but curious insider or outsiders. The notion of integrity in cloud computing concerns are both data integrity and computation integrity. Data integrity implies that data should be accurately stored on cloud servers, and any unauthorized modifications or updation (e.g., data is lost, altered, or compromised) are to be detected. Computation integrity implies the notion that programs are executed without being distorted by malware, cloud providers, or other malicious users, and that any incorrect computing should also be detected. So it is desirable to enable cloud clients to verify the integrity of their outsourced data in the cloud, in case their data has been accidentally corrupted or maliciously compromised by insider/outsider Byzantine attacks.

To meet the requirements of the massive volume of storage, erasure codes have gained a significant amount of attention in cloud systems. In this paper we introduce DIP scheme on FMSR code which provides additional security against mobile adversary and byzantine attack. We also evaluate running time for various operations of FMSR-DIP like upload, download and repair for the distinct values of that parameters.

The rest of the paper proceeds as follows. Section II reviews related work in remote data integrity checking. Section III provides description regarding necessary preliminaries for our design. Section IV presents the problem occurs in the existing system. Section V presents our proposed algorithms working behind this. Section VI provides the implementation details of FMSR-DIP. Section VII reports evaluation results of FMSR-DIP. Section VIII shows the comparison of FMSR-DIP with Replication and Reed Solomon code. Finally, Section IX concludes the paper.

ii. related work

There are various study performed to check the integrity of data, which are typical in long-term archival storage systems. This problem is first considered by Juels et al. [27] and Ateniese et al. [28], giving rise to the similar notions proof of retrievability (POR) and proof of data possession (PDP), respectively, which are proposed to verify the integrity of a large file by spot-checking only a fraction of the file via various cryptographic primitives. The basic POR scheme [27] embeds a set of pseudorandom blocks called sentinels into an encrypted file stored on the server, and the client can check if the server keeps the pseudorandom blocks later on. Error correcting codes are also included in the stored file to allow recovery of a small amount of errors within a file. However, the number of checks that the client can issue is limited by the number of the embedded random blocks. On the other hand, PDP [28] allows the client to keep a small amount of metadata. The client can then challenge the server against a set of random file blocks to see if the server returns the proofs that match the metadata on the client side. These both schemes are single server storage scheme. So in these methods whole the data are stored on a single server in which single-point-failure [2] and vendor-lock-ins [1] problems are arises. To overcome these problems one possible solution is to stripe data across multiple servers. Thus, to repair a failed server, we can (i) read data from other surviving servers, (ii) reconstruct the corrupted data of the failed server, and (iii) write the reconstructed data to a new server. MR-PDP [28] and HAIL [7] extend integrity checks to a multi-server setting using replication and erasure coding, respectively. In erasure coding based system (e.g. Reed Solomon Code) requires less storage overhead compare to Replication based system [24] for the same fault-tolerance level.

1. Replication Based system

Ensuring reliability requires the introduction of redundancy. The simplest form of redundancy is replication, which is adopted in many practical storage systems. In which k identical copies of each data object are kept at each instant by system members. Figure 1 shows an example of replication based system [24].

**S1 F**

**F**

**S2 F**

**S3 F**

**S4 F**

**S' F**

**New Replica Created**

**2 MB**

**S4 Fails**

Fig. 1 Replication based distributed system

Simple replication offers one avenue to higher-assurance data archiving. Only single copy of file is required to repair any node. For example if any node fails then simply copy the replica of that file from healthy node and store it on new node. But it requires often unnecessarily and unsustainably high expense. The storage cost for replication based system is very high.

1. Erasure Coding Based (Reed Solomon Code) system

As a generalization of replication, erasure coding offers better storage efficiency. For instance, we can divide a file of size M into k pieces (to be called fragments), each of size M/k, encode them into n encoded fragments (of the same size) using an (n, k) maximum distance separable (MDS) code, and store them at n nodes. Then, the original file can be recovered from any set of k coded fragments. This performance is optimal in terms of the redundancy–reliability trade-off because k pieces, each of size M/k, provide the minimum data for recovering the file, which is of size M. Example of (4, 2) Erasure coding based system is shown in figure 2. In which repair traffic of the system if M which is same as our file size.

