International Journal of Software Engineering and Knowledge Engineering

World Scientific
www.worldscientific.com

RELIABLE AND SECURE DISTRIBUTED CLOUD DATA STORAGE USING REED-SOLOMON CODES

HAIPING XU[*] and DEEPTI BHALERAO[†]

Computer and Information Science Department, University of Massachusetts Dartmouth
North Dartmouth, MA 02747, USA
[*]*hxu@umassd.edu*
[†]*dbhalerao@umassd.edu*

Received (28 August 2015)
Revised (18 October 2015)
Accepted (Day Month Year)

Despite the popularity and many advantages of using cloud data storage, there are still major concerns about the data stored in the cloud, such as security, reliability and confidentiality. In this paper, we propose a reliable and secure distributed cloud data storage schema using Reed-Solomon codes. Different from existing approaches to achieving data reliability with redundancy at the server side, our proposed mechanism relies on multiple cloud service providers (CSP), and protects users' cloud data from the client side. In our approach, we view multiple cloud-based storage services as virtual independent disks for storing redundant data encoded with erasure codes. Since each CSP has no access to a user's complete data, the data stored in the cloud would not be easily compromised. Furthermore, the failure or disconnection of a CSP will not result in the loss of a user's data as the missing data pieces can be readily recovered. To demonstrate the feasibility of our approach, we developed a prototype distributed cloud data storage application using three major CSPs. The experimental results show that, besides the reliability and security related benefits of our approach, the application outperforms each individual CSP for uploading and downloading files.

Keywords: Distributed cloud data storage; software reliability; data security; erasure codes; cloud service provider (CSP); integer linear programming.

1. Introduction

In the past decades, many businesses have provided their online services to users in a variety of convenient ways such as search engines, webmail, social networks, online shopping, online backup, and online information storage [1]. Online services with large groups of users inevitably generate tremendous amounts of personal and professional digital data, and thus, they require efficient and cost-effective mechanisms to store them. As an ever-growing data storage solution, cloud-based storage services have become a highly practical way for both people and businesses to store their data online [2]. The pay-as-per-use model of cloud computing eliminates the upfront commitment from cloud customers; thereby it allows the customers to start small businesses quickly, and increase resources only when they are needed. Although cloud computing and its underlying

[*] Corresponding author: Dr. Haiping Xu, Associate Professor, Computer and Information Science Department, University of Massachusetts Dartmouth, Email: hxu@umassd.edu.

virtualization technique bring customers many advantages such as elasticity, scalability, flexibility, zero maintenance overhead and reduced costs, there are still major concerns about the data stored in the cloud, e.g., reliability, security and confidentiality [3]. Reliability has been one of the most important concerns for cloud data storage since users typically expect services to be available whenever they need them. This requirement pushes cloud service providers (CSP) to deliver reliable cloud services, which can perform as expected, handle failures without downtime, and recover from failures without affecting the large set of customers. However, cloud downtime statistics studies show an average 7.5 hours of unavailability per year, which is 99.9% availability [4]. This is quite far away from the expected availability for critical businesses, which is 99.99% availability (i.e., 1 hour of unavailability per year). Major CSPs have addressed the issue of availability using data redundancy distributed over multiple physical machines at their server sites; however, recent cloud outage examples show that the added redundancy is not sufficient in case of a complete cloud service failure. In May 2014, Adobe's ID service went down, leaving Creative Cloud users locked out of their software and account for over 24 hours [5]. In early 2013, Dropbox service had a major cloud outage that kept users offline and unable to synchronize using their desktop apps for more than 15 hours [6]. In the same year, Amazon EC2 suffered from an outage for about an hour causing many dependent businesses such as Vine and Instagram, to fall down [7]. Such incidents have made data-critical business owners apprehensive of completely relying on cloud data storage, and also made cloud users question the reliability of cloud storage services provided by even the world-leading CSPs.

In addition to the reliability of cloud storage services, there are also many known security breaches of cloud data in recent years [8]. For example, Amazon's simple storage service has been compromised twice in 2009, which brought many dependent network sites become unavailable [9]. In a recent security breach in Amazon cloud, hackers broke into LinkedIn user accounts and created fake profiles [10]. In spite of several security measures, hackers managed to copy information from thousands of LinkedIn users. From the above examples, we can see that even the largest and the most reputable CSPs have been affected by security and data breaches. As the cloud computing environment becomes more and more open and ubiquitous, it becomes very difficult for a CSP to apply traditional security measures, designed for closed systems, to the open and multi-shared computing environment. On the other hand, since data storage locations and security measures at the server sites are unknown, most of the users have not yet started to feel comfortable with exploiting the full potential of the cloud.

Prolonged cloud data service outages and security concerns can be fatal for businesses with data critical domains such as healthcare, banking and finance. Today, almost all major CSPs have implemented fault-tolerance and security mechanisms at their server sides to recover original data from service failure or data corruption, and to prevent data from being compromised by hackers. Such mechanisms are suitable for a small number of hard disk failures as well as attacks from external hackers; however, they are of no use for the end users to ensure the reliability and security of their cloud data when major

cloud services fail or the cloud services have been compromised by internal hackers, such as employees of the CSP companies. Hence, to achieve high reliability and security of critical data, users should not depend upon a single CSP.

In this paper, we propose an approach that can provide security and fault tolerance to the user's data from the client side. In our approach, we decompose an original data file into multiple data pieces, and generate checksum pieces using erasure codes [11]. The pieces of data are spread over multiple cloud storage services, which can be retrieved and combined to recover the original file. We achieve data redundancy in our approach using erasure codes at the software level across multiple CSPs. Therefore, the original data can be recovered even when there is a cloud outage where some cloud service fails completely. Using this approach, users' data would not be easily compromised by unauthorized access and security breach, as no single cloud service has the complete knowledge of the users' data. Users could have the sole control of their cloud data, and do not need to completely rely on the security measures provided by the CSPs. In addition, due to the ever changing and growing set of users and data centers, the performance of cloud services is also a major concern. With more and more users share the same cloud storage infrastructure provided by a CSP, the cloud performance issue is getting more and more serious than ever before. To improve the network performance of our approach, we adopt the multithreading technology, and fully utilize the network bandwidth in order to minimize the time required to access data over the cloud.

This work is based on our previously proposed reliable and secure cloud storage schema using multiple CSPs [12]. In this paper, we provide additional details about how to use Reed-Solomon codes [13] to achieve high reliability and security of cloud data. We also perform further experiments to demonstrate the consistently high performance of our approach.

The rest of the paper is organized as follows. Section 2 discusses previous work related to our research. Section 3 presents a framework for reliable and secure distributed cloud data storage. Section 4 describes how to use erasure codes, in particular, the Reed-Solomon codes to implement such a framework. Section 5 discusses how to calculate the optimal number of checksum pieces. Section 6 presents a case study to demonstrate the feasibility and high performance of our approach. Section 7 concludes the paper and mentions future work.

2. Related Work

There have been many research efforts on using erasure codes at the server side to make cloud storage service reliable. Huang *et al.* proposed to use erasure codes in Windows Azure storage [14]. They introduced a new set of codes for erasure coding called Local Reconstruction Codes (LRC) that could reduce the number of erasure coding fragments required for data reconstruction. Their approach divides redundant data into both local and global sets of parities, and stores them in geographically separated servers. Since local parities minimize I/O and network overhead during data recovery, the overall reconstruction cost can be significantly reduced. Gomez *et al.* introduced a novel

persistency technique that leverages erasure codes to save data in a reliable fashion on IaaS clouds [15]. They presented a scalable erasure coding algorithm that could support a high degree of reliability for local storage with the cost of low computational overhead and a minimal amount of communication. Experimental results show that their approach may improve the overall performance of real-life High Performance Computing (HPC) applications. Khan *et al.* provided some guidance for applying erasure coding techniques in cloud file systems to support load balance and incremental scalability in data centers [16]. Their proposed approach can prevent correlated failures with data loss and mitigate the effect of any single failure on a data set or an application. Although the above approaches can significantly enhance the reliability of cloud data at data centers, they provide no support for end users to deal with cloud outage of the service providers. Different from the existing approaches, we apply erasure-coding techniques at the application level using multiple CSPs. By deploying a user's encoded redundant data across multiple cloud storage services, our approach is fault tolerant for storing cloud data when any of the cloud services fails.

There is also a considerable amount of work on securing cloud data, to which this work is closely related. Santos *et al.* proposed a secure and trusted cloud computing platform (TCCP) for Infrastructure as a Service (IaaS) providers such as Amazon EC2 [17]. The platform provides a closed box execution environment that guarantees confidential execution of guest virtual machines on a cloud infrastructure. Hwang and Li proposed to use data coloring and software watermarking techniques to protect shared cloud data objects [18]. Their approach can effectively prevent data objects from being damaged, stolen, altered or deleted, and users may have their sole access to their desired cloud data. Wang *et al.* described an effective and flexible distributed schema that integrates storage correctness insurance and identification of misbehaving servers [19]. Using erasure coding techniques, the schema provides security to cloud data storage servers against Byzantine failures as well as malicious data modification attacks. The existing approaches to securing cloud data typically assume that the CSPs are trustable and they can prevent physical attacks to their servers. However, this might not be true in reality because service providers typically tend to collect users' cloud data for their commercial purposes such as targeted adverting. Furthermore, there have been many incidents that cloud services were compromised by either internal or external hackers, and thousands of users' critical data was compromised. Therefore, merely relying on service providers' security mechanisms is not a feasible solution for both people and businesses to store their critical data in the cloud. It is required that users should be allowed to apply security mechanisms to their own data at the client side. Different from the aforementioned methods to securing users' data in the cloud, our approach does not rely on any security measures at the server side. Instead, the cloud data storage application running at the client side can split users' data into pieces, encode them using erasure codes, and distribute them to multiple CSPs. Assume there are no collusions among the CSPs, users can securely store their data in the cloud.

Additional work on cloud storage emphasizes on improving the cloud storage performance. Shue *et al.* presented a cloud-based system for balancing workloads in multi-tenant systems to achieve fairness in shared storage systems [20]. They distributed workload uniformly across virtual machines in order to achieve high utilization and improve the system performance at the server side. Zia and Khan identified a number of key challenges in performance issues in cloud computing [21]. They summarized potential performance improvement in different areas such as storage services, scaling, network services, scheduling, optimal location of data centers, and efficient SQL query processing. Different from the above methods, our approach focuses on network service performance and adopts the multithreading technique for uploading and downloading pieces of data with multiple CSPs. Experimental results show that with reasonable number of data pieces, the system performance can be significantly improved.

In this paper, we extend the methodology and results of a preliminary study on secure and fault-tolerant model of cloud information storage [3]. In the previous work, we followed the RAID (Redundant Array of Independent Disks) approach to encode users' data using `XOR` parity, and developed a hierarchical colored Petri nets (HCPN) model for secure and fault-tolerant cloud information storage systems. In this paper, we adopted more advanced erasure coding techniques to achieve security and fault tolerance for cloud data storage, and presented a detailed design for a reliable and secure distributed cloud data storage schema. To demonstrate the effectiveness of our proposed approach, we implemented a prototype using three major CSPs, which allows users to securely, reliably and efficiently store their critical data in the cloud.

3. A Framework for Reliable and Secure Distributed Cloud Data Storage

To address the aforementioned major concerns in cloud storage services, we propose a reliable and secure distributed cloud data storage schema using multiple CSPs. Figure 1 shows a framework for such a distributed storage system. The major component of the system is the cloud data storage application that uses erasure codes to encode and decode file pieces at the client side, and upload and download encoded file pieces concurrently at multiple CSPs. Note that each CSP typically stores more than one file piece; therefore, concurrent file transfers happen at two different levels, namely among multiple CSPs and within a single CSP. As shown in the figure, when a user wants to upload a file into the cloud, the application first splits the file into multiple data pieces, say *n* pieces, and then encode them into optimal number of *m* checksum pieces using the erasure coding technique. Once the data pieces and checksum pieces are ready, they are concurrently uploaded into multiple cloud storages maintained by different CSPs, noted as *CSP*_1, *CSP*_2, …, and *CSP_N* in Fig. 1. As none of the CSPs has the complete knowledge about the user's data, this approach can effectively defend against data breaches from any single CSP.

On the other hand, when a user wants to download a stored file, the application will first try to download the *n* data pieces from the multiple cloud services concurrently. If all data pieces are available, they can be efficiently combined into the original file without

any additional decoding related cost. However, in the case when one or more service providers fail, the application will automatically download all available data pieces (n') and available checksum pieces (m'). As long as $n' + m' \geq n$, due to the erasure coding technique, the application can always successfully decode the missing data pieces using the available pieces of data, and restore the original file. Note that the checksum pieces serve as the redundant information of the original file, which makes our approach reliable and fault tolerant.

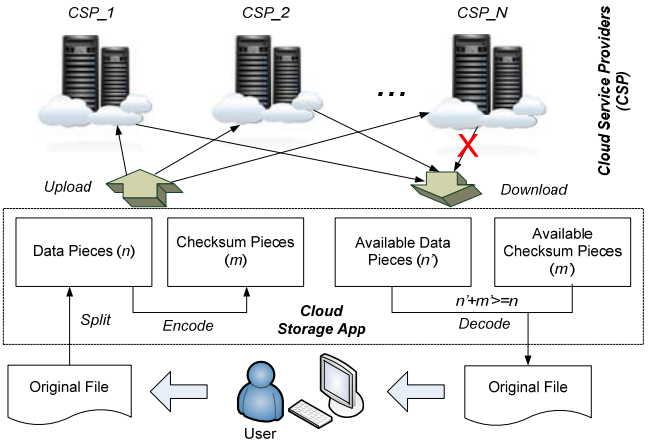

Fig. 1. A framework for reliable and secure distributed cloud data storage systems

4. Erasure Codes and Reed-Solomon Coding

4.1. Erasure Codes

In early days, fault tolerance of cloud data is commonly achieved through simple data replication. Multiple copies of original data have to be maintained on different cloud servers in order to make data more reliable. However, data replication now becomes highly unfeasible due to its low space efficiency and the ever-increasing amount of cloud data. Erasure codes, also known as forward error correction (FEC) codes, manage to overcome the disadvantages of the data replication approach, and can achieve a high degree of fault tolerance with a much lower cost of physical storage [11]. Erasure codes use mathematical functions to convert original data words into encoded code words, and to decode the code words in order to recover the data words when some of them are lost. They can be very efficient in providing fault tolerance for large quantities of data, hence they are very appropriate for large-scale cloud data storage systems.