**File of size M**

**(n,k) MDS property:** any k out of n servers can rebuild original file

Fig. 2 Reed Solomon Erasure Coding

**Proxy**

**n=4 , k=2**

**Reed Solomon codes**

**Repair traffic = M**

**B**

**A**

**A**

**A**

**A+B**

**B**

**A+2B**

**A+B**

**B**

**A**

1. Regenerating Coding Based system

For an erasure coded system, a common practice to repair from a single node failure is for a new node to reconstruct the whole encoded data object to generate just one encoded block. This is clearly an inefficient way of regeneration, since the network bandwidth is often a critical resource. This has motivated the development of family of codes, referred as regenerating codes, designed to carry out the regeneration efficiently. Regenerating codes [20] have been proposed to minimize this repair traffic (i.e., the amount of data being read from surviving servers). In essence, they achieve this by not reading and reconstructing the whole file during repair as in traditional erasure codes, but instead reading a set of chunks smaller than the original file from other surviving servers and reconstructing only the lost (or corrupted) data chunks. Regenerating codes are constructed systematically such that the source symbols are stored in k nodes called as data nodes, the remaining n-k nodes are called as parity nodes which contain symbols obtained through suitable encoding operation. Such systematic codes, where the data nodes are regenerated exactly but only functionally equivalent form of parity nodes are regenerated. Example of (4,2) Regenerating code is shown in figure 3. In which repair traffic is 0.75M which is less than the Reed Solomon Erasure coding based system.

**File of size M**

**Regenerating codes**

**Repair traffic = 0.75M**

**B**

**A**

**P2**

**P1**

**A**

**A**

**Proxy**

**D**

**C**

**A**

**P4**

**P3**

**A**

**A**

**P3**

**A**

**P3**

**A**

**P3**

**A**

**P1'**

**A**

**P5**

**P6**

**A**

**A**

**A**

**P2'**

**P7**

**P8**

**A**

**A**

**n=4 , k=2**

Fig. 3 Regenerating Code based system

iii. Preliminaries

1. Functional Minimum Storage Regenerating Code

FMSR [10] belongs to Maximum Distance Separable (MDS) codes. An MDS code is defined by the parameters (n, k), where k < n. It encodes a file F of size |F| into n pieces of size |F|/k each. An (n, k)-MDS code states that the original file can be reconstructed from any k out of n pieces (i.e., the total size of data required is |F|). An extra feature of FMSR is that a specific piece can be reconstructed from data of size less than |F|. FMSR is built on regenerating codes, which minimize the repair bandwidth while preserving the MDS property based on the concept of network coding. FMSR codes have three design properties, which we elaborate below.

1. *Preserve the fault tolerance and storage efficiency of MDS Codes*

MDS codes are defined by two parameters n and k (k < n). An (n, k)-MDS code divides a file of size M into k pieces of size M/k each, and encodes them into n pieces such that any k out of n encoded pieces suffice to recover the original file. By storing the n encoded pieces over n nodes, a storage system can tolerate at most n − k node failures. An example of MDS codes is Reed-Solomon codes [19].

1. *FMSR codes minimize the repair bandwidth*

If a node fails, we must reconstruct the lost data of the failed node to preserve fault tolerance. The conventional repair of Reed-Solomon codes reads k pieces from any k surviving nodes to restore the original file (by the design of MDS codes). Clearly, the amount of data read is the file size M. FMSR codes seek to read less than M units of data to reconstruct the lost data. FMSR codes are designed to match the minimum storage point of regenerating codes when repairing a node failure, while having each node store M/k units of data as in Reed-Solomon codes. To repair a failed node in FMSR codes, each surviving node transfers data of size  units or equivalently, a size of one parity chunk. In a special case of n = 4 and k = 2, the repair bandwidth is 0.75M, i.e., 25% less than that of conventional repair of Reed-Solomon codes. In general, the repair bandwidth of FMSR codes for k = n − 2 is  , and its saving compared to RAID-6 codes [15] (which are also double-fault tolerant) is up to 50% if n is large.