Data redundancy through parity codes represents the simplest form of erasure codes, which overcomes the drawback of data replication. RAID-5 is the most commonly used technique that uses parity codes. It calculates parities from the original data to achieve fault tolerance. However, this technique is typically used by CSPs at the hardware level,

and very few research efforts attempted to apply the RAID concept at the software level to resolve issues related to the major data failures of a service provider, which happen quite often nowadays [3].

4.2. Reed-Solomon Coding for Cloud Data Storage

Use of error-correction codes for redundancy has become prevalent due to its various advantages. Reed-Solomon (RS) coding is a type of optimal erasure codes, which follows the basic error-correction techniques [22]. There are many different ways to implement error-correction using erasure codes, such as parity check, polynomial oversampling, Tornado codes and RS codes, but RS technique is a good compromise between efficiency and complexity [23]. Traditionally, RS technique has been used in various applications such as error-correction in CD-ROM and DVDs, satellite communications, digital television, and wireless or mobile communications [13]. The use of RS technique to provide fault tolerance over the cloud is a fairly new idea. Our approach to distributing data and checksum pieces with multiple cloud data services could build a RAID-like system with less storage overhead and more flexibility in the degree of fault tolerance for the stored data. Different from the RAID-5 based approach in previous work [3], the RS-based approach allows multiple failures of cloud services. As a brief introduction to the RS algorithm, let there be n data pieces. We encode all data pieces using the RS algorithm into m checksum pieces such that out of $(n+m)$ pieces, any n pieces are enough to recover the original n data pieces. If the $(n+m)$ pieces of data are distributed over $(n+m)$ cloud services, this algorithm can be used to handle m failures of the services.

To simplify matters, we assume each data piece is an unsigned byte ranged from 0 to 255. In order to calculate the checksum bytes, we first create an $(m+n)\times n$ Vandermonde matrix $\mathbf{A}$ as in Eq. (1), where the i, j-th element of $\mathbf{A}$ is defined to be i^j [22]. Based on this definition, when m rows are deleted from $\mathbf{A}$, the newly formed matrix is invertible. Then we derive the information dispersal matrix $\mathbf{B}$ from $\mathbf{A}$ using a finite sequence of row elementary operations. The information dispersal matrix $\mathbf{B}$ is defined as in Eq. (2), where $\mathbf{I}$ is an $n\times n$ identity matrix, and $\mathbf{F}$ is an $m\times n$ matrix. Note that elementary matrix transformation does not change the rank of a matrix and each row in $\mathbf{A}$ is linear independent, thus the information dispersal matrix $\mathbf{B}$ maintains this property from matrix $\mathbf{A}$ such that when m rows are deleted from $\mathbf{B}$, the newly formed matrix is invertible.

$$\mathbf{A}=\begin{bmatrix} 1 & \cdots & 0^{n-1} \\ \vdots & \ddots & \vdots \\ 1 & \cdots & (n-1)^{n-1} \\ \hline 1 & \cdots & n^{n-1} \\ \vdots & \ddots & \vdots \\ 1 & \cdots & (m+n-1)^{n-1} \end{bmatrix} \quad (1) \qquad \mathbf{B}=\begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \\ \hline f_{0,0} & \cdots & f_{0,n-1} \\ \vdots & \ddots & \vdots \\ f_{m-1,0} & \cdots & f_{m-1,n-1} \end{bmatrix}=\begin{bmatrix} \mathbf{I} \\ \hline \mathbf{F} \end{bmatrix} \quad (2)$$

Let **D** be a vector of n-byte data $[d_0, d_1, ..., d_{n-1}]$, and **C** be a vector of m-byte checksum $[c_0, c_1, ..., c_{m-1}]$. With the information dispersal matrix **B**, we can calculate the checksum vector **C** from the data vector **D** as in Eqs. (3.1) and (3.2), where $f_{i,j}$, for $0 \le i \le m$-1 and $0 \le j \le n$-1, are elements of the $m \times n$ matrix **F**.

$$\mathbf{BD} = \begin{bmatrix} \mathbf{I} \\ -- \\ \mathbf{F} \end{bmatrix} \mathbf{D} = \begin{bmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \\ \hline f_{0,0} & \cdots & f_{0,n-1} \\ \vdots & \ddots & \vdots \\ f_{m-1,0} & \cdots & f_{m-1,n-1} \end{bmatrix} \begin{bmatrix} d_0 \\ \vdots \\ d_{n-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ \vdots \\ d_{n-1} \\ \hline c_0 \\ \vdots \\ c_{m-1} \end{bmatrix} = \begin{bmatrix} \mathbf{D} \\ -- \\ \mathbf{C} \end{bmatrix} = \mathbf{E} \tag{3.1}$$

$$\left.\begin{aligned} c_0 &= f_{0,0} * d_0 + f_{0,1} * d_1 + ... + f_{0,n-1} * d_{n-1} \\ c_1 &= f_{1,0} * d_0 + f_{1,1} * d_1 + ... + f_{1,n-1} * d_{n-1} \\ &... \\ c_{m-1} &= f_{m-1,0} * d_0 + f_{m-1,1} * d_1 + ... + f_{m-1,n-1} * d_{n-1} \end{aligned}\right\} \tag{3.2}$$

Now suppose k bytes, where $k \le m$, are missing from vector **D**. By deleting the missing k elements from **D** as well as any m-k elements from **C**, we derive a new n-byte vector **E'** as in Eq. (4), where **D'** is a (n-k)-byte vector $[d_0', d_1', ..., d_{n-k-1}']$, and **C'** is a k-byte vector $[c_0', c_1', ..., c_{k-1}']$.

$$\mathbf{E}' = \begin{bmatrix} \mathbf{D}' \\ -- \\ \mathbf{C}' \end{bmatrix} \tag{4}$$

$$\mathbf{B}' = \begin{bmatrix} \mathbf{I}' \\ -- \\ \mathbf{F}' \end{bmatrix} \tag{5}$$

$$\mathbf{B}'\mathbf{D} = \begin{bmatrix} \mathbf{I}' \\ -- \\ \mathbf{F}' \end{bmatrix} \mathbf{D} = \begin{bmatrix} \mathbf{D}' \\ -- \\ \mathbf{C}' \end{bmatrix} = \mathbf{E}' \tag{6}$$

Similarly, in Eq. (2), by deleting m rows from **B** that correspond to the deleted rows in **E**, we drive an $n \times n$ matrix **B'** as in Eq. (5), where **I'** is an (n-k)$\times n$ matrix, and **F'** is a $k \times n$ matrix. After the row deletion, Eq. (3.1) becomes Eq. (6). Since matrix **B'** is invertible, we can calculate the inverse matrix $\mathbf{G} = \mathbf{B}'^{-1}$ using Gaussian elimination method, and recover the data vector **D** as in Eqs. (7.1) and (7.2), where $g_{i,j}$, for $0 \le i \le n$-1 and $0 \le j \le n$-1, are elements of the $n \times n$ matrix **G**.

$$\mathbf{D} = \mathbf{B}'^{-1}\mathbf{E}' = \mathbf{G} \begin{bmatrix} \mathbf{D}' \\ -- \\ \mathbf{C}' \end{bmatrix} = \begin{bmatrix} g_{0,0} & \cdots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{n-1,0} & \cdots & g_{n-1,n-1} \end{bmatrix} \begin{bmatrix} d_0' \\ \vdots \\ d_{n-k-1}' \\ --- \\ c_0' \\ \vdots \\ c_{k-1}' \end{bmatrix} \tag{7.1}$$

$$\left.\begin{aligned}
d_0 &= g_{0,0} * d_0^{'} + g_{0,1} * d_2^{'} + \ldots + g_{0,n-k-1} * d_{n-k-1}^{'} + \\
&\quad g_{0,n-k} * c_0^{'} + g_{0,n-k+1} * c_1^{'} + \ldots + g_{0,n-1} * c_{k-1}^{'} \\
d_1 &= g_{1,0} * d_0^{'} + g_{1,1} * d_2^{'} + \ldots + g_{1,n-k-1} * d_{n-k-1}^{'} + \\
&\quad g_{1,n-k} * c_0^{'} + g_{1,n-k+1} * c_1^{'} + \ldots + g_{1,n-1} * c_{k-1}^{'} \\
&\ldots \\
d_n &= g_{n-1,0} * d_0^{'} + g_{n-1,1} * d_2^{'} + \ldots + g_{n-1,n-k-1} * d_{n-k-1}^{'} + \\
&\quad g_{n-1,n-k} * c_0^{'} + g_{n-1,n-k+1} * c_1^{'} + \ldots + g_{n-1,n-1} * c_{k-1}^{'}
\end{aligned}\right\} \quad (7.2)$$

Once the n-byte vector **D** is restored, the m-byte vector **C** can be recalculated using vector **D** and the information dispersal matrix **B** as in Eq. (3.2).

4.3. Computation over Galois Field Using Signed Bytes

Implementation of the RS algorithm for data files requires performing computations on binary words of a fixed length w. For example, when the binary word is a byte, w equals 8. To ensure that the RS algorithm works correctly for fixed-size words, all arithmetic operations must be performed over a Galois Field with 2^w elements, denoted as $GF(2^w)$ [22]. A Galois Field $GF(2^w)$ is also known as a finite field, which contains finitely many elements, namely 0, 1, ..., 2^w-1. Arithmetic operations performed over Galois Fields result in finite values in $GF(2^w)$. Addition and subtraction of elements over $GF(2^w)$ are simply `XOR` operations, but multiplication and division must use two tables called `gflog` and `gfilog` for their computations, where the `gflog` and `gfilog` tables map an index to its logarithm and inverse logarithm in $GF(2^w)$, respectively. Table 1 shows a partial logarithm and inverse logarithm table for $GF(2^w)$, where $w = 8$.

Table 1. Partial logarithm and inverse logarithm table for GF(2^w), where $w = 8$

i	0	1	2	3	4	5	6	7	...	12	...	155	...	205	...	217	...	254	255
gflog[i]	0	0	1	25	2	50	26	198	...	27	...	217	...	12	...	96	...	88	175
gfilog[i]	1	2	4	8	16	32	64	128	...	205	...	114	...	167	...	155	...	142	0

In this paper, we use a byte as a word; therefore, w equals 8, and the elements in $GF(2^w)$ are 0, 1, 2, ..., 255. This allows us to perform arithmetic on single bytes. When a program language supports unsigned byte directly, such as language C, the algorithms for multiplication and division over $GF(2^8)$ could be straightforward [24]. However, for a programming language that does not support unsigned byte, such as Java, the range of bytes is -127 to 128 rather than 0 to 255. To avoid calculation with negative values, it is important to convert signed bytes into nonnegative integers in $GF(2^8)$, which must be within the range of [0, 255]. The following two methods `gf_multiply` and `gf_divide`, which support multiplication and division operations over $GF(2^8)$ using signed bytes, respectively, are adapted from reference [24].

```
byte gf_multiply(byte a, byte b) {
  int int_a, int_b, int_sum;
  if (a == 0 || b == 0) return 0;
  int_a = (int) (a & 0xFF);
  int_b = (int) (b & 0xFF);
  int_sum = (int)(gflog[int_a] & 0xFF) +
            (int)(gflog[int_b] & 0xFF);
  if (int_sum >= 255) int_sum -= 255;
  return gfilog[int_sum];
}

byte gf_divide(byte a, byte b) {
  int int_a, int_b, int_diff;
  if (a == 0) return 0;
  if (b == 0) return -1;
  int_a = (int) (a & 0xFF);
  int_b = (int) (b & 0xFF);
  int_diff = (int)(gflog[int_a] & 0xFF) -
             (int)(gflog[int_b] & 0xFF);
  if (int_diff < 0) int_diff += 255;
  return gfilog[int_diff];
}
```

In the above two methods, the two byte arrays `gflog[0..255]` and `gfilog[0..255]` define the tables that map an index in [0, 255] to its logarithm and inverse logarithm for $GF(2^8)$, respectively. By applying the bit operation "`& 0xFF`", it masks a signed byte into a nonnegative integer in [0, 255]. Note that all arithmetic operations mentioned in Section 4.2, including the matrix inverse, encoding and recovery of data, must be calculated using Galois Field arithmetic operations. To demonstrate how multiplication and division can be done over $GF(2^8)$ using signed bytes, we provide two examples as follows.

```
Before conversion: a = (byte)0b10011011, b = (byte)0b00000101
After conversion:  int_a = 155, int_b = 5
gf_multiply(a, b) = gfilog[gflog[155] + gflog[5]]
                  = gfilog[217 + 50 - 255]  // (217 + 50) > 255
                  = gfilog[12]
                  = 205
Before conversion: a = (byte)0b11001101, b = (byte)0b00000101
After conversion: int_a = 205, int_b = 5
gf_divide(a, b) = gfilog[gflog[205] - gflog[5]]
                  = gfilog[12 - 50 + 255]   // (12 - 50) < 0
                  = gfilog[217]
                  = 155
```

Note that in the above examples, the results would not be correct if the signed bytes were not properly converted into nonnegative integers within [0, 255]. In addition, it is easy to see that the regular operations of multiplication and division are not suitable for

the calculations on single bytes, as in the above examples, the value of (155 * 5) goes out of the range of an unsigned byte.

5. Optimal Number of Checksum Pieces

5.1. Calculating the Optimal Number of Checksum Pieces

In order to achieve the highest space efficiency in our approach, we propose a procedure to compute the minimal number of checksum pieces that allow the failures of multiple CSPs. Let N be the total number of CSPs, where $N \geq 2$, $\Gamma = \{1, 2, ..., N\}$ be the set of CSPs, and M be the maximal number of services allowed to fail or become unavailable at the same time, where $1 \leq M \leq N\text{-}1$. We define a failure set Φ as follows:

$\Phi \in P(\Gamma)$, where $P(\Gamma)$ is the power set of Γ, and $|\Phi| \leq M$.

The set of available CSPs Ω due to the failure set Φ can be defined as in Eq. (8).

$$\Omega = \Gamma - \Phi \tag{8}$$

Suppose we divide a user's file into n data pieces, where $n \geq 2$. To distribute n data pieces evenly over N CSPs, where $2 \leq N \leq (n+1)$, we calculate the number of data pieces $n_1, n_2, ...,$ and n_N stored at $CSP1, CSP2, ...,$ and CSP_N, respectively, as in Eq. (9).

$$n_i = \begin{cases} \lceil n/N \rceil & \text{when} \quad i = 1 \\ \left\lceil (n - \sum_{j=1}^{i-1} n_j)/(N-i+1) \right\rceil & \text{when} \quad 1 < i < N \\ n - \sum_{j=1}^{N-1} n_j & \text{when} \quad i = N \end{cases} \tag{9}$$

where $n = n_1 + n_2 + ... + n_N$. Eq. (9) allows even distribution of n data pieces over N CSPs such that $|n_i - n_j| \leq 1$ for $1 \leq i, j \leq N$. For example, when $N = 3$ and $n = 7$, the numbers of data pieces distributed over the three cloud service providers $CSP1$, $CSP2$ and $CSP3$ will be 3, 2, 2, respectively.