*3: FMSR codes use uncoded repair*

During repair, each surviving node under FMSR codes transfers one parity chunk, without any encoding operations. This also minimizes the amount of data read from disk.

1. NCCloud

NCCloud (formerly known as CloudNCFS) [10],[22] is a proof-of-concept prototype of a network-coding-based file system that aims at providing fault tolerance and reducing data repair cost when storing files using multiple-cloud storage (or any other kinds of raw storage devices). NCCloud is a proxy-based file system that interconnects multiple (cloud) storage nodes. It can be mounted as a directory on Linux, and file uploading/downloading are done by copying files to/from the mounted directory. NCCloud is built on FUSE, an open-source, programmable user-space file system that provides application programmable interfaces (APIs) for file system operations. From the point of view of user applications, NCCloud presents a file system layer that transparently stripes data across storage nodes. Network codes for storage repair require that storage nodes encode the stored data during the repair process. However, this may not be feasible for some storage systems where nodes only provide the basic I/O functionalities but do not have the encoding capability. Our work is to adapt the benefits of network codes in the storage repair of a practical storage setting, by relaxing the encoding requirement of storage nodes. NCCloud supports a variety of coding schemes, in particular the Functional Minimum Storage Regenerating (F-MSR) codes. Compared to traditional optimal erasure codes (e.g., Reed-Solomon), FMSR codes maintains the same storage overhead under the same data redundancy level, but uses less repair traffic during the recovery of a single failed storage node. NCCloud realizes regenerating codes in a practical cloud storage system that does not require any encoding/decoding intelligence on the cloud storage nodes.

iv. Problems in existing system

As it is noted in the different techniques of checking integrity of cloud storage data in analysis part there are some drawbacks in the existing system, which are like not secure against byzantine mobile adversary. Mobile Byzantine means

that the adversary compromises a subset of servers in different time epochs (i.e., mobile) and exhibits arbitrary behaviors on the data stored in the compromised servers (i.e., Byzantine). To ensure file availability, we assume that the adversary can compromise and corrupt data in at most n−k out of the n servers in any epoch, subject to the (n, k)-MDS fault tolerance requirement. At the end of each epoch, the client can ask for randomly chosen parts of remotely stored data and run a probabilistic checking protocol to verify the data integrity. Servers under the control of the adversary may or may not correctly return data requested by the client. If corruption is detected, then the client may trigger the repair phase to repair corrupted data. Instead of performing whole-file checking, which incurs a substantial transfer overhead, it is only feasible for the client to randomly sample data for integrity checking. The adversary may corrupt a small portion of data within the error-correcting capability in each epoch, but the level of corruption can render the errors unrecoverable after several epochs if they are not spotted early. This leads to creeping corruption [12]. Thus, it is necessary that the client can quickly spot the corrupted data without accessing the whole file.

v. design

Our goal is to augment the basic file operations Upload, Download, and Repair of NCCloud [10] with the DIP feature. to preserve data redundancy. NCCloud [10] is a proxy-based storage system for fault-tolerant multiple-cloud storage, 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. Our DIP scheme operates on the FMSR code chunks generated by NCCloud [22], which

is deployed as a client-side proxy that stripes data among multiple servers which is shown in figure 4.



Fig. 4 Integration of DIP scheme with NCCloud

Our DIP scheme is built on several cryptographic primitives, which include (i) Symmetric encryption, (ii) Pseudorandom functions (PRFs), (iii) Pseudorandom permutations (PRPs), and (iv) message authentication codes (MACs). Each of these primitives takes a secret key thus it means that it is computationally infeasible for an adversary to break the security of a primitive without knowing its corresponding secret key.

There is also need of a systematic adversarial error-correcting code (AECC) [5], [18] to protect against the corruption of a chunk. When a large file is encoded using ECC, it is first broken down into smaller stripes and then ECC is applied independently on a single stripe. AECC uses a family of PRPs as a building block to randomize the stripe structure so that it is computationally infeasible for an adversary to target and corrupt any particular stripe. Here both FMSR and AECC provide fault tolerance but the only difference is that FMSR applied to a file that is striped across servers, while AECC applies to a single chunk stored within a server.

a3

g

f

e

d

c

b3

h

8

6

2

3

2

1

7

5

(4,2) RS

Fragment 4

Fragment 3

Fragment 2

Fragment 1

(a) Standard ECC

f

g

h

e

a

d

b

c

7

6

5

4

3

2

7

6

5

4

3



1

1

8

8

PRP

PRP

a

g

f

e

d

c

b3

h

(4,2) RS

7

5

8

6

2

4

3

1

(b) Adversarial ECC

Fig. 5 Comapring Standard ECC with Adversarial ECC

As shown in figure 4 NCCloud generates code chunks for a file based on FMSR. The code chunks will be temporarily stored in the local file system instead of being uploaded to the servers. The DIP module then reads the FMSR code chunks from the local file system, encodes them with DIP, and passes the resulting FMSR-DIP code chunks to the storage interface module, which will upload the FMSR-DIP chunks to multiple servers (or a cloud-of-clouds). Figure 6 illustrates the entire file upload process in NCCloud using FMSR codes for n=4 and k=2. In which the DIP encoding module is added after encoding the native chunks using FMSR. Here data is stored in four servers, in which the data of any two servers suffice to recover the original file. Each server stores two code chunks of total size .

P3

P6

P5

P7

P8

P2

P1

P4

Server 1

P5

P6

P7

P8

P3

P4

P1

P2

F1

F2

F

F3

F4

encode

partition

Server 4

Inside NCCloud

Server 2

distribute

Server 3

FMSR

native chunks

size: |F|/4 each

FMSR

code chunks

size: |F|/4 each

Outsourced storage

size: |F|/2 on each server

Fig. 6 File uploading process in FMSR-DIP model

In this file uploading process code chunks are generated by applying encoding process on native chunks. In this encoding process native chunks are multiplied with encoding matrix. This whole process in shown in figure 7. Here for encoding matrix vandermonde matrix is used.

**File**

P2,1, ........., P2,b

F= F1...........F4

P3,1, ........., P3,b

F4,1 ............ F4,b

F1,1 ............ F1,b

F2,1 ............ F2,b

F3,1 ............ F3,b

P4,1, ........., P4,b

α1,4

α1,1

**=**

P5,1, ........., P5,b

α8,4

α8,1

P6,1, ........., P6,b

P7,1, ........., P7,b

P8,1, ........., P8,b

P1,1, ........., P1,b

**FMSR encoding coefficients**

**FMSR native chunks**

**FMSR code chunks**

Fig. 7 Encoding process using matrix multiplication

Now figure 8 shows an overview of how we augment FMSR code chunk into an FMSR-DIP code chunk. First our file is divided into fragments and then ECC is applied to those native chunks. After generating these FMSR code chunks Pi AECC is applied to those chunks and new code chunks Pi’ are generated. Once these code chunks are generated, they are uploaded on multiple servers. Total  number of code chunks are generated from the  number of native chunks. And these code chunks are uploaded on n number of servers.