As a major requirement for fault tolerance, when up to M CSPs become unavailable, the original data must be recovered from the remaining CSPs in the available set Ω. Let m be the number of checksum pieces required, and $m_1, m_2, ..., m_N$ are the numbers of checksum pieces distributed over $CSP1, CSP2, ...,$ and CSP_N, respectively. Obviously, we have $m = m_1 + m_2 + ... + m_N$. To calculate the minimal number of checksum pieces m, we can solve the integer linear programming (ILP) problem as in Eq. (10).

$$\begin{aligned} &\text{minimize} \quad \sum_{i=1}^{N} m_i \\ &\text{subject to} \quad \textit{for} \text{ each failure set } \Phi \\ &\qquad \sum_{i \in \Omega} m_i \geq \sum_{j \in \Phi} n_j \\ &\qquad \text{where } \Phi \in P(\Gamma) \text{ and } |\Phi| = M \end{aligned} \tag{10}$$

Note that a solution to the above optimal problem automatically satisfies the fault-tolerance requirement when $|\Phi| < M$. The space efficiency e of a solution can be calculated as in Eq. (11).

$$e = 1 - m/(n+m), \text{ where } n = \sum_{i=1}^{N} n_i \text{ and } m = \sum_{i=1}^{N} m_i \tag{11}$$

As an example, when $N = 3$ and $M = 1$, the ILP problem defined in Eq. (10) can be simplified as the one in Eq. (12).

$$\begin{aligned} \text{minimize} \quad & m_1 + m_2 + m_3 \\ \text{subject to} \quad & m_1 + m_2 \geq n_3 \qquad // \Phi = \{3\} \\ & m_2 + m_3 \geq n_1 \qquad // \Phi = \{1\} \\ & m_1 + m_3 \geq n_2 \qquad // \Phi = \{2\} \end{aligned} \tag{12}$$

Table 2 shows the optimal solutions and their space efficiency for the case when $N = 3$, $M = 1$, and n from 2 to 15. For example, when $n = 8$ ($n_1 = 3$, $n_2 = 3$, $n_3 = 2$), the optimal solution is ($m_1 = 1$, $m_2 = 1$, $m_3 = 2$), and the space efficiency $e = 1 - 4/(8+4) = 0.6667$. For this example, if any service provider becomes unavailable, the missing 4 pieces of data can always be recovered from the remaining data pieces and the checksum pieces stored with the other two service providers.

Table 2. Optimal number of checksum pieces and space efficiency

Data Pieces (n)	(n_1, n_2, n_3)	(m_1, m_2, m_3)	Checksum Pieces(m)	Space Efficiency(e)
2	(1, 1, 0)	(0, 0, 1)	1	0.6667
3	(1, 1, 1)	(1, 1, 0)	2	0.6000
4	(2, 1, 1)	(0, 1, 1)	2	0.6667
5	(2, 2, 1)	(1, 1, 1)	3	0.6250
6	(2, 2, 2)	(1, 1, 1)	3	0.6667
7	(3, 2, 2)	(1, 2, 1)	4	0.6364
8	(3, 3, 2)	(1, 1, 2)	4	0.6667
9	(3, 3, 3)	(2, 2, 1)	5	0.6429
10	(4, 3, 3)	(1, 2, 2)	5	0.6667
11	(4, 4, 3)	(2, 2, 2)	6	0.6471
12	(4, 4, 4)	(2, 2, 2)	6	0.6667
13	(5, 4, 4)	(2, 3, 2)	7	0.6500
14	(5, 5, 4)	(2, 2, 3)	7	0.6667
15	(5, 5, 5)	(3, 3, 2)	8	0.6522

5.2. Distribution of Data and Checksum Pieces over multiple CSPs

When dealing with a file with k bytes, if k is not a multiple of n, we need to append r bytes with random values to the end of the file such that $((k + r) \bmod n) = 0$. Then we split the $(k+r)$ bytes into n data pieces so that each of them contains exactly $(k+r)/n$ bytes. By applying Eq. (9) and solving the ILP problem in Eq. (10), we can calculate the distribution of the n data pieces over multiple CSPs and the optimal number of checksum pieces, respectively. Finally, using the equations defined in Eq. (3.2), we can calculate the checksum pieces. Fig. 2 shows an example of file distribution at service providers *CSP*1, *CSP*2 and *CSP*3 when $N = 3$, $M = 1$, $n = 8$ and $m = 4$.

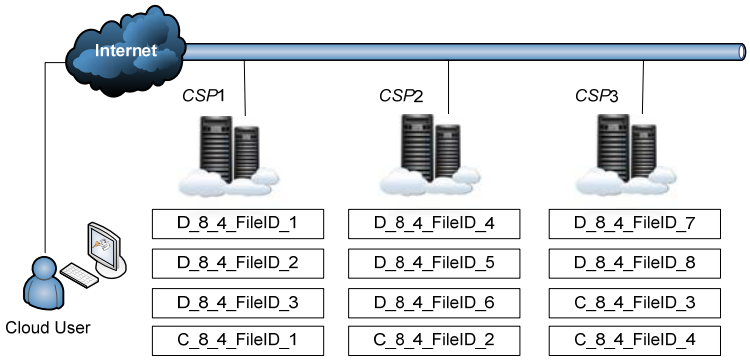

Fig. 2. Distribution of data and checksum pieces at three CSPs

As shown in Fig. 2, we distribute 3, 3 and 2 data pieces (denoted by the file names starting with the letter "D") over *CSP*1, *CSP*2 and *CSP*3, respectively. Based on the optimal solution given in Table 2, we also distribute 1, 1 and 2 checksum pieces (denoted by the file names starting with the letter "C") over *CSP*1, *CSP*2 and *CSP*3, respectively. When any of the service providers fails, the original data can be recovered from the remaining 8 pieces of data using the equations defined in Eq. (7.2). It is worth noting that by the definition of the RS coding algorithm, when the 4 missing or corrupted data pieces are from more than one CSPs, the original file can still be recovered using the same equations defined in Eq. (7.2).

6. Case Study

To demonstrate the feasibility as well as the high performance of our RS-based approach, we developed a prototype secure and reliable distributed cloud data storage application in Java. We adopt three different cloud storage services supported by major CSPs to store our data pieces and checksum pieces in the cloud. The selected cloud storage services are Amazon S3, Google App Engine, and Core Dropbox APIs with free user accounts. The application was running on a Windows machine with a 3.40 GHz Intel Core i7 processor and 8.00 GB of RAM. All experiments have been conducted with excellent Internet

connections at University of Massachusetts Dartmouth, where the download speed was around 160 Mbps (~20MB/s) and the upload speed was around 400 Mbps (~50MB/s). Therefore, the network connection at the client side would not become a bottleneck for all of our experiments. For each experiment in our case study, we repeat it at least 3 times and choose the median in an attempt to estimate the typical upload or download time for each particular setting. Table 3 shows the upload time and download time for two files with different file sizes using a single CSP. From the table, we can see that among the three CSPs, Dropbox has the best performance for file uploading; while Google App Engine has the worst performance. On the other hand, Google App Engine and Dropbox have the almost equivalently best performance for file downloading; while Amazon S3 has the worst performance. As our approach requires using the three CSPs concurrently, the overall performance of the application would be restricted by the ones with the worst performance. In other words, the upload speed and download speed of the application may be held back by Google App Engine and Amazon S3, respectively.