**FMSR code chunk P1**

**P1,1, ... , P1,b**

(n',k') AECC

**FMSR code chunk**

**with (n',k') AECC**

**P1,1, ... , P1,b , ... , P1,b'**

Apply PRF

to each byte

**FMSR-DIP code chunk P'1**

**P'1,1, ... , P'1,b , ... , P'1,b'**

Fig. 8 Augmentation of FMSR code chunk into an FMSR-DIP

code chunk

vi. algorithm working behind the system

a. *Upload*

|  |
| --- |
| **Step 1:** Generate the per-file secrets. Before uploading F, generate per-file secrets. |
| **Step 2:** Divide the file F into k(n−k) equal size native chunks. |
| **Step 3:** Produce encode matrix (Vandermonde matrix) |
| **Step 4:** Encode these k(n − k) native chunks into n(n − k) code chunks with FMSR. |
| **Step 5:** Encode each code chunk with FMSR-DIP.   * Apply AECC to the b bytes of each code chunk * Then apply PRF to all b’ bytes of code chunk      * Compute a MAC Mi for first b bytes of code chunk. |
| **Step 6:** Append n', k' and also MACs of all chunks to metadata and upload the file. |

*b. download*

|  |
| --- |
| **Step 1:** First download the corresponding metadata object that contains the ECVs |
| **Step 2:** Select any k of the n storage nodes, and download the k(n−k) code chunks |
| **Step 3:** Form a k(n−k)×k(n−k) square matrix from code chunks |
| **Step 4:**  Calculate the inverse of the matrix |
| **Step 5:** Multiply the inverse of the square matrix with the code chunks and obtain the original k(n − k) native chunks |

*c. repair*

|  |
| --- |
| For example if node 1(code chunk P1 and P2 fails) |
| Step 1: Get all the existing ECVs |
| Step 2: Randomly select ECVs from each existing nodes |
| Step 3: Randomly generate a repair matrix: EM |
| Step 4: Obtain ECVs in new node: |
| Step 5: Construct new EM’ and test it:    Check both MDS and repair MDS property in EM’. If it fails then go to step2 otherwise go to step 6. |
| Step 6: Download P3, P5, P7; Regenerate |

vii. implementation details

In this section we first conduct test-bed experiments on a local storage, we can also implement it on the local cloud storage that is built on OpenStack Swift 1.4.2 [23], Hadoop Distributed System etc. We deploy our FMSR-DIP implementation in single-threaded mode on a machine equipped with Intel Core i3, 8GB RAM, and 64-bit Ubuntu 12.04.

We implement all cryptographic operations using OpenSSL 1.0.0g [23]. All cryptographic primitives use 128-bit secret keys. We require that all secret keys be securely stored on the client side without being revealed to any server. Since the files

in the cloud are typically of large size, we assume that the secret keys only incur a small constant overhead.

*A. The primitives are instantiated as described below.*

* ***Symmetric encryption :*** For symmetric encryption AES-128 is used in cipher-block chaining (CBC) mode.
* ***Pseudorandom function :***AES-128 is used as a pseudorandom function. The PRF input is first transformed to a plaintext block, which is then encrypted with AES-128.
* ***Pseudorandom permutation :*** PRP implementation is based on AES-128, but applied in a different way as in PRF. Note that the domain size of the PRP is the number of elements to be permuted. To implement a PRP with a small and flexible domain size, the approach in Method 1 of [10] is used. In that first a list of indices from 0 to d − 1 is created, where d is the desired domain size of our PRP. Then encrypt each index in turn with AES-128 and sort the encrypted indices. Finally, the permutation is given by the positions of the original indices in the sorted list of encrypted indices.
* ***Message authentication codes :*** HMAC SHA-1 is used to compute MACs.
* ***Adversarial error-correcting codes :*** Here the systematic AECC is applied as described in Section V. First, for efficiency, do not encrypt the AECC parities, since PRF will be applied to the entire DIP-encoded chunk after applying AECC. PRF itself serves as an encryption. Second, and most notably, instead of applying a single PRP to the entire code chunk, first divide the code chunk into k' fragments, and apply a different PRP to each fragment. The secret key of the PRP for each fragment is formed by the XOR sum of a master PRP secret key and the fragment number. Applying PRP to a fragment rather than a chunk reduces the domain size and hence the overall memory usage.

viii. running time & cost analysis

We focus on measuring the running time of each operation. We assume that all file objects being processed remain intact (i.e., without corruptions) throughout an operation, so that we can measure the overhead of FMSR-DIP in normal usage. By default we evaluate the results of all operations for 100MB file, (4,2)-FMSR, (110,100) AECC, and a block size of 256B for both PRP and PRF. We can vary one set of these parameters each time, while fixing the other three sets at default values. In the above results we vary the file size and fix all three parameters to default values.