Table 3. File uploading and downloading time using a single CSP

File Size	CSP	Upload Time	Download Time
156 MB	Amazon S3	2 min 52 sec	1 min 9 sec
	Google App Engine	5 min 39 sec	33 sec
	Dropbox	2 min 19 sec	39 sec
317 MB	Amazon S3	5 min 8 sec	2 min 17 sec
	Google App Engine	11 min 36 sec	1 min 3 sec
	Dropbox	4 min 13 sec	1 min 1 sec

Figure 3 shows the user interface of the application that allows one to select a file and upload it into the cloud. After choosing the number of data pieces (n), the optimal number of checksum pieces (m) can be automatically calculated by solving the corresponding ILP problem. When clicking on the "Encode and Upload" button, the selected file is divided into n data pieces, which are then automatically encoded into m checksum pieces by the application. Once all pieces of data become ready, they are uploaded into the selected cloud storages using multithreading techniques. The message box in the user interface displays the encoding time, the uploading time and the total processing time.

Figure 4 shows the user interface for downloading and decoding an uploaded file in the cloud. As shown in the figure, a user first selects a file from the list of uploaded files, and then chooses at least two CSPs because the maximal number of failed cloud services M equals 1. Note that in Fig. 4, the Dropbox service is not selected as we assume it is not available at this moment. When clicking on the "Download and Decode" button, the available file pieces are concurrently downloaded to the local computer, where the original file is recovered using the RS coding techniques. The message box in the user interface displays the downloading time, the decoding time and the total processing time, as well as the location of the downloaded file on the user's local computer.

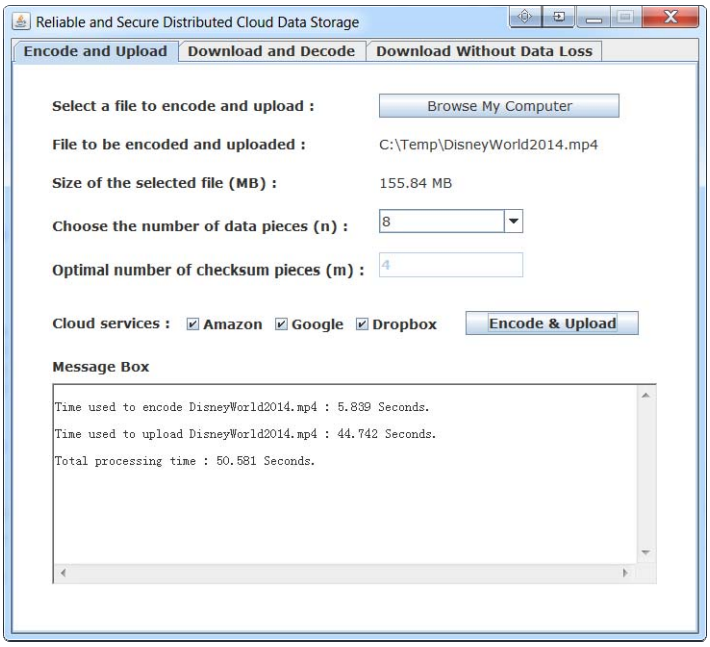

Fig. 3. Encode and upload a file to multiple cloud data storages

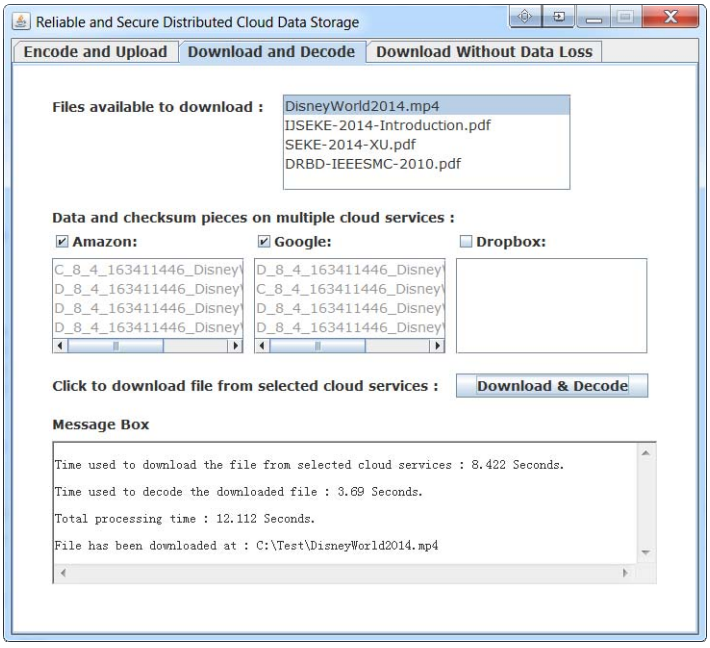

Fig. 4. Download and decode a file from the cloud with a failed service

To analyze the performance of our approach, we selected a video file with a file size of 156 MB. Figure 5 shows the encoding and uploading time *vs.* the number of data pieces set by the user. From the figure, we can see that when we increase the number of data pieces from 2 to 8, the uploading time drops down significantly; while the encoding time has slightly increased. The significant performance improvement for uploading is due to the use of multithreading technique; while the increased number of data pieces along with more checksum pieces results in more overhead for encoding. However, when the number of data pieces n is further increased, the uploading time and the total processing time become relatively stable. This result is quite different from our previous findings, where the uploading time dramatically goes up when the number of data pieces $n \geq 10$ [12]. Based on our further investigation, we notice that the former result is due to a sudden performance drop at Amazon S3 when the file size of a data piece becomes less than 16 MB. Note that when $n = 10$, the size of each data piece equals 156MB/10, i.e., 15.6MB. With the default configuration, the `TransferManager` from the Amazon S3 API uses a single low-speed thread for uploading files of less than 16 MB. In this case study, we modify the configuration to allow multi-part upload of small files. As a result, the sudden performance drop (when $n \geq 10$) disappeared. As shown in Fig. 5, the total processing time becomes constantly less than one minute. Comparing to the best performance for uploading the 156 MB file using a single CSP (as listed in Table 3), where Dropbox used 2 minutes 19 seconds, our approach demonstrates a significant performance improvement for uploading the file.

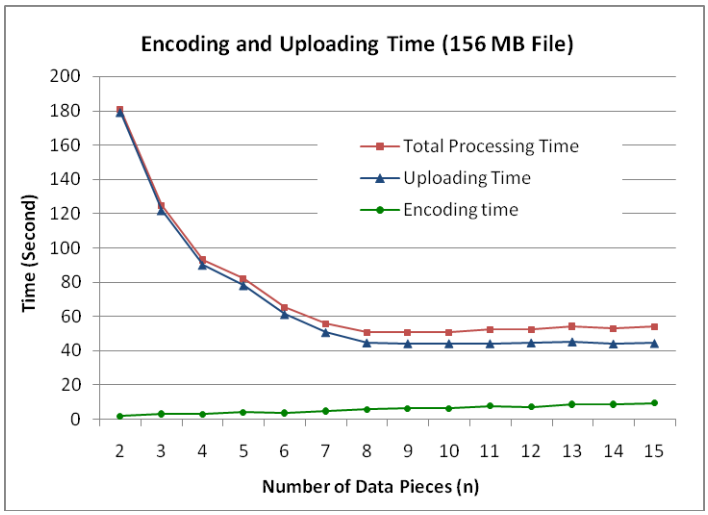

Fig. 5. Encoding & uploading time vs. number of data pieces (156 MB file)

Figure 6 shows the downloading and decoding time *vs.* the number of data pieces set by the user. From the figure, we can see that when we increase the number of data pieces

from 2 to 8, the downloading time drops down significantly; while the decoding time has slightly increased. Similar to the case of uploading, the significant performance improvement for downloading is also due to the use of multithreading technique, and the increased number of data pieces along with more checksum pieces results in more overhead for decoding. When the number of data pieces n is further increased, the total processing time only slightly goes up due to the overhead of decoding files. However, when $n \geq 8$, the total processing time is constantly below 15 seconds. Comparing to the best performance for downloading the 156 MB file using a single CSP (as listed in Table 3), where Google App Engine used 33 seconds, our approach demonstrates a significant performance improvement for downloading the file.