(a) Running time for different values of (n,k)

(b) Running time for different values of (n',k')

(c) Running time for different values of (PRP,PRF)

Fig. 11 Running time of FMSR-DIP for different set of parameters

From above figure we can analyse that the different parameters effect the running time of operations. As shown in figure 11 (a) running time of operations are increased whenever we increase the values of (n,k). Here we take three different values of (n,k). As shown in figure 11 (b) and figure 6.4 (c) running time is decreased whenever we change the parameters of AECC (n’,k’) and block size of PRP and PRF. So in this section we see We see that the monetary cost of cloud storage is mainly attributed to three components: (i) amount of storage, (ii) amount of data transfer outbound from the cloud, and (iii) number of requests made.

viii. comparison of fmsr-dip with replication and reed solomon code

Finally we evaluate the running time of Replication based system and Reed Solomon Erasure codes for compare them with FMSR-DIP code. From that we make the comparison of all these three different techniques and analysis the overhead of DIP encoding and also advantage of that during repairing time.

*a. response time of upload operation*

The major source of the monetary overhead of our DIP scheme compared to NCCloud is (n',k') AECC, which expands the stored data and increases the storage cost by roughly n’/k’ (note that the inbound transfer cost is free for all commercial cloud providers that we consider). The cost due to the expanded file metadata is a negligible constant if the file size is large enough. For example, when using (4,2)-FMSR, our encrypted metadata size is 320B, which is 160B more than the current NCCloud implementation. Furthermore, some cloud providers such as Rackspace and Azure allow a small metadata to be associated with an uploaded object for free.

Fig. 12 Comparison of running time for upload operation

*b. response time of download operation*

From the figure we can say that running time of FMSR-DIP for download is smaller than replication based system but it is greater than FMSR because of overhead of DIP encoding. But when no corrupted data is detected, we do not have to download the AECC parities. Thus, the monetary cost incurred by DIP is similar to NCCloud. Our DIP scheme adds a small constant overhead (independent of the file size) in downloading the metadata.

Fig. 13 Comparison of running time for download operation

*a. response time of repair operation*

The major monetary overhead again comes from (n', k')-AECC in encoding the new FMSR code blocks. As discussed above, if there is no corrupted data in surviving servers, we preserve the network transfer cost of NCCloud when downloading data from the surviving servers (aside from the small constant metadata traffic). Also, the inbound transfer cost of writing reconstructed FMSR-DIP chunks to a new server is free for many commercial cloud storage providers [10]. Therefore, we still preserve the cost saving property of the repair operation in NCCloud when compared to the conventional repair method (by up to 50% for RAID-6 [9]).

Fig. 14 Comparison of running time for repair operation

vi. Conclusions

Seeing the popularity of outsourcing archival storage to the cloud, it is desirable to enable clients to verify the integrity of their data in the cloud. We study design of data integrity protection (DIP) scheme for functional minimum storage regenerating (FMSR) codes under a multi-server setting. This DIP scheme preserves the fault tolerance and repair traffic saving properties of FMSR. And also it allows clients to remotely verify the integrity of random subsets of long term archival data under multi-server cloud storage setting. We evaluate the running time under various parameter choices. We compare the three different techniques for checking integrity and fault tolerance of system and also evaluate the overhead of DIP scheme. Our DIP scheme preserves the fault tolerance and also less repair traffic.

There are 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). While double-fault tolerance is the default setting of today’s enterprise storage systems 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, it is interesting to study how to generalize FMSR codes to support efficient repairs of concurrent node failures.