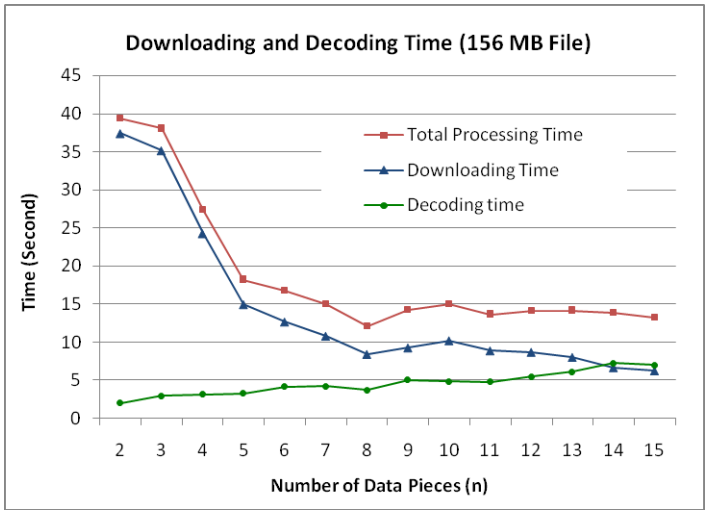

Fig. 6. Downloading and decoding time vs. number of data pieces (156 MB file)

From the above experimental results, we can see that both the uploading and downloading time can be significantly reduced by selecting a reasonable number of data pieces. For example, when the file size is about 156 MB, based on our experiments, the number of data pieces should normally be set to 8 as long as the network condition is excellent, and the client machine has similar performance as the one used in the case study. Note that according to Table 2, when $n = 8$ and $m = 4$, the space efficiency e reaches its highest value 0.6667. It is also worth noting that when no service provider fails, the application only requires downloading the data pieces, and no checksum pieces are needed for restoring the original file. In this case, the downloading time can be further reduced, and the decoding time becomes merely the time needed to combine the data pieces into the original file. Therefore, in a usual case with no failures of service providers, the overall performance for file retrieval could be better than the results demonstrated as in Fig. 6.

To further investigate the system performance for larger files, we use another video file with a file size of 317 MB for this case study. Figure 7 shows the encoding and uploading time *vs.* the number of data pieces set by the user. From the figure, we can see that when we increase the number of data pieces from 2 to 8, the uploading time drops down significantly; while the encoding time has slightly increased. When $n > 8$, the uploading time is further decreased along with the encoding time being slightly increased. As shown in the figure, for $n \geq 8$, the total processing time is constantly below 100 seconds, i.e., 1 minute and 40 seconds. Comparing to the best performance for uploading the 317 MB file using a single CSP (as listed in Table 3), where Dropbox used 4 minutes and 13 seconds to upload the file, our approach again demonstrates a significant performance improvement for uploading the file.

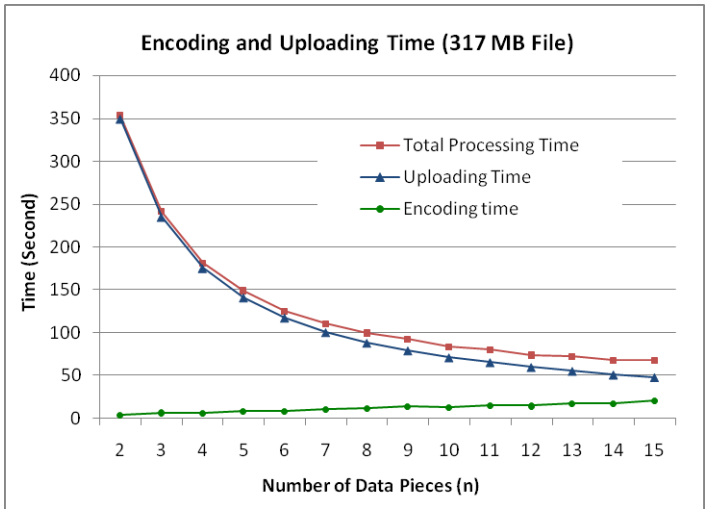

Fig. 7. Encoding & uploading time vs. number of data pieces (317 MB file)

Figure 8 shows the downloading and decoding time *vs.* the number of data pieces set by the user for the 317 MB file. From the figure, we can see that when we increase the number of data pieces from 2 to 8, the downloading time drops down significantly; while the decoding time has slightly increased. When the number of data pieces n is further increased, the total processing time slightly goes up and then goes down. However, for $n \geq 8$, the total processing time is constantly below 45 seconds. Comparing to the best performance for downloading the 317 MB file using a single CSP (as listed in Table 3), where Dropbox used 1 minute and 1 second to download the file, our approach again demonstrates a significant performance improvement for downloading the file.

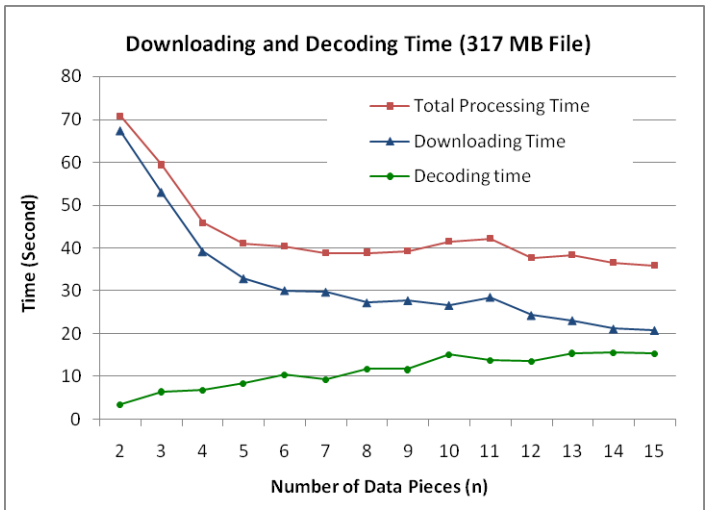

Fig. 8. Downloading and decoding time vs. number of data pieces (317 MB file)

From the above experimental results, we notice that the number of data pieces should also be normally set to 8 for a file of about 317 MB, as long as the network condition is excellent, and the performance of the client machine is good enough. Note that when $n >$ 8, although the uploading and downloading time may be slightly reduced, the overhead of encoding and decoding files can be increased; thus the benefits of having too many data pieces and checksum pieces is not significant. To reduce the overhead of encoding and decoding a large number of file pieces, especially on a low-end PC or a tablet, selecting a reasonably small number of data pieces might be more desirable.

To gain insights for more general cases, we further performed many experiments with different file types and file sizes. As our approach is applied at the byte-level of a file, the file type does not affect the system performance. This is consistent with our experimental results. In addition, with much smaller file sizes (e.g., less than 50 MB), the overhead for encoding and decoding also becomes very small, which can usually be ignored.

7. Conclusions and Future Work

In this paper, we addressed three major issues with cloud storage, namely reliability, security and performance. Instead of achieving data reliability using redundancy at the server side, we presented a reliable and secure distributed cloud data storage schema for end users. In our approach, we view multiple cloud storage services as virtual disks, and upload redundant data files into multiple cloud storages. The redundant data files are calculated using erasure codes techniques, which allow multiple failures of the cloud data services. By forming an optimal problem for calculating the number of checksum pieces, we can achieve the best space efficiency in our approach. Furthermore, we divide the user's data into pieces, and distribute them across multiple cloud services; therefore, no

single CSP can understand the uploaded user's data. As a result, our approach can effectively protect user data from unauthorized access in the cloud, and provide security at the software level for the end users. Finally, our experimental results show that besides the advantages of being secure and fault tolerant, our approach provides very good performance in both file uploading and downloading using the multithreading techniques, with the cost of minor overhead for encoding and decoding data.

For future work, we will investigate possible ways to automatically select a suitable number of data pieces based on the network condition, machine performance and the file size. We will consider other major aspects of cloud data, such as data integrity and confidentiality. For example, it would be feasible to adopt the digital signature technique to verify the integrity of the data stored in the cloud to ensure they were not altered by the service providers. Furthermore, when large cloud files are involved, the overhead for encoding and decoding may become a concern, especially for resource constrained devices such as a smartphone or a tablet. To improve the overall performance in this case, we need to look into more advanced techniques for erasure codes, such as regenerating codes and non-MDS codes [11, 25]. Finally, as a worthy future direction, we will attempt to integrate our approach with cloud-based big data analysis for reliable and secure massive datasets stored in the cloud.

Acknowledgments

We thank Ashok Peeta for his useful insights about erasure codes, and Kirti Dighe for her contribution to the implementation using Dropbox Core API. We also thank all anonymous referees for the careful review of this paper and the many suggestions for improvements they provided.

References

1. C. Kozyrakis, A. Kansal, S. Sankar and K. Vaid, Server engineering insights for large-scale online services, *IEEE Micro* **30**(4) (2010) 8-19.
2. E. K. Kolodner, S. Tal, D. Kyriazis, D. Naor *et al.*, A cloud environment for data-intensive storage services, in *Proc. of the IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, Athens, Greece, November 29-December 1, 2011, pp. 357-366.
3. D. Fitch and H. Xu, A RAID-based secure and fault-tolerant model for cloud information storage, *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)* **23**(5) (2013) 627-654.
4. M. Gagnaire, F. Diaz , C. Coti, *et al.*, *Downtime Statistics of Current Cloud Solutions*, International Working Group on Cloud Computing Resiliency, 2014. Retrieved on March 8, 2015 from http://iwgcr.org/category/downtime/
5. S. Yegulalp, Adobe creative cloud crash shows that no cloud is too big to fail, *InfoWorld*, May 16, 2014. Retrieved on March 7, 2015 from http://www.infoworld.com/article/2608200/cloud-computing/adobe - creative-cloud-crash-shows-that-no-cloud-is-too-big-to-fail.html
6. C. Talbot, Dropbox outage represents first major cloud outage of 2013, *Talkin'Cloud*, Jan 15, 2013. Retrieved on May 18, 2014 from http://talkincloud.com/cloud-storage/dropbox-outage-represents-first-major-cloud-outage-2013
7. Z. Whittaker, Amazon web services suffers outage, takes down Vine, Instagram, others with it, *ZDNet*, August 26, 2013. Retrieved on September 22, 2014 from http://www.zdnet.com/article/amazon-web-services-suffers-outage-takes-down-vine-instagram-others-with-it

8. M. Alia, S. U. Khana and A. V. Vasilakosb, Security in cloud computing: opportunities and challenges, *Information Sciences* **305**(1) (2015) 357-383.
9. D. Chen and H. Zhao, Data security and privacy protection issues in cloud computing, in *Proc. of the International Conference on Computer Science and Electronics Engineering (ICCSEE)*, Hangzhou, China, March 23-35, 2012, pp. 647-651.
10. M.-A. Russon, LinkedIn sues hackers for using Amazon cloud platform to make fake profiles, *International Business Times*, January 9, 2014. Retrieved on March 8, 2015 from http://www.ibtimes.co.uk/linkedin-sues-hackers-using-amazon-cloud-platform-make-fake-profiles-1431669
11. J. S. Plank, Erasure codes for storage systems: a brief primer, *Login: The USENIX Magzine* (*www.usenix.org*) **38** (6) (2013) 44-50.
12. H. Xu and D. Bhalerao, A reliable and secure cloud storage schema using multiple service providers, in *Proc. of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE 2015)*, Pittsburgh, USA, July 6-8, 2015, pp. 116-121.
13. S. B. Wicker and V. K. Bhargava (Eds.), *Reed-Solomon Codes and Their Applications*, The Institute of Electrical and Eltronics Engineers (IEEE), Inc., New York, June 1994.
14. 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 the 2012 USENIX Annual Technical Conference*, Boston, MA, USA, June 13-15, 2012, pp. 15-26.
15. L. B. Gomez, B. Nicolae, N. Maruyama, F. Cappello and S. Matsuoka, Scalable Reed-Solomon-based reliable local storage for HPC applications on IaaS clouds, in *Proc. of the 18th International Euro-Par Conference on Parallel Processing (Euro-Par '12)*, Rhodes, Greece, August 2012, pp. 313-324.
16. O. Khan, R. Burns, J. Plank and W. Pierce, Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads, in *Proc. of the 10th USENIX Conference on File and Storage Technologies (FAST-2012)*, San Jose, CA, USA, February 2012, pp. 20-33.
17. N. Santos, K. Gummadi and R. Rodrigues, Towards trusted cloud computing, in *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud09)*, San Diego, CA, June 15, 2009, Article No. 3.
18. K. Hwang and D. Li, Trusted cloud computing with secure resources and data coloring, *IEEE Internet Computing* **14** (5) (2010) 14-22.
19. C. Wang, Q. Wang, K. Ren and W. Lou, Ensuring data storage security in cloud computing, in *Proc. of the 17th International Workshop on Quality of Service (IWQoS)*, Charleston, SC, USA, July 13-15 2009, pp. 1-9.
20. D. Shue, M. J. Freedman and A. Shaikh, Performance isolation and fairness for multi-tenant cloud storage, in *Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, USA, October 8-10, 2012, pp. 349-362.
21. A. Zia and M. Khan, Identifying key challenges in performance issues in cloud computing, *International Journal of Modern Education and Computer Science (IJMECS)* **4** (10) (2012) 59-68.
22. F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland Mathematical Library, Amsterdam, London, New York, Tokyo, 1977.
23. C. K. Clarke, Reed-Solomon error correction, *R&D White Paper*, British Broadcasting Corporation, July 2002.
24. J. S. Plank, A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems, *Software - Practice & Experience* **27** (9) (1997) 995-1012.
25. J. Li and B. Li, Erasure coding for cloud storage systems: a survey, *Tsinghua Science and Technology* **18** (3) (2013) 259-272.