**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.

**Integration of modules:** Our current implementation uses a modular approach that separates the FMSR code module (i.e., NCCloud) and the DIP module. It is possible to combine the two modules into a single design to reduce the overhead, so as to eliminate the passing of FMSR code chunks between NCCloud and our DIP module using a local staging directory. In addition, we may exploit certain inherent properties of such combination to further reduce the computational overhead, and we pose this issue as our extended work. On the other hand, our modular approach allows us to flexibly enable DIP

on demand in real deployment.

References

1. H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In Proc. of ACM SoCC, 2010.
2. 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.
3. J. Breckling, Ed., The Analysis of Directional Time Series: Applications to Wind Speed and Direction, ser. Lecture Notes in Statistics. Berlin, Germany: Springer, 1989, vol. 61.
4. Henry C. H. Chen, Patrick P. C. Lee, "Enabling Data Integrity Protection in Regenerating-Coding-Based Cloud Storage: Theory and Implementation," IEEE Transactions on Parallel and Distributed Systems, 24 July 2013. IEEE computer Society Digital Library. IEEE Computer Society.
5. Dimitris S. Papailiopoulos, Jianqiang Luo, Alexandros G. Dimakis, Cheng Huang, Jin Li. Simple Regenerating Codes: Network Coding for Cloud Storage, Sep 2011.
6. K. Bowers, A. Juels, and A. Oprea. Proofs of Retrievability: Theory and Implementation. In Proc. of ACM CCSW, 2009.
7. zfec. http://pypi.python.org/pypi/.
8. K. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In Proc. of ACM CCS, 2009.
9. 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.
10. Intel. Intelligent RAID-6 Theory Overview and Implementation, 2005.
11. Chen, H.C.H.; Yu chong Hu; Lee, P.P.C.; Yang Tang, "NCCloud: A Network-Coding-Based Storage System in a Cloud-of-Clouds," Computers, IEEE Transactions on , vol.63, no.1, pp.31,44, Jan. 2014.
12. R. Li, J. Lin, and P. Lee. CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems. arXiv, preprint arXiv:1302.3344, 2013.
13. K. D. Bowers, A. Juels, and A. Oprea. HAIL: A High-Availability and Integrity Layer for Cloud Storage. In Proc. of ACM CCS, 2009.
14. 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.
15. N Cao, S Yu, Z Yang, W Lou, YT Hou. LT codes-based secure and reliable cloud storage service- INFOCOM, 2012 Proceeding IEEE, 2012.
16. 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.
17. 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.
18. J. Black and P. Rogaway. Ciphers with arbitrary finite domains. In *Topics in Cryptology – CT-RSA 2002*, volume 2271 of *LNCS*, pages 114–130. Springer, 2002.
19. R. Curtmola, O. Khan, and R. Burns. Robust remote data checking. In *Proc. of ACM StorageSS*, 2008.
20. 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.
21. A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on* *Information Theory*, 56(9):4539–4551, 2010.
22. Hakim Weatherspoon and John D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison University of California, Berkeley.
23. Y. Hu, H. Chen, P. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In Proc. of USENIX FAST, 2012.
24. OpenStack Object Storage. http://www.openstack.org/projects/storage/.
25. Oktay Olmez, Aditya Ramamoorthy. Replication based storage systems with local repair. Network Coding (NetCod), 2013 International Symposium on. 7-9 June 2013.
26. M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative Internet backup scheme. In Proc. USENIX Annual Technical Conference, General Track 2003, pages 29—41, 2003.
27. A. Juels and B. Kaliski. PORs: Proofs of retrievability for large files. In Proc. ACM CCS, pages 584–597, 2007.
28. G. Ateniese, R. Burns, R. Curtmola, J. Herring, O. Khan, L. Kissner, Z. Peterson, and D. Song. Remote Data Checking Using Provable Data Possession. ACM Trans. on Information and System Security, May 2011.
29. R. Curtmola, O. Khan, R. Burns, and G. Ateniese. MR-PDP: Multiple-Replica Provable Data Possession. In Proc. of IEEE ICDCS, 2008